Author: Chris Mungall
Contributors: *<add names here>*
Type: White Paper
Status: Request for comments
Projects: Monarch, GO, all KGs, NMDC
Audience: ETL developers
Keywords: ETL, QC, Dashboards
Created: 2020
Updated: 2020
See also: SHACL, Shex and SPARQL for validation of ontologies and data graphs

Purpose

This doc serves two purposes:

1. [ambitious] specification for a generic ‘dasher’ dashboard QC framework for any data/knowledge-oriented project
2. [immediate/urgent] repository of design patterns for integrating semantic QC with ETL and provision of databases/ontology collections/other similar artefacts

Strawman Implementation

https://github.com/cmungall/dasher

Background

We have many projects that do the same thing, or need to do the same thing

- Assemble upstream sources into a standard form
- Run QC checks over sources
    - *Local*: checks internal correctness/consistency/quality of source. Can often be done on a ‘line by line’ basis of input files
    - *Global*: checks global correctness/consistency/quality of source (e.g ISS check for GO, or source + ontology(s) reasoning check)
- Make aggregate stats
- Produce a **dashboard**

    - Typical layout is a grid: Source x Metric
    - Cells are pass/fail/info/skip/SHOULD-NOT-violation
    - Can zoom in on source, metric, or individual source x metric result

These sometimes run over ‘ontologies’, sometimes ‘KGs’, sometimes ‘databases’. For our purposes here these distinctions are irrelevant, they are all databases with some amount of semantic smarts/’knowledge’/rules.

Examples for us:

- OBO http://obo-dashboard-test.ontodev.com/ (developed by Becky and Nico)
    - OWL is the standard representation (via an RDF syntax)
    - Semantic checks: reasoning (via robot)
    - Lexical and structural (sparql queries via robot)
    - Metadata checks (python)
    - Additional procedural (e.g. query GH API)
    - Note that individual ontology repos can run these in isolation which is nice
    - No good aggregate stats yet
    - See also: old central build pipeline
- GO Pipeline
    - http://current.geneontology.org/reports/gorule-report.html
    - QC:
        - Semantic (DL reasoning over OWL representation of GAFs + go-plus)
        - Structural/OO/CW (ShEx for GO-CAMs)
        - GO-Rules - mostly Python implementation
            - Line-by-line
            - Global
        - Additional ontology checks
- NMDC
    - No dashboard yet
    - NMDC json is the standard representation
    - All checks currently in JSON schema (compiled from biolinkml)
    - JSON is JSON-LD so could in principle leverage RDF tooling
- Monarch
    - https://qc.monarchinitiative.org/
        - Summary stats
        - Rules: https://qc.monarchinitiative.org/rules/monarch-rules.html#genes-in-more-than-one-taxa-fail-inferred
    - Scigraph and solr qc - https://data.monarchinitiative.org/qc/
    - Clique leader violations (two ids with same namespace declared clique leader)
    - https://github.com/monarch-initiative/release-utils (previously in analysis-sandbox)

    - Ran with jenkins data pipeline - https://github.com/monarch-initiative/jenkins/
    - rdf/turtle and nt is the standard representation for dipper, based on ns-predicate-ns facets
    - Neo4j is representation for final KG
- VFB
    - Sparql, shacl, cypher
- KG-COVID-19
    - No QC?
    - Aggregate stats/dashboard: https://knowledge-graph-hub.github.io/kg-covid-19-dashboard/
- KG-Hub
- KG-Microbe
    - No dashboard
- Translator uber-graph
    - No dashboard
    - Biolink model checks

As can be seen, each source has very particular checks that need to be executed, as well as generic ones. However, there are many similarities, and similar patterns that are encountered:

- In-memory vs against database
    - In memory: lightweight and predictable, good for local (row-by-row) checks
    - Against database: unpredictable query times?
    - Local checks: easier to report individual line numbers
        - But if sufficient provenance is provided by upstream ETL we can use this
    - Local checks ultimately limited
- Declarative vs procedural
    - Examples of declarative specs:
        - OWL (semantic, open world)
        - LinkML, ShEx, JSON-Schema, UML (structural, closed-world)
        - RegExs (lexical)
        - Queries in declarative language (sparql, cypher, sql, jq) that yield violations [can still be low level]
    - Examples of procedural:
        - Python code if/then/else etc
    - Pros/cons
        - Declarative is higher level, but sometimes less predictable performance, paradoxically sometimes harder to mentally reason over, possibility of frameworkitis
- Pipeline language vs Procedural
    - Declarative Pipeline languages:
        - Make
        - Snakemake

        - Nextflow
        - NOT CWL
    - Procedural:
        - Python code that has logic: for s in sources, download source, for c in checks, run c on s
- Infrastructure
    - Crons, Jenkins, GH actions
        - Where and how to to deploy? S3? Zenodo
- Sociological/policy
    - Consensus on what constitutes quality
        - Governance
        - Notions of severity. Is WARN vs ERROR sufficient? Grading?
        - What if a source disagrees?
    - Repair or report problem upstream?
    - Notions of severity
    - When should something be rejected vs accepted with warning?
        - Revert to previously cached copy?
    - Should an entire source be rejected if threshold is reached?

Requirements for a dashboard

The individual checks MUST have computable metadata

- ID/URI
- Name
- Text description
- Status (e.g. active/disabled)
- History metadata
- MUST vs SHOULD (always use ISO language)
- Scope
- group
- Computable spec, if possible
- …

Test results MUST be expressed in conformance with some LinkML datamodel

- Consider LinkML validation model (see OAK)
    - Based on SHACL
- Minimally:
    - test/check ID
    - Source ID
    - Result (enum of pass/fail/etc)
- Can be serialized as TSV/SQL/JSON/YAML

    - Doesnt’ matter, so long as conformant with datamodel

Vision

General toolkit: dasher

Primary input is a **registry**. There would be one registry per **aggregate KG**. Could be in ad-hoc yaml, void, etc. Agree on standard.

Example: http://purl.obolibrary.org/meta/ontologies.jsonld see also kg-hub

A central procedure would iterate over all **sources** in registry. Here we would have the usual logic for caching.

Each source would be in one of: json-ld, owl, rdf, kgx, kgtk. Not the responsibility of dasher to make these. Up to separate etl pipeline.

All sources loaded into a triplestore. Each source in a separate NG.

Some sources declare dependencies. These may (should?) be separately registered, or registration would be induced [TODO link to obo base documentation]

Semantic validation

All sources would have DL reasoning performed on: Source + dependencies [e.g. zfin gaf + go-plus.owl; omim.ttl + monarch.owl]; also DL reasoning on whole graph.

Entailed edges could be re-inserted [here we drift beyond scope of dasher but that’s OK]

Inconsistencies and incoherencies reported [need better explanations]

Implementation possibilities:
- dump from triplestore into OWL, use robot + appropriate reasoner
- No OWLAPI: directly on triplestore, e.g. whelk/arachne via go-plus

Structural Validation

Generic structural validation: many of the sparql checks built for robot are actually quite generic beyond the OBO-ontology use case. These could be trivially run on a triplestore (may need some awkward rewrite to restrict query to run on a single NG per time).

For AKG specific logic (e.g. GO-rules), declarative frameworks include

- ShEx
- JSON-Schema
- SPARQL rules
- LinkML (can compile to all of the above)

Of course at some point may need to bottom out in procedural code, but that is OK

I think blml can serve as an uber-language for most of the things we need here. It is more powerful than json-schema. It should also be trivial to compile subsets of blml semantics to SPARQL queries which will scale against triplestore [in principle this could be done for any blml serialization including SQL]. We can also compile blml tto ShEx and use PyShex against the triplestore.

Quantitative Metrics and Machine Learning

Perform aggregate statistisc - e.g number of nodes and edges by type. These are useful in their own right. Also as QC. E.g if number of human genes drops below 20k we know something is wrong.

Minimum threshold can be set manually based on prior knowledge. Or could be learned based on past patterns. Maybe a strict threshold is not always appropriate. Could be a probability something is wrong. E.g learn a distribution for each statistic

Dashboard

All of the above would generate instance report data using a standard schema. The central class in the schema would be something like MetricResult. It would have fields metric->Metric and source->Source, status->pass/fail/warn/etc, message->string[markdown?]

Simple mustache logic could build the HTML, make something nice looking a la obo or go

The dashboard could also have aggregate statistics, as these are closely tied to QC checks (and some QC checks may be about stats - e.g. minimum % classes with definitions). These could link to nice bar charts etc.

This would also incorporate some diffing functionality. E.g some AKGs may run dasher per release. We would want summary stats of changes. Some of these may themselves be QC checks (e.g. if # annotations drops precipitously then this is a red flag)

Implementation

One possibility is to build this into BGR https://github.com/balhoff/blazegraph-runner

Another is to loosely couple with BGR. Lightweight python framework is probably easiest for people.

While Makefiles are generally good (no need to write python logic for dependencies/triggering) needs to be balanced here against other factors.

May also want to consider groovy/Jenkins

Difficult Q is how dasher integrates with existing ETL pipelines? E.g rather than separate step, think of EQTQLQ per source. If ETL is in Python, and dasher is in Pypi, then relevant logic can be executed. Don't be monolithic. Modular pieces.

In fact there could be a single standalone registry_loader that is independent of any actual qc logic

Assumption here is a triplestore like BG or graphdb or virtuoso. James O is experimenting with a sqlite representation of RDF/OWL. Not clear if all sparql would need to be rewritten.

Glossary

- Registry
- Source
- Aggregate KG:
    - e.g.

-
    - go-plus + GAFs as OWL + GO-CAMs
    - Monarch RDF KG: dipper ttls plus ontologies (monarch.owl)
- NG: Named Graph
  - In a triplestore, each triple belongs to a single graph, ie quadstore
- ETL
- EQTQLQ: ETL + QC steps at each point

Appendix: Existing Approaches and Resources

Monarch QC

- https://qc.monarchinitiative.org/
- Scigraph and solr qc - https://data.monarchinitiative.org/qc/
  - Clique leader violations (two ids with same namespace declared clique leader)
  - https://github.com/monarch-initiative/release-utils (previously in analysis-sandbox)
  - Ran with jenkins data pipeline - https://github.com/monarch-initiative/jenkins/
  - Prints TSV of solr rows lost given a configurable percent drop
- rdf/turtle and nt is the standard representation for dipper, based on ns-predicate-ns facets, https://archive.monarchinitiative.org/beta/visual_reduction/
- Neo4j is representation for final KG

VFB QC

todo

Appendix: Seth's thoughts

I think it might be an interesting thought experiment to work backwards and see what steps could be taken to work from the local problems of projects back up to the more abstract issues laid out here. For example, what if we could work out a common error/warning/qc reporting format between a few projects? What if a common display widget or template could be made to consume that? What if a common qc descriptor could be used between projects (i.e. something like go rule)?

Ben’s thoughts.
I would approach this as a pipeline-building problem. The goal would be well defined input-output blocks that could be chained together. The code that did the work for each block could be different (e.g. procedural versus a db query) but the blocks ought to be able to be strung together. In an ideal world, you would then have a UI that would allow a user to assemble a pipeline given a list of potential processes. I think display/dashboard is important, but cart before horse.