Design:

1. Identification of data source, data can be structured or unstructured. For the initial experiment, CSV files are considered.
2. Systematic conversion of CSV file to RDF, otherwise RDF reasoner does not work.
3. Represent the quality of dataset along with statistics in terms of RDF(triple).
4. Model an ontology that can store the triple generated in step 3.
5. Design a knowledge base that does reasoning.
    a. Reasoners can point out the violation in triples at the abox level.

    For example, constraints can be written to evaluate data at the literal (object) level and some at the property (predicate) level.

    Ex,

    :type xsd:int //literal level
    :value 123
    :belongstoColumn C1

    C1 :hasType xsd:String //column level
        :uniqueness true
        ....

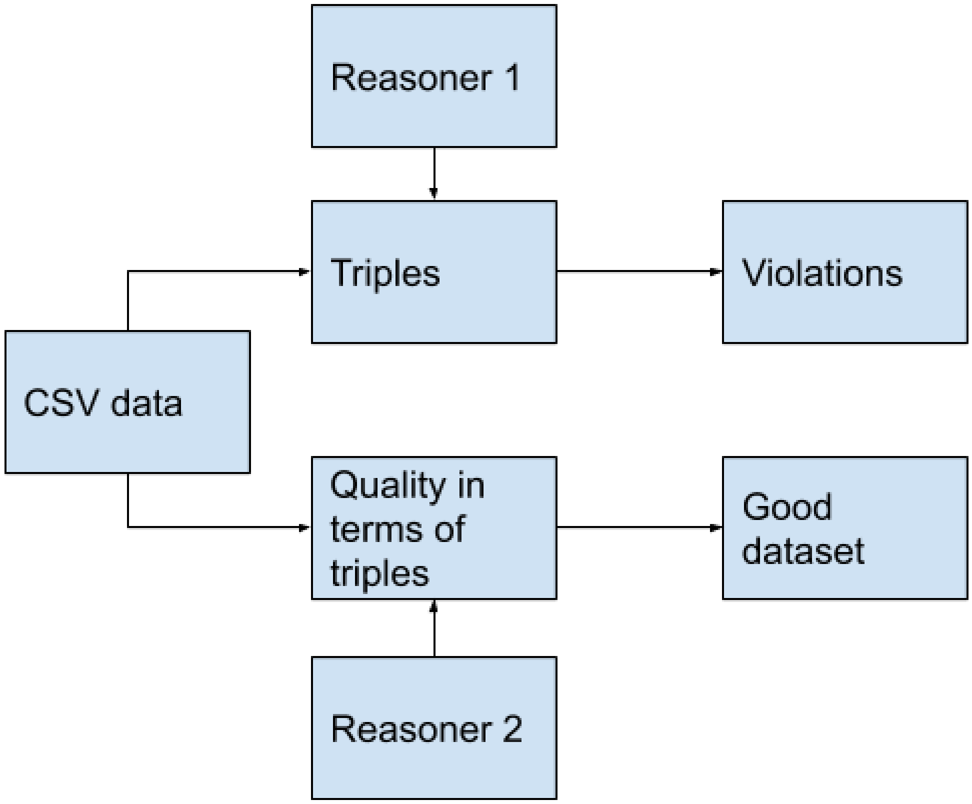
    b. Reasoners can gather better quality data if there are multiple files that exist in the same domain. (Ranking function)

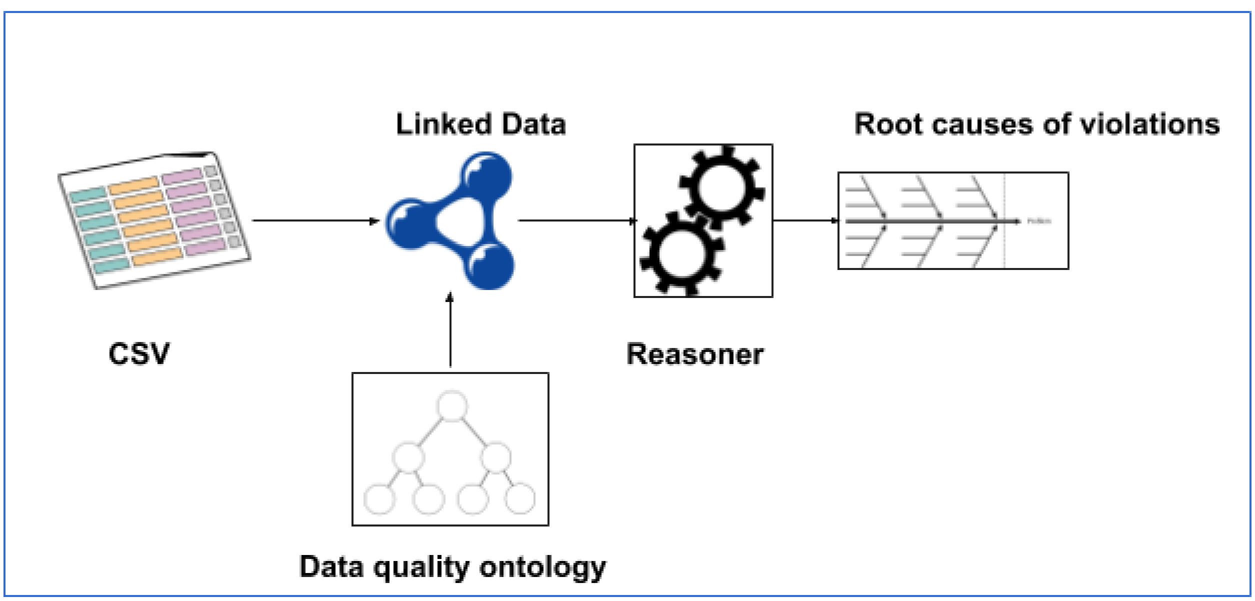
    For example, if ontology has quality information about multiple files that belong to the same domain, the reasoner can point out better quality dataset.

Example for the incompatible datatype, the column has string datatype mentioned whereas int value is present in the column.

:R0 :hasDatatype :string .
:R1 :hasDatatype :int .
:C1 :hasDatatype :string .
:C1 :hasInstance :list .
:list :hasMember :m1.
:list :hasMember :m2.
:m1 :hasDataType :int.
:m2 :hasDataType :int.
:m1 :hasValue 23.
:m2 :hasValue 12.

[r1: (?s :hasDatatype ?c) (?s :hasInstance ?list) (?list :hasMember ?m1) (?m1 :hasDatatype ?d) equal(?c,?d) -> (?m1 :incompatiblewith ?d)]


Reasoner 1
Triples
Violations
CSV data
Quality in terms of triples
Good dataset
Reasoner 2


Linked Data
Root causes of violations
CSV
Reasoner
Data quality ontology

Documentation:

Apache Jena supports 4 types of reasoners.

1. Transitive reasoner
2. RDFS reasoner
3. OWL reasoner
4. Generic rule reasoner

Procedure

1. Identify and configure a reasoner based on the requirement.
2. Apply the reasoner on data and schema(rules).
3. Access the inference.

Property	RDFS	OWL	Generic

Reading and attaching data and schema	`Model schema = RDFDataMgr.loadModel("file:data/rdfsDemoSchema.rdf");`
	
	`Model data = RDFDataMgr.loadModel("file:data/rdfsDemoData.rdf");`
		`Model schema = RDFDataMgr.loadModel("file:data/owlDemoSchema.owl");`
		
		`Model data = RDFDataMgr.loadModel("file:data/owlDemoData.rdf");`
			`Model model = ModelFactory.createDefaultModel();`
			
			`model.read( "dataset.ttl" );`

Configuring reasoner	`InfModel infmodel = ModelFactory.createRDFSModel(schema, data);`	`Reasoner reasoner = ReasonerRegistry.getOWLReasoner();`
		
		`reasoner = reasoner.bindSchema(schema);`
		
		`InfModel infmodel = ModelFactory.createInfModel(reasoner, data);`
			`Reasoner reasoner = new GenericRuleReasoner( Rule.rulesFromURL( "rules.n3" ) );`
			
			`InfModel infmodel = ModelFactory.createInfModel(reasoner, model);`

Validation	`ValidityReport validity = infmodel.validate();`
	`if (validity.isValid()) {`
	`    System.out.println("OK");`
	`} else {`
	`    System.out.println("Conflicts");`
	`    for (Iterator i =`
			Validation is not required as the program validated data against the defined rules.

```
validity.getReports(); i.hasNext(); ) {
        ValidityReport.Report report =
(ValidityReport.Report)i.next();
        System.out.println(" - " + report);
    }
}
```

RDFS entailments: RDFSrulereasoner has 3 flavours. Full, Default and Simple. Each supports various entailments of RDFS. Depending on our requirement we can set RDFS reasoner in Jena.
https://www.w3.org/TR/2004/REC-rdf-mt-20040210/#entail

literal generalization rule.

Rule Name	If E contains	then add
lg	uuu aaa lll .	uuu aaa _:nnn . where _:nnn identifies a blank node allocated to the literal lll by this rule.

literal instantiation rule.

Rule Name	If E contains	then add
gl	uuu aaa _:nnn . where _:nnn identifies a blank node allocated to the literal lll by rule lg.	uuu aaa lll .

RDF entailment rules

Rule Name	if E contains	then add
rdf1	uuu aaa yyy .	aaa `rdf:type rdf:Property .`
rdf2	uuu aaa lll . where lll is a well-typed XML literal .	_:nnn `rdf:type rdf:XMLLiteral .` where _:nnn identifies a blank node allocated to lll by rule lg.

RDFS entailment rules.

Rule Name	If E contains:	then add:
rdfs1	uuu aaa lll. where lll is a plain literal (with or without a language tag).	_:nnn `rdf:type rdfs:Literal .` where _:nnn identifies a blank node allocated to lll by rule rule lg.
rdfs2	aaa `rdfs:domain` xxx . uuu aaa yyy .	uuu `rdf:type` xxx .
rdfs3	aaa `rdfs:range` xxx . uuu aaa vvv .	vvv `rdf:type` xxx .
rdfs4a	uuu aaa xxx .	uuu `rdf:type rdfs:Resource .`
rdfs4b	uuu aaa vvv.	vvv `rdf:type rdfs:Resource .`
rdfs5	uuu `rdfs:subPropertyOf` vvv . vvv `rdfs:subPropertyOf` xxx .	uuu `rdfs:subPropertyOf` xxx .
rdfs6	uuu `rdf:type rdf:Property .`	uuu `rdfs:subPropertyOf` uuu .
rdfs7	aaa `rdfs:subPropertyOf` bbb . uuu aaa yyy .	uuu bbb yyy .
rdfs8	uuu `rdf:type rdfs:Class .`	uuu `rdfs:subClassOf rdfs:Resource .`
rdfs9	uuu `rdfs:subClassOf` xxx . vvv `rdf:type` uuu .	vvv `rdf:type` xxx .
rdfs10	uuu `rdf:type rdfs:Class .`	uuu `rdfs:subClassOf` uuu .
rdfs11	uuu `rdfs:subClassOf` vvv . vvv `rdfs:subClassOf` xxx .	uuu `rdfs:subClassOf` xxx .
rdfs12	uuu `rdf:type rdfs:ContainerMembershipProperty .`	uuu `rdfs:subPropertyOf rdfs:member .`
rdfs13	uuu `rdf:type rdfs:Datatype .`	uuu `rdfs:subClassOf rdfs:Literal .`

Some additional inferences which would be valid under the extensional versions of the RDFS semantic conditions.

ext1	uuu `rdfs:domain` vvv . vvv `rdfs:subClassOf` zzz .	uuu `rdfs:domain` zzz .
ext2	uuu `rdfs:range` vvv . vvv `rdfs:subClassOf` zzz .	uuu `rdfs:range` zzz .
ext3	uuu `rdfs:domain` vvv . www `rdfs:subPropertyOf` uuu .	www `rdfs:domain` vvv .
ext4	uuu `rdfs:range` vvv . www `rdfs:subPropertyOf` uuu .	www `rdfs:range` vvv .
ext5	`rdf:type rdfs:subPropertyOf` www . www `rdfs:domain` vvv .	`rdfs:Resource rdfs:subClassOf` vvv .
ext6	`rdfs:subClassOf rdfs:subPropertyOf` www . www `rdfs:domain` vvv .	`rdfs:Class rdfs:subClassOf` vvv .
ext7	`rdfs:subPropertyOf rdfs:subPropertyOf` www . www `rdfs:domain` vvv .	`rdf:Property rdfs:subClassOf` vvv .
ext8	`rdfs:subClassOf rdfs:subPropertyOf` www . www `rdfs:range` vvv .	`rdfs:Class rdfs:subClassOf` vvv .
ext9	`rdfs:subPropertyOf rdfs:subPropertyOf` www . www `rdfs:range` vvv .	`rdf:Property rdfs:subClassOf` vvv .

rdfD1	ddd `rdf:type rdfs:Datatype .` uuu aaa "sss"^^ddd .	_:nnn `rdf:type` ddd . where _:nnn identifies a blank node allocated to "sss"^^ddd by rule rule lg.

Suppose it is known that two lexical forms sss and ttt map to the same value under the datatype denoted by ddd; then the following rule applies:

rdfD2	ddd `rdf:type rdfs:Datatype .` uuu aaa "sss"^^ddd .	uuu aaa "ttt"^^ddd .

Suppose it is known that the lexical form sss of the datatype denoted by ddd and the lexical form ttt of the datatype denoted by eee map to the same value. Then the following rule applies:

rdfD3	ddd `rdf:type rdfs:Datatype .` eee `rdf:type rdfs:Datatype .` uuu aaa "sss"^^ddd .	uuu aaa "ttt"^^eee .

xsd 1a	uuu aaa "sss".	uuu aaa "sss"^^`xsd:string .`
xsd 1b	uuu aaa "sss"^^`xsd:string .`	uuu aaa "sss".

OWL reasoners: RDFS reasoner + additional
OWL reasoner supports 3 configurations: full, mini and micro. None of them supports all the features of OWL-DL. This can be implemented with the help of external reasoners such as Pellet, Racer or Fact.

Constructs	Supported by	Notes
rdfs:subClassOf, rdfs:subPropertyOf, rdf:type	all	Normal RDFS semantics supported including meta use (e.g. taking the subPropertyOf subClassOf).
rdfs:domain, rdfs:range	all	Stronger if-and-only-if semantics supported
owl:intersectionOf	all	
owl:unionOf	all	Partial support. If C=unionOf(A,B) then will infer that A,B are subclasses of C, and thus that instances of A or B are instances of C. Does not handle the reverse (that an instance of C must be either an instance of A or an instance of B).
owl:equivalentClass	all	
owl:disjointWith	full, mini	
owl:sameAs, owl:differentFrom, owl:distinctMembers	full, mini	owl:distinctMembers is currently translated into a quadratic set of owl:differentFrom assertions.
Owl:Thing	all	
owl:equivalentProperty, owl:inverseOf	all	
owl:FunctionalProperty, owl:InverseFunctionalProperty	all	
owl:SymmetricProperty, owl:TransitiveProperty	all	
owl:someValuesFrom	full, (mini)	Full supports both directions (existence of a value implies membership of someValuesFrom restriction, membership of someValuesFrom implies the existence of a bNode representing the value). Mini omits the latter "bNode introduction" which avoids some infinite closures.
owl:allValuesFrom	full, mini	Partial support, forward direction only (member of a allValuesFrom(p, C) implies that all p values are of type C). Does handle cases where the reverse direction is trivially true (e.g. by virtue of a global rdfs:range axiom).
owl:minCardinality, owl:maxCardinality, owl:cardinality	full, (mini)	Restricted to cardinalities of 0 or 1, though higher cardinalities are partially supported in validation for the case of literal-valued properties. Mini omits the bNodes introduction in the minCardinality(1) case, see someValuesFrom above.
owl:hasValue	all	

Builtin Primitives: Builtin primitives available in Jena which can be implemented via BuiltinRegistry.
https://jena.apache.org/documentation/inference/#rules

Builtin	Operations
isLiteral(?x) notLiteral(?x) isFunctor(?x) notFunctor(?x) isBNode(?x) notBNode(?x)	Test whether the single argument is or is not a literal, a functor-valued literal or a blank-node, respectively.
bound(?x...) unbound(?x..)	Test if all of the arguments are bound (not bound) variables
equal(?x,?y) notEqual(?x,?y)	Test if x=y (or x != y). The equality test is semantic equality so that, for example, the xsd:int 1 and the xsd:decimal 1 would test equal.
lessThan(?x, ?y), greaterThan(?x, ?y) le(?x, ?y), ge(?x, ?y)	Test if x is <, >, <= or >= y. Only passes if both x and y are numbers or time instants (can be integer or floating point or XSDDateTime).
sum(?a, ?b, ?c) addOne(?a, ?c) difference(?a, ?b, ?c) min(?a, ?b, ?c) max(?a, ?b, ?c) product(?a, ?b, ?c) quotient(?a, ?b, ?c)	Sets c to be (a+b), (a+1) (a-b), min(a,b), max(a,b), (a*b), (a/b). Note that these do not run backwards, if in `sum` a and c are bound and b is unbound then the test will fail rather than bind b to (c-a). This could be fixed.
strConcat(?a1, .. ?an, ?t) uriConcat(?a1, .. ?an, ?t)	Concatenates the lexical form of all the arguments except the last, then binds the last argument to a plain literal (strConcat) or a URI node (uriConcat) with that lexical form. In both cases if an argument node is a URI node the URI will be used as the lexical form.
regex(?t, ?p) regex(?t, ?p, ?m1, .. ?mn)	Matches the lexical form of a literal (?t) against a regular expression pattern given by another literal (?p). If the match succeeds, and if there are any additional arguments then it will bind the first n capture groups to the arguments ?m1 to ?mn. The regular expression pattern syntax is that provided by java.util.regex. Note that the capture groups are numbered from 1 and the first capture group will be bound to ?m1, we ignore the implicit capture group 0 which corresponds to the entire matched string. So for example `regexp('foo bar', '(.*) (.*)', ?m1, ?m2)` will bind `m1` to `"foo"` and `m2` to `"bar"`.
now(?x)	Binds ?x to an xsd:dateTime value corresponding to the current time.
makeTemp(?x)	Binds ?x to a newly created blank node.
makeInstance(?x, ?p, ?v) makeInstance(?x, ?p, ?t, ?v)	Binds ?v to be a blank node which is asserted as the value of the ?p property on resource ?x and optionally has type ?t. Multiple calls with the same arguments will return the same blank node each time - thus allowing this call to be used in backward rules.
makeSkolem(?x, ?v1, ... ?vn)	Binds ?x to be a blank node. The blank node is generated based on the values of the remain ?vi arguments, so the same combination of arguments will generate the same bNode.
noValue(?x, ?p) noValue(?x ?p ?v)	True if there is no known triple (x, p, *) or (x, p, v) in the model or the explicit forward deductions so far.
remove(n, ...) drop(n, ...)	Remove the statement (triple) which caused the n'th body term of this (forward-only) rule to match. Remove will propagate the change to other consequent rules including the firing rule (which must thus be guarded by some other clauses). In particular, if the removed statement (triple) appears in the body of a rule that has already fired, the consequences of such rule are retracted from the deducted model. Drop will silently remove the triple(s) from the graph but not fire any rules as a consequence. These are clearly non-monotonic operations and, in particular, the behaviour of a rule set in which different rules both drop and create the same triple(s) is undefined.
isDType(?l, ?t) notDType(?l, ?t)	Tests if literal ?l is (or is not) an instance of the datatype defined by resource ?t.
print(?x, ...)	Print (to standard out) a representation of each argument. This is useful for debugging rather than serious IO work.
listContains(?l, ?x) listNotContains(?l, ?x)	Passes if ?l is a list which contains (does not contain) the element ?x, both arguments must be ground, can not be used as a generator.
listEntry(?list, ?index, ?val)	Binds ?val to the ?index'th entry in the RDF list ?list. If there is no such entry the variable will be unbound and the call will fail. Only usable in rule bodies.
listLength(?l, ?len)	Binds ?len to the length of the list ?l.
listEqual(?la, ?lb) listNotEqual(?la, ?lb)	listEqual tests if the two arguments are both lists and contain the same elements. The equality test is semantic equality on literals (sameValueAs) but will not take into account owl:sameAs aliases. listNotEqual is the negation of this (passes if listEqual fails).
listMapAsObject(?s, ?p ?l) listMapAsSubject(?l, ?p, ?o)	These can only be used as actions in the head of a rule. They deduce a set of triples derived from the list argument ?l : listMapAsObject asserts triples (?s ?p ?x) for each ?x in the list ?l, listMapAsSubject asserts triples (?x ?p ?o).
table(?p) tableAll()	Declare that all goals involving property ?p (or all goals) should be tabled by the backward engine.
hide(p)	Declares that statements involving the predicate p should be hidden. Queries to the model will not report such statements. This is useful to enable non-monotonic forward rules to define flag predicates which are only used for inference control and do not "pollute" the inference results.

Timeline:
August 1st week: converting CSV into RDF
August 2nd week: data quality ontology & reasoner
August 3rd week: Reasoner
August 4th week: onto and reasoner

Name	Available?	External mapping is required	Additional info	Year
RDF Extension	No	Yes	Project does not exist in Open Refine github.	----
RDF123	No	Yes	https://link.springer.com/chapter/10.1007/978-3-540-88564-1_29	2008
XLWrap	Yes	Yes	Mapping is given in Trig which is similar to N3.	2012-2013
Tarql	Yes	No	SPARQL queries need to be written by looking at the columns. Column binding to URI and datatype conversion need to be done explicitely using SPARQL queries. Ex, http://tarql.github.io/	2017
Anzo for Excel	-	-	Not available	
TopBraid	--	--	Paid	
TabLinker	Yes	--	Manual conversion of annotated excel	
NOR2O	--	--	Not available	
Esxcel2rdf	--	--	EXE file. Supported only in Windows.	
Spread2RDF	Yes	Yes	Written in Ruby	2013
Sheet2RDF	Yes	Yes	Mapping is written in PEARL. PEARL is declarative language which is used to write mappings of excel to	2015

			RDF	
RDFToOnto	-	-		
http://explore.dublincore.net/competency_index/d2695955/s2696022/s2696080/s2742465/	Some more links			