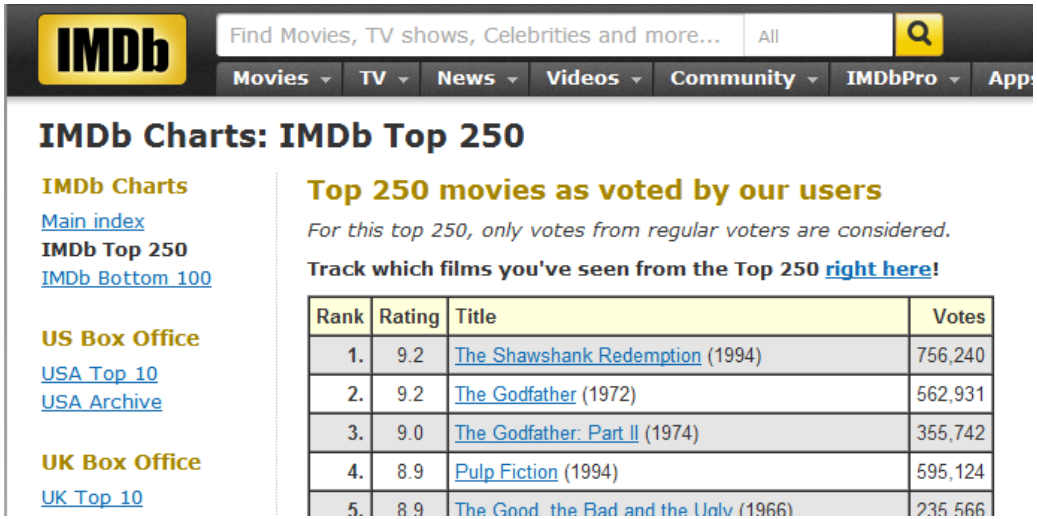
Movies Web App

Data Layer

Outline

- Movies and Producers
- Standard approaches
- Semantics and Data
- Data Flow

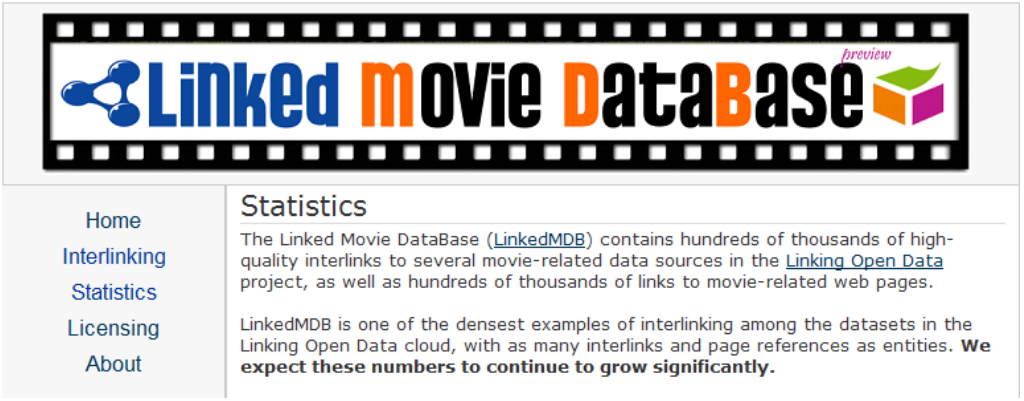
Inspiration



It would be great to pull these together to figure out who the most important producers are

It'd be even better if it could keep itself updated as new movies come out

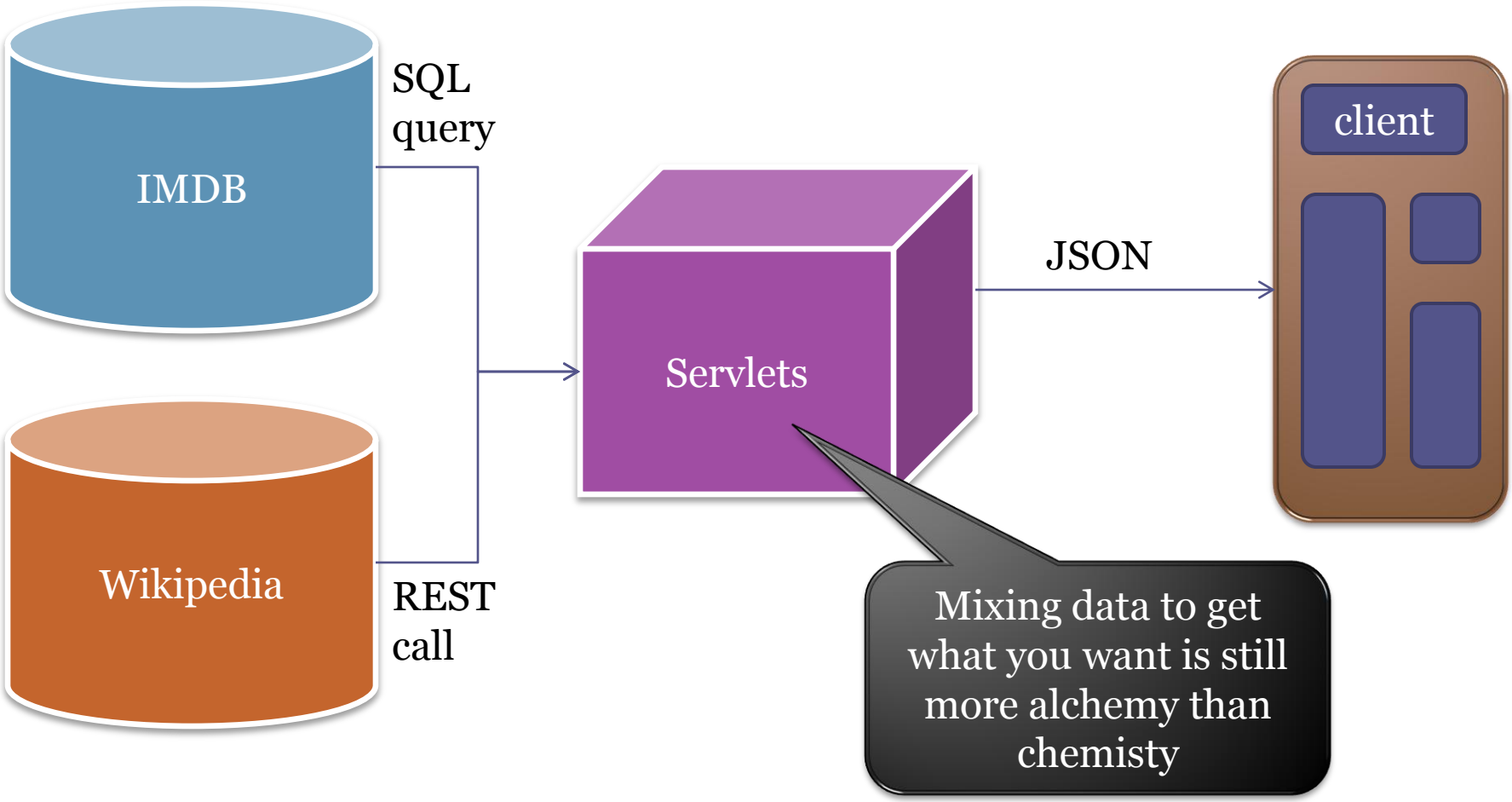
Opportunity



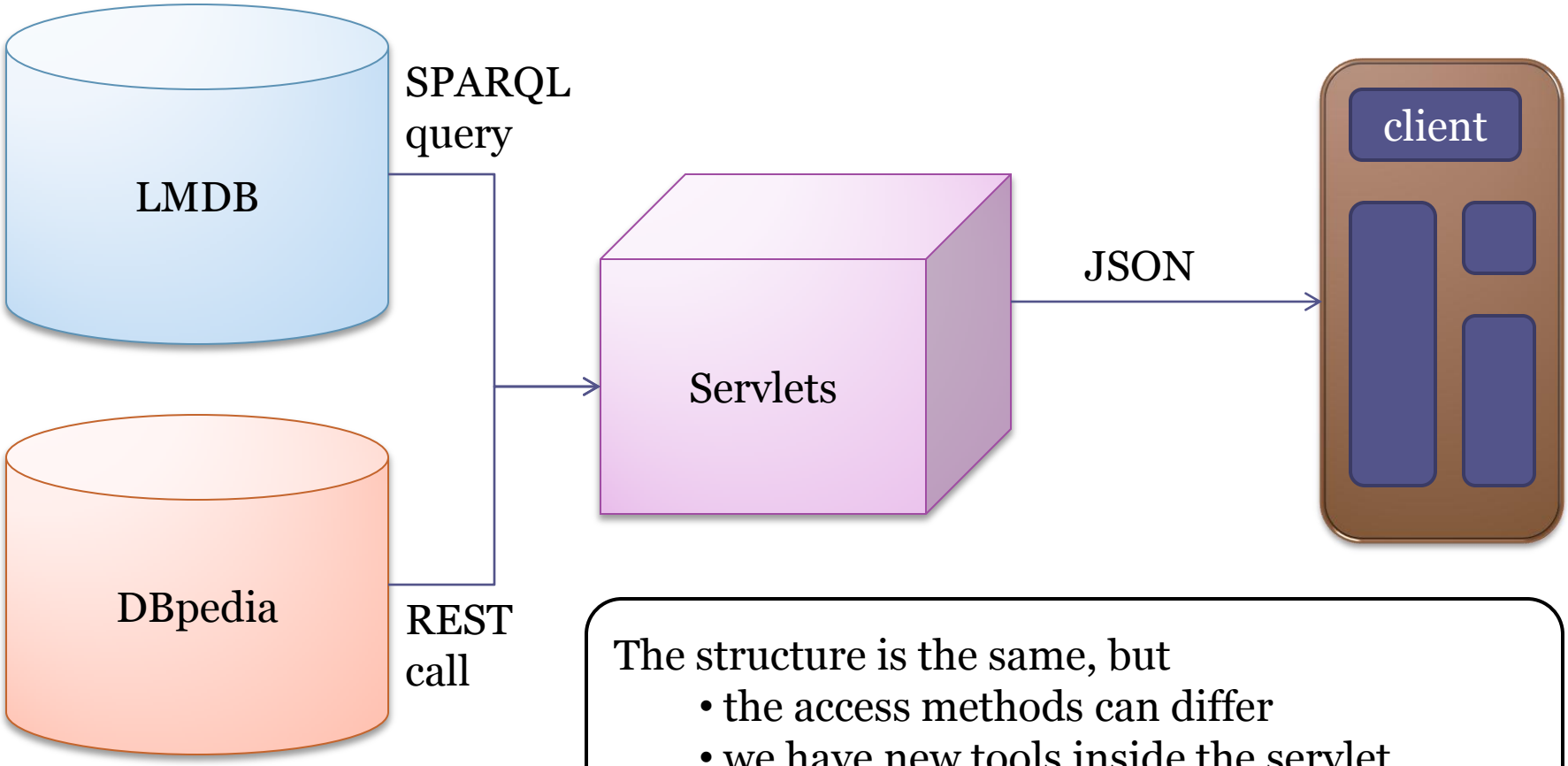
Fortunately, there are open data sources with similar content

Oddly, they insist on being Semantic Web sources

The Standard Approach



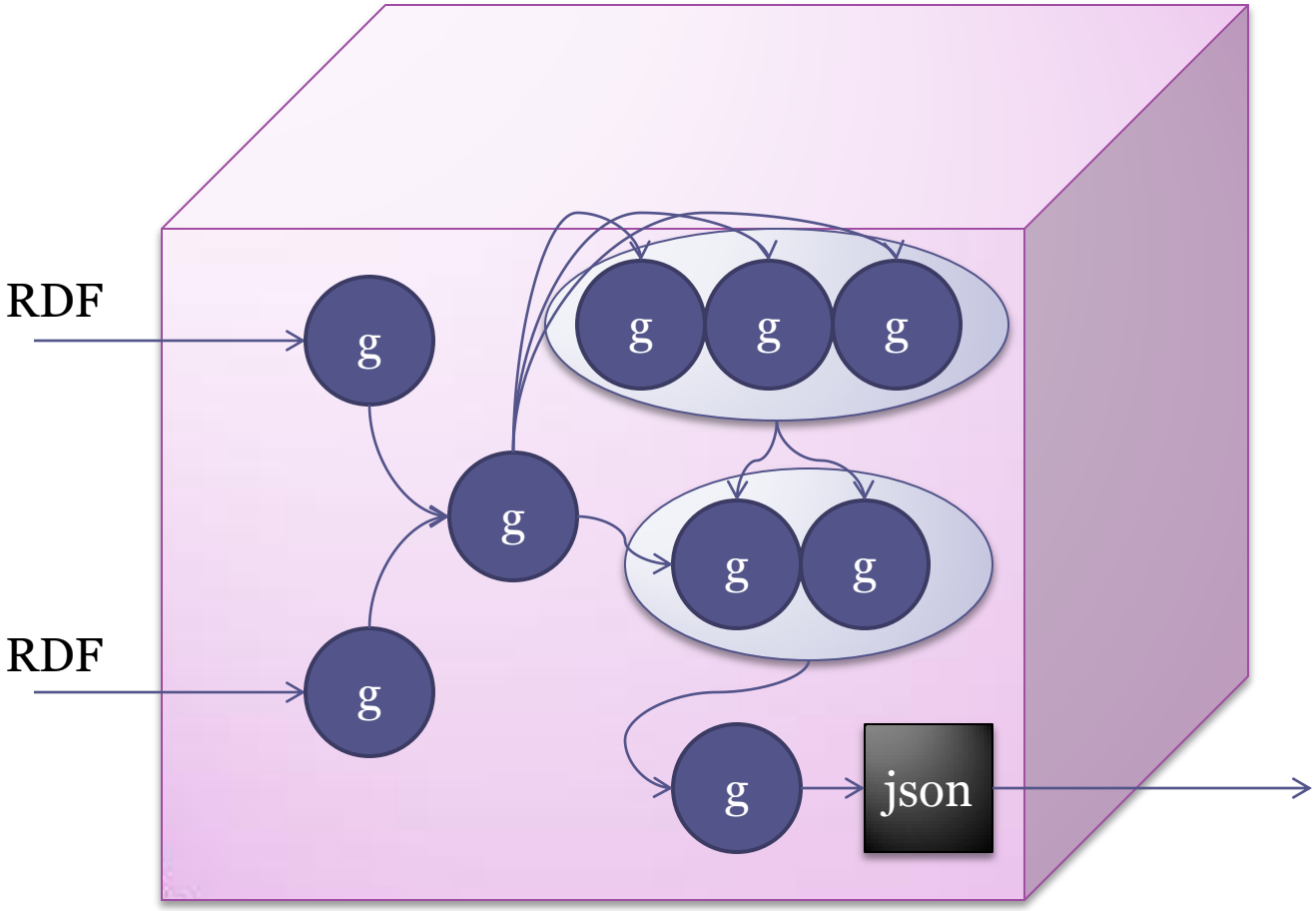
Semantic Alchemy



The structure is the same, but

- the access methods can differ
- we have new tools inside the servlet

Semantics is odd

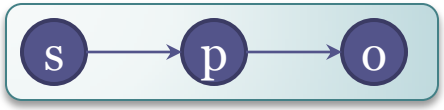


We tend not to rip through XML, JSON, or POJOs

Rather, semantic processing is a cascade of graph transformations

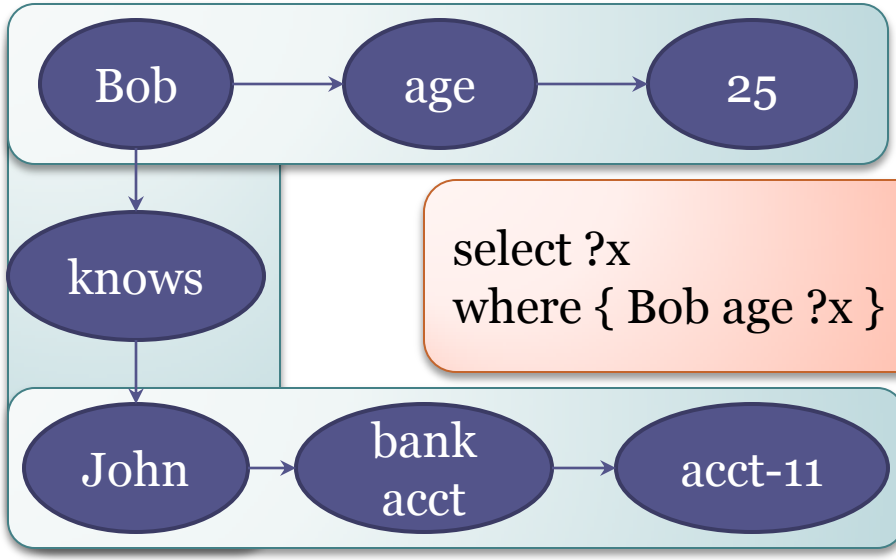
At the end, we still respond to the client with a sensible JSON object

Standardization in Semantics



An RDF fact:

- subject
- predicate
- object




```
select ?x
where { Bob age ?x }
```

A SPARQL query against three facts

It's not down to a science, but there are standards to lean on

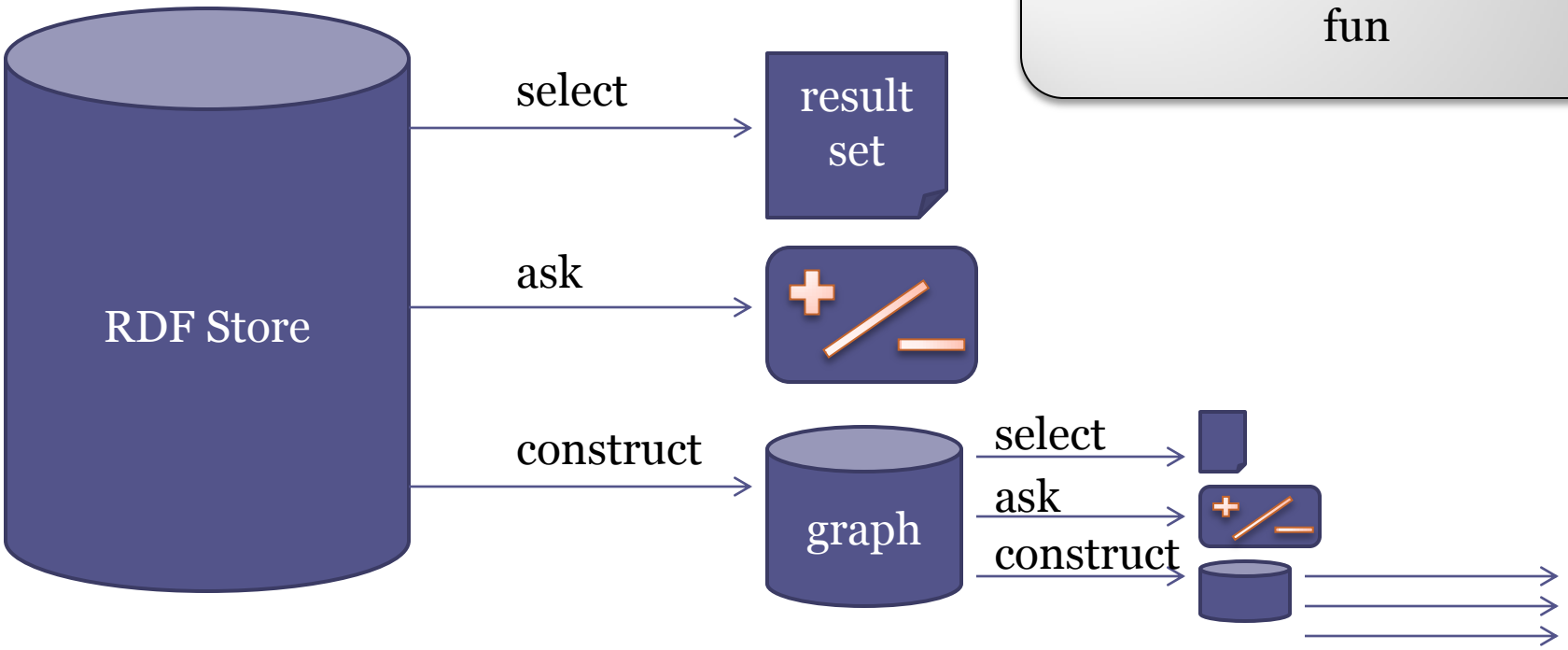
RDF – data language

SPARQL – query language

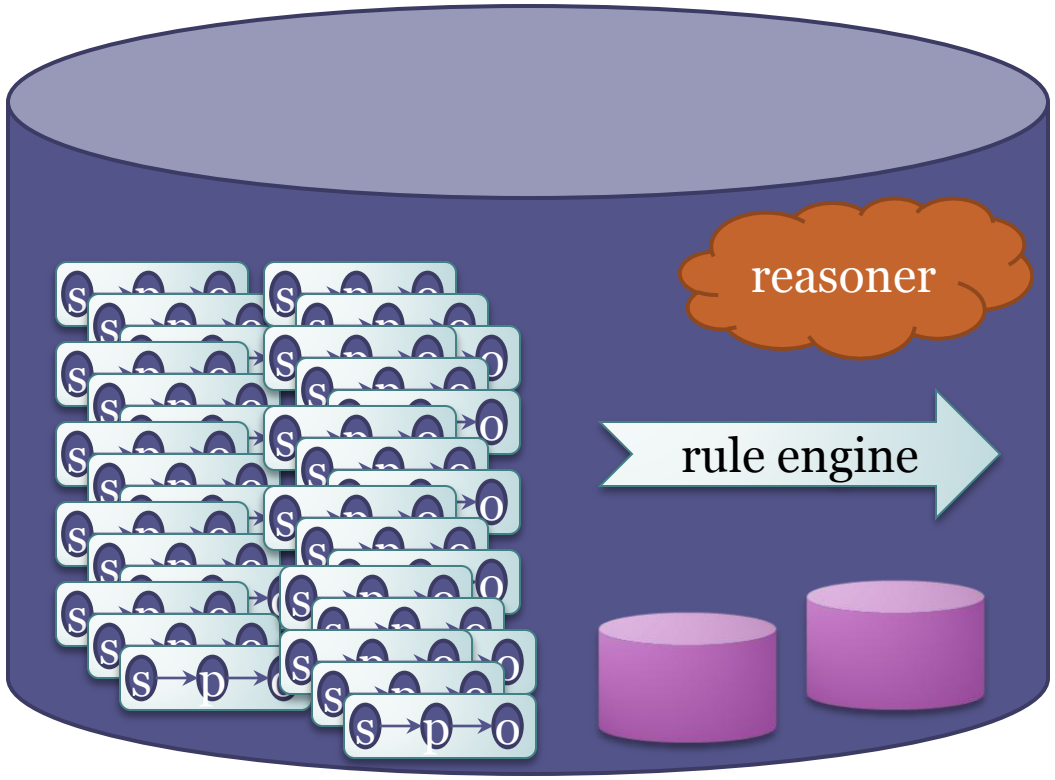
Jena – de facto standard Java library for semantics

SPARQL is special

Those constructs give you composition, which is a lot of fun



RDF graphs are special too



Primarily they are nests of RDF facts

But we also get to put in

- reasoners
- rule engines
- sub-graphs

Semantics and Movies

Who are the top 100 movie producers?

- Name
- # of films
- Gravity
- Experience
- Street Cred
- Most frequent film rating
- Associated Production Companies

Some of these properties require a bit of elucidation:

Gravity: How well does the producer attract stars?

Experience: How much in the way of quality film-making has this producer given us?

Street Cred: How many directors and writers are willing to work with the producer?

Tools

- Clojure
- Seabass
- RDF/RDFS (Turtle)
- SPARQL
- JSON

```
(defn lcd [n t]
  (loop [a t]
    (cond (> (* a a) n) n
          (= (rem n a) 0) a
          (= 2 a) (recur 3)
          true (recur (+ a 2)))))
(defn prime? [x] (= x (lcd x 2)))
(defn next-prime [x]
  (loop [a (+ x 2)]
    (cond (= x 2) 3
          (prime? a) a
          true (recur (+ a 2)))))
```

- A Lisp on the JVM
- Handles concurrency very well
- I like it
- Everything done here could be done in Java, Scala, Groovy, Python, Ruby, or even C++

Tools

- Clojure
- Seabass
- RDF/RDFS (Turtle)
- SPARQL
- JSON

```
(def q "    select ?x ?y ?z
            where {
                ?x <http://ex.org/foo> ?y .
                ?z <http://ex.org/bar ?y . }")
(bounce q (build "data/my-ont.ttl" "data/your-ont.owl"))
```

- A Clojure library I wrote around Jena
- Simplified interface:
  - **build** – create a model from local files, remote files via URL, and other models
  - **bounce** – get results from a model with a SELECT query
  - **pull** – get a model from a model with a CONSTRUCT query
  - **ask** – get a boolean from a model with an ASK query
  - **stash** – save a model as a local file as N-triples

Tools

- Clojure
- Seabass
- RDF/RDFS (Turtle)
- SPARQL 1.1
- JSON

```
nhl:hometeam    rdfs:domain nhl:Game ;
                rdfs:range nhl:Team ;
                rdfs:subPropertyOf nhl:team .

nhl:awayteam    rdfs:domain nhl:Game ;
                rdfs:range nhl:Team ;
                rdfs:subPropertyOf nhl:team .
```

```
(def team-names "
prefix : <http://www.nhl.com/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
construct { ?team :name ?name }
{ select distinct ?team ?name
 { ?game :team ?team . ?team rdfs:label ?name }
}
")
```

- RDF/S is neat
- Turtle is the only RDF syntax
- SPARQL 1.1 is likewise neat

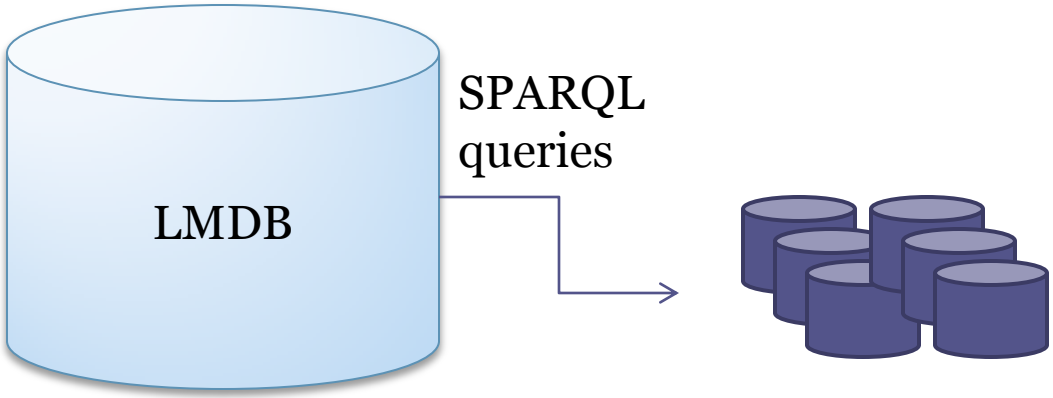
Tools

- Clojure
- Seabass
- RDF/RDFS (Turtle)
- SPARQL
- JSON

```
game: {
      awayteamid: 8,
      awayteamname: "Montreal Canadiens",
      hometeamname: "Toronto Maple Leafs",
      plays: {
            play: [
                  {
                        sweater: "37",
                        localtime: "7:29 PM",
                        xcoord: 87,
                        desc: "Tim Brent HIT on Josh Gorges",
                        teamid: 10,
                        strength: 701,
                        pid: 8470283,
                        formalEventId: "TOR51",
                        period: 1,
                        type: "Hit",
                        p3name: "",
                        eventid: 51,
                        p2name: "Josh Gorges",
                        ycoord: 36,
```

- The very popular data format for web things
- Better than XML in every way
- It's the native object syntax for Javascript

Data Flow

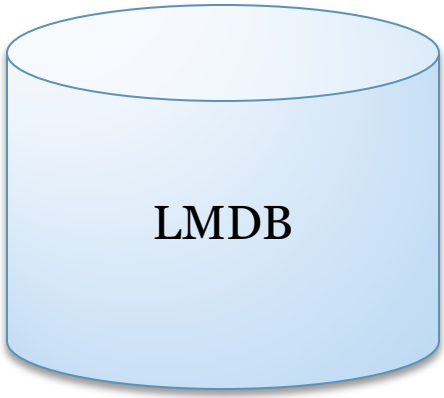


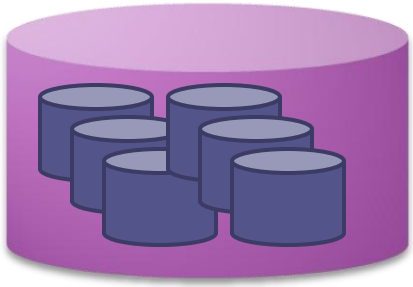
We want a flow that can be run every morning and update our RDF store.

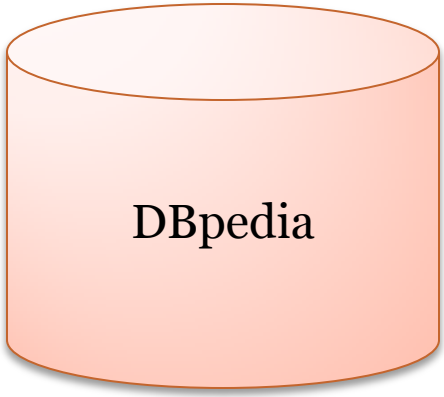
We want to be good netizens, so we shouldn't ask for more than we need.

```
;;   which films did they produce?
(def lmdb-0 (str prefix
                 "construct {?x :produced ?f . ?f :title ?t . ?f :date ?d} "
                 "{?f lmdb:producer ?x . ?f dc:title ?t . ?f dc:date ?d}"))
;;   which production companies have they worked with?
(def lmdb-1 (str prefix
                 "construct {?x :worked-with ?y . ?y a :Production-Company} "
                 "{?f lmdb:producer ?x . ?f lmdb:production_company ?y}"))
;;   which writers have they worked with?
(def lmdb-2 (str prefix
                 "construct {?x :worked-with ?y . ?y a :Writer} "
                 "{?f lmdb:producer ?x . ?f lmdb:writer ?y}"))
```

Data Flow



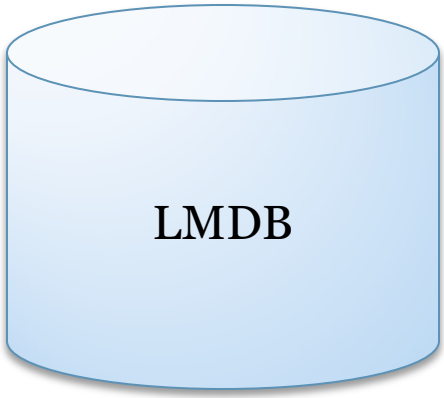


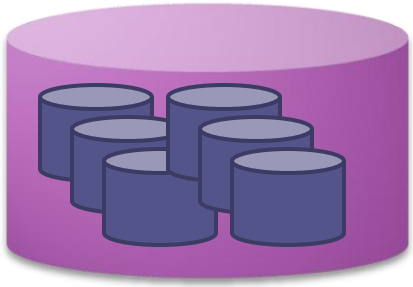


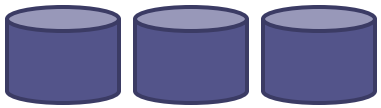
Another nice thing
about RDF is that it is
- very easy –
to combine graphs...

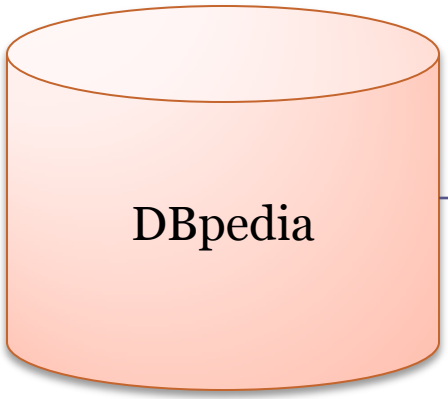
just smoosh them
together

Data Flow









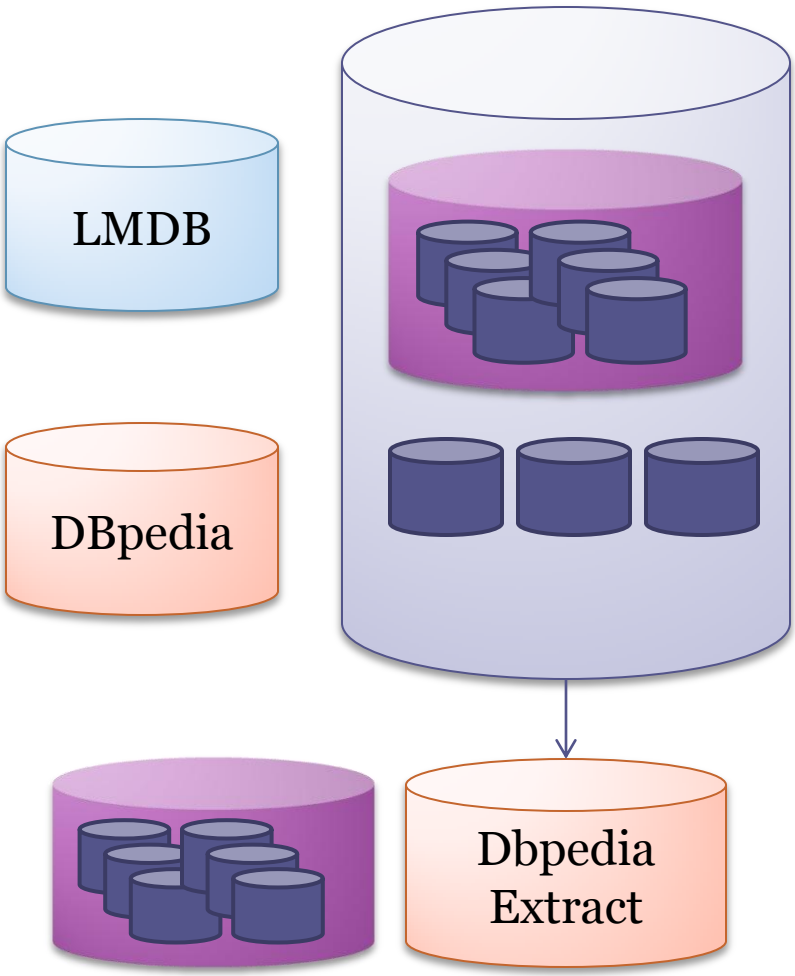
SPARQL queries

Up to now, we've ignored Dbpedia

That's because we're using it as icing on our LMDB cake

```
;;   what were these films' budgets? (dbp)
(def dbp-0 (str prefix
                "construct {?f :dbp-budget ?b} "
                "{?f dbp:budget ?b}"))
;;   what were these films' grosses? (dbp)
(def dbp-1 (str prefix
                "construct {?f :dbp-gross ?g} "
                "{?f dbp:gross ?g}"))
;;   who were these films' stars? (dbp)
(def dbp-2 (str prefix
                "construct {?f :dbp-starring ?a . ?a :name ?name} "
                "{?f dbp:starring ?a . ?a rdfs:label ?name}"))
```

Data Flow




```
;;   the integration queries clean up the ties betw the sources
(def int-0 (str prefix
                "construct {?f :budget ?v} "
                "{ select ?f ?v "
                "  { ?f owl:sameAs/:dbp-budget ?b "
                "    bind (xsd:float(?b) as ?v) "
                "  } "
                "}"))
(def int-1 (str prefix
                "construct {?f :gross ?v} "
                "{ select ?f ?v "
                "  { ?f owl:sameAs/:dbp-gross ?g "
                "    bind (xsd:float(?g) as ?v) "
                "  } "
                "}"))
(def int-2 (str prefix
                "construct {?f :starring ?a} "
                "{?f owl:sameAs/:dbp-starring ?a}"))
```

Integration Time

LMDB uses some Dbpedia resources

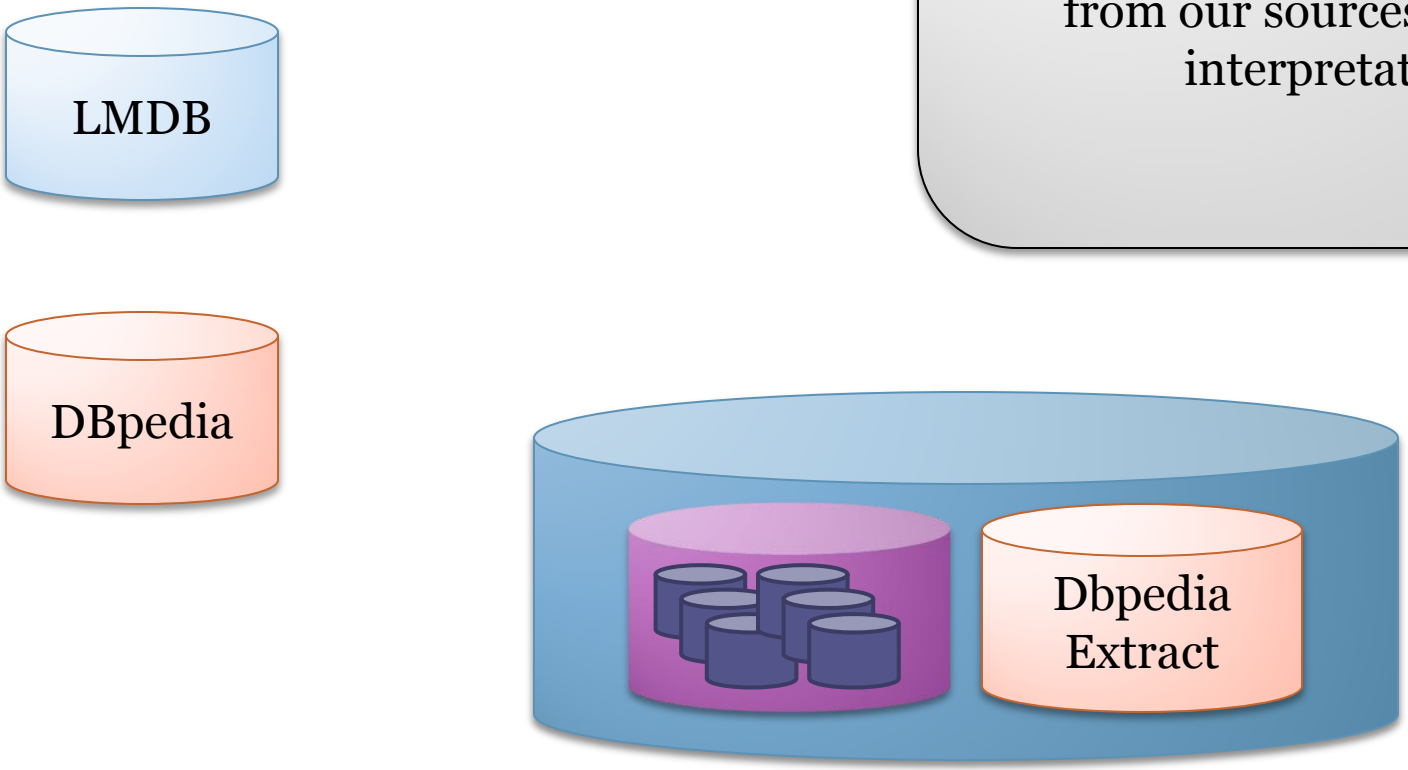
So we want to pull out of our Dbpedia results only those resources that we're using

We'll have to make a temporary copy of our LMDB extract to do this

Data Flow

Our Integration Graph is done

This represents the base data set drawn from our sources, with minimal interpretation done.



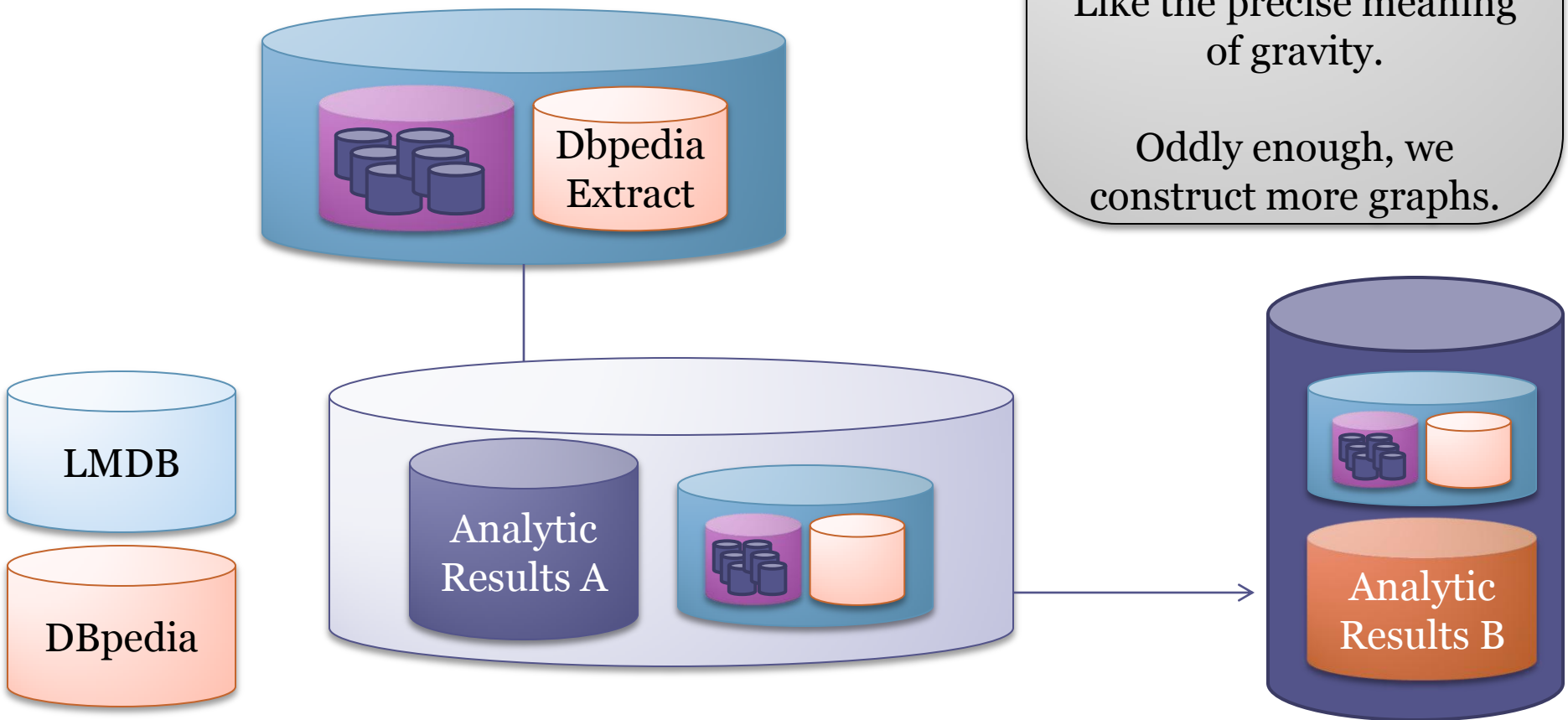
Data Flow

Now we're on to analysis.

This is where we add lots of new semantics, providing our own interpretation of things.

Like the precise meaning of gravity.

Oddly enough, we construct more graphs.

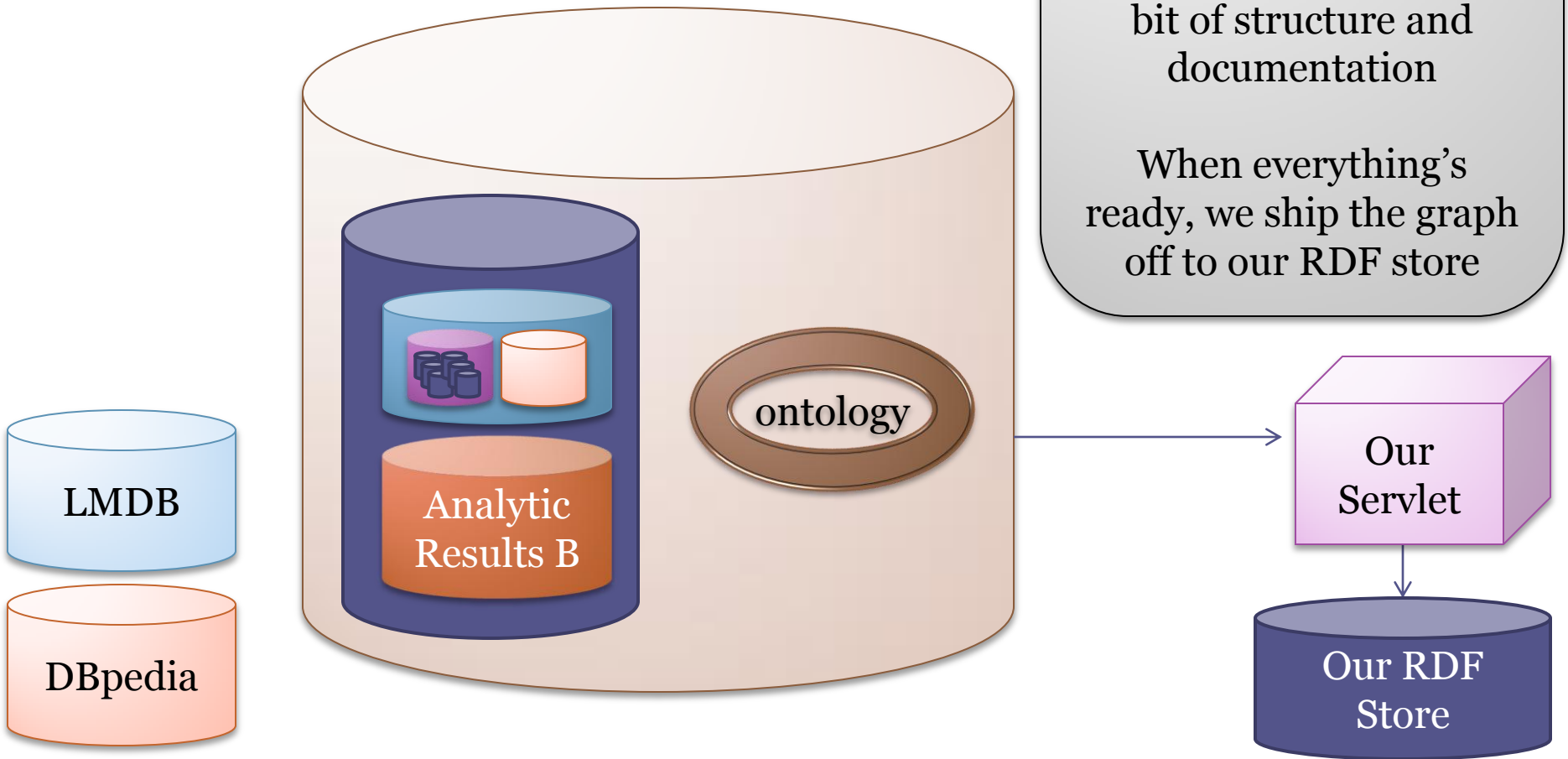


Data Flow

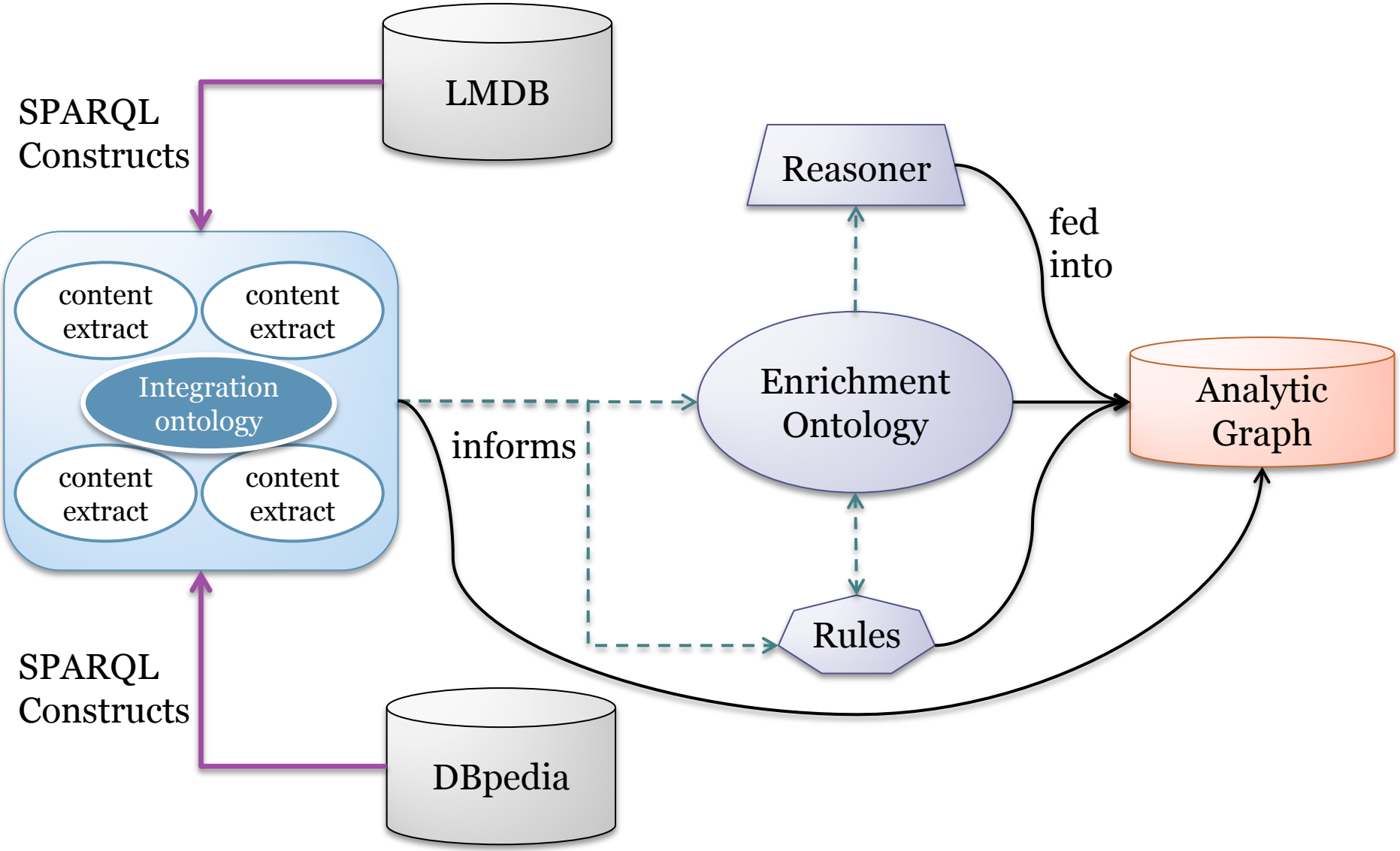
The only thing we have left to do is package up our results.

We add an ontology to give our final dataset a bit of structure and documentation

When everything's ready, we ship the graph off to our RDF store



Overview of the Data Flow



What's the payoff for this oddity?

- The algorithms are fairly resilient
  - Only W3C standards are used, so rewriting in Python, Ruby, Groovy, or even R is somewhat straightforward
  - The approach works for pretty much any RDF endpoint you want to touch
  - You can encapsulate all this semantic logic in a single place, so the rest of your code looks normal
  - It's fun

Questions?