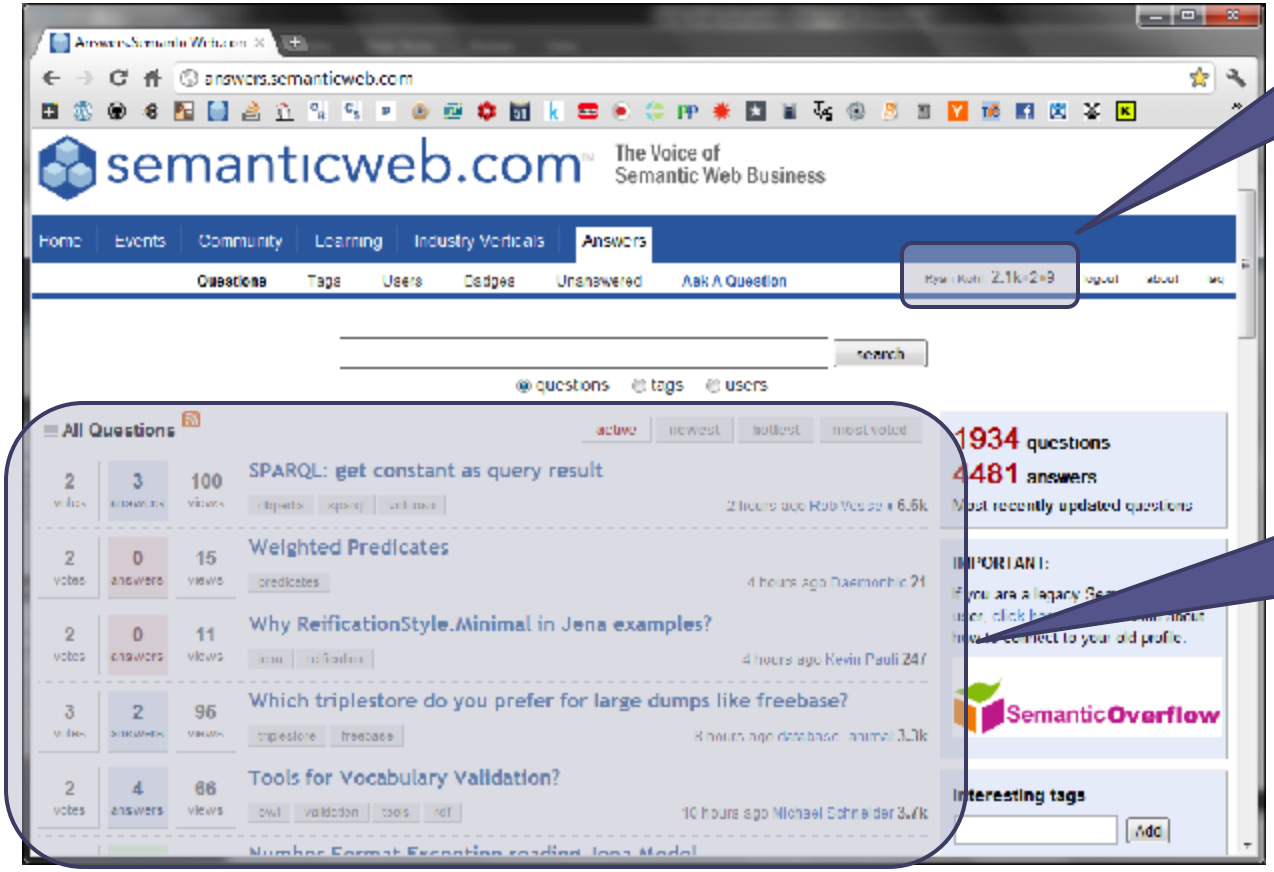
NHL Web App

Data Layer

Outline

- Questions and Games
- Seabass
- Construction
- Flow
- OWL?

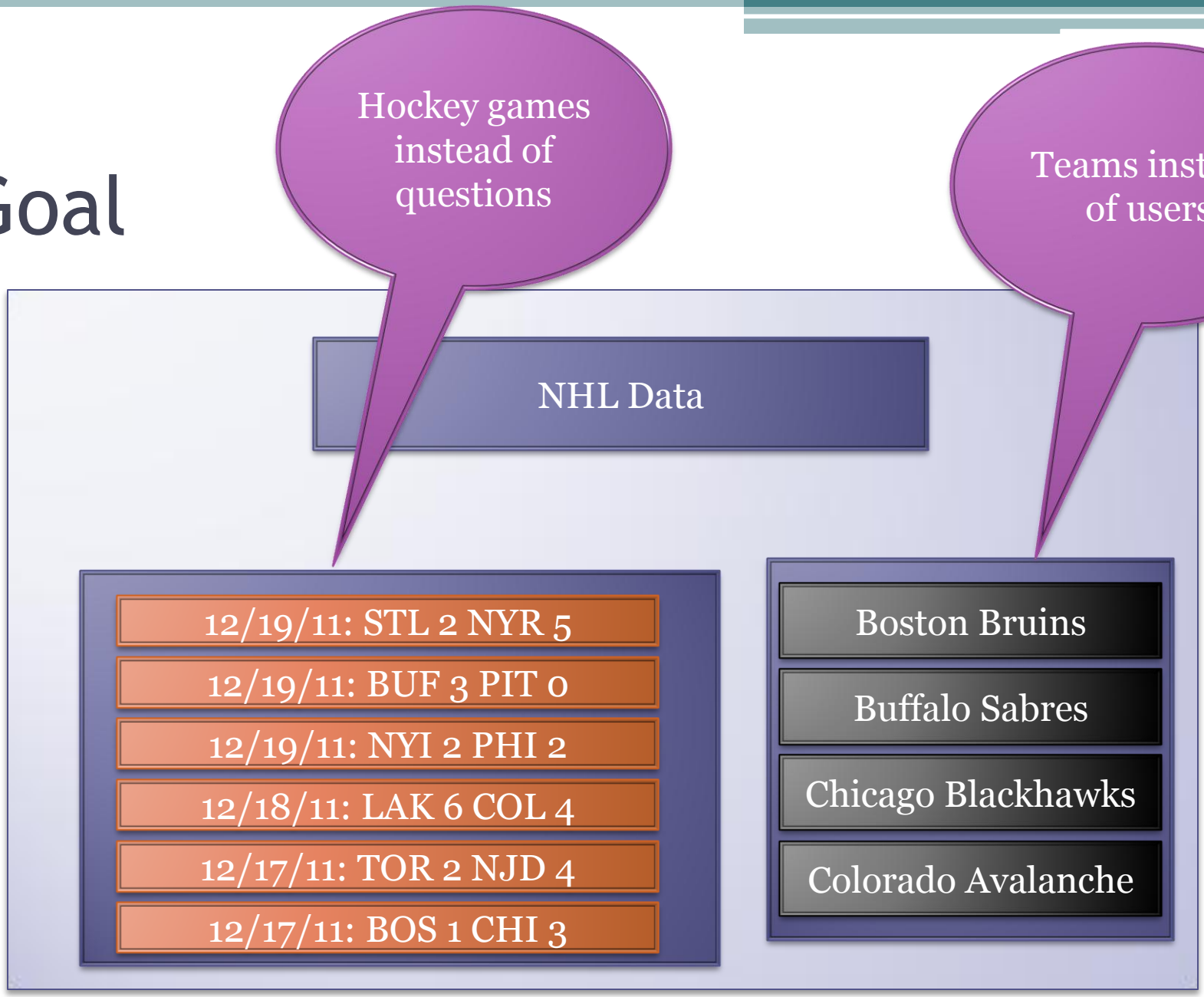
Inspiration



Badges, Stats, and Trends for users

Questions supplied by users

Goal


Hockey games instead of questions
Teams instead of users
NHL Data
12/19/11: STL 2 NYR 5
12/19/11: BUF 3 PIT 0
12/19/11: NYI 2 PHI 2
12/18/11: LAK 6 COL 4
12/17/11: TOR 2 NJD 4
12/17/11: BOS 1 CHI 3
Boston Bruins
Buffalo Sabres
Chicago Blackhawks
Colorado Avalanche

Initial Requirements

- Summary Stats for every game of the 2010-11 season
- Badges for teams based on game performance
- Metrics that compare teams
- Badges associated with both teams and the games in which they were earned

Game 23: NYR 4 BUF 0
Hits: 79 Shots: 64 PIM: 35

Shutout!
Opposing team scores no goals

Teams that hit PHI the most:
PIT – 78.3
NYI – 67.1
FLA – 66.8

Game 23:
NYR – Shutout!

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

The NHL publishes data

- Every single NHL regular-season game since 2007 is published on the web.

- It's in JSON

- You get an event-level record:
  - goals
  - shots
  - hits
  - penalties

- Each event has all kinds of info:
  - players involved by ID
  - the time of event
  - X,Y coordinates
  - event type
  - …


```
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,
```

Plan of Attack

1. Download the NHL JSON Data
2. Convert it into RDF
3. Extract required info:
   - Badges
   - Game Summaries
   - Team Comparisons
4. Transform results into JSON
5. Publish JSON on web

Semantic Workflow

1. Build a model
2. Pull out triples
3. Build a new model
4. repeat…

N. Make a result set

N+1. Transform into JSON

```
(def m (build "data/game-1.nt"))

(def p1 (pull construct-A m))
(def p2 (pull construct-B m))

(def n (build p1 p2 m)


(def r (bounce query-C n))

(def j (json-str r))
```

Workflow Parts

- Lots of SELECT and CONSTRUCT queries

429 lines of queries

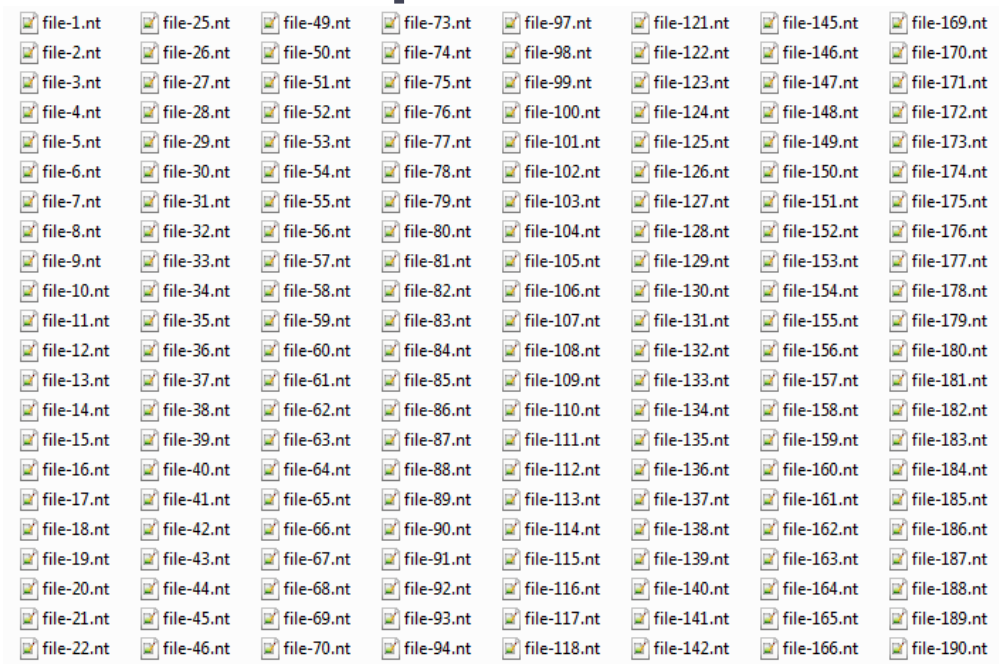
- A little code

199 lines of code

  - functions to download NHL data
  - functions to transform NHL data into RDF
  - functions to execute semantic workflow

Workflow Details – set-up

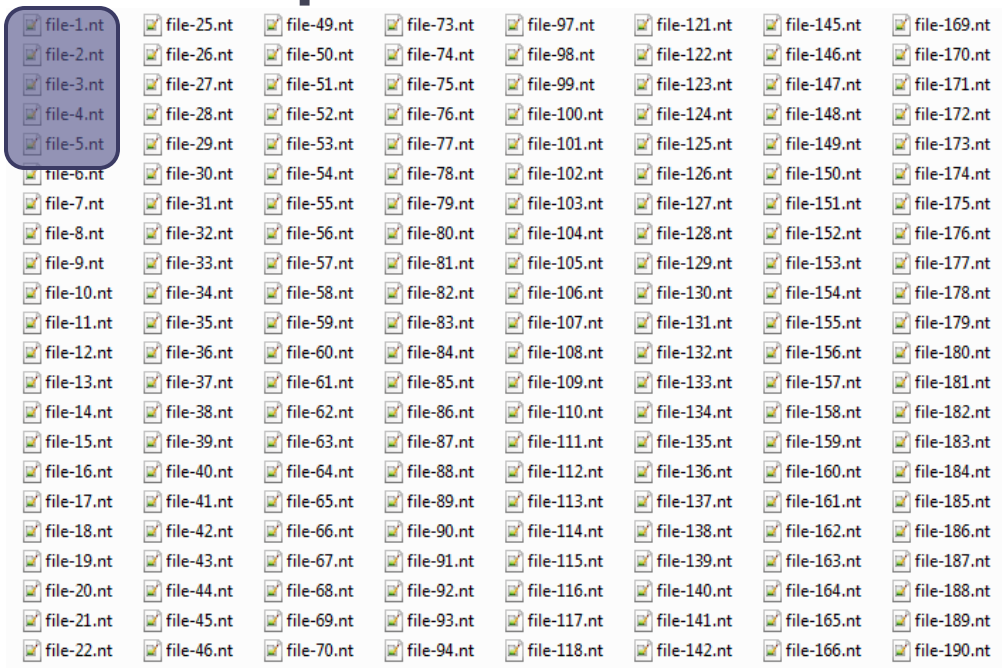
- A full season has 1230 games
- We break it up into groups of 5
- The whole workflow is one big map-reduce job:
  - map: workflow process
  - reduce: **build** (from seabass)



Workflow Details – step 1

Take the batch of 5 games and combine them into a single model

(def M (**build** “file-1.nt” “file-2.nt” “file-3.nt” “file-4.nt” “file-5.nt”))

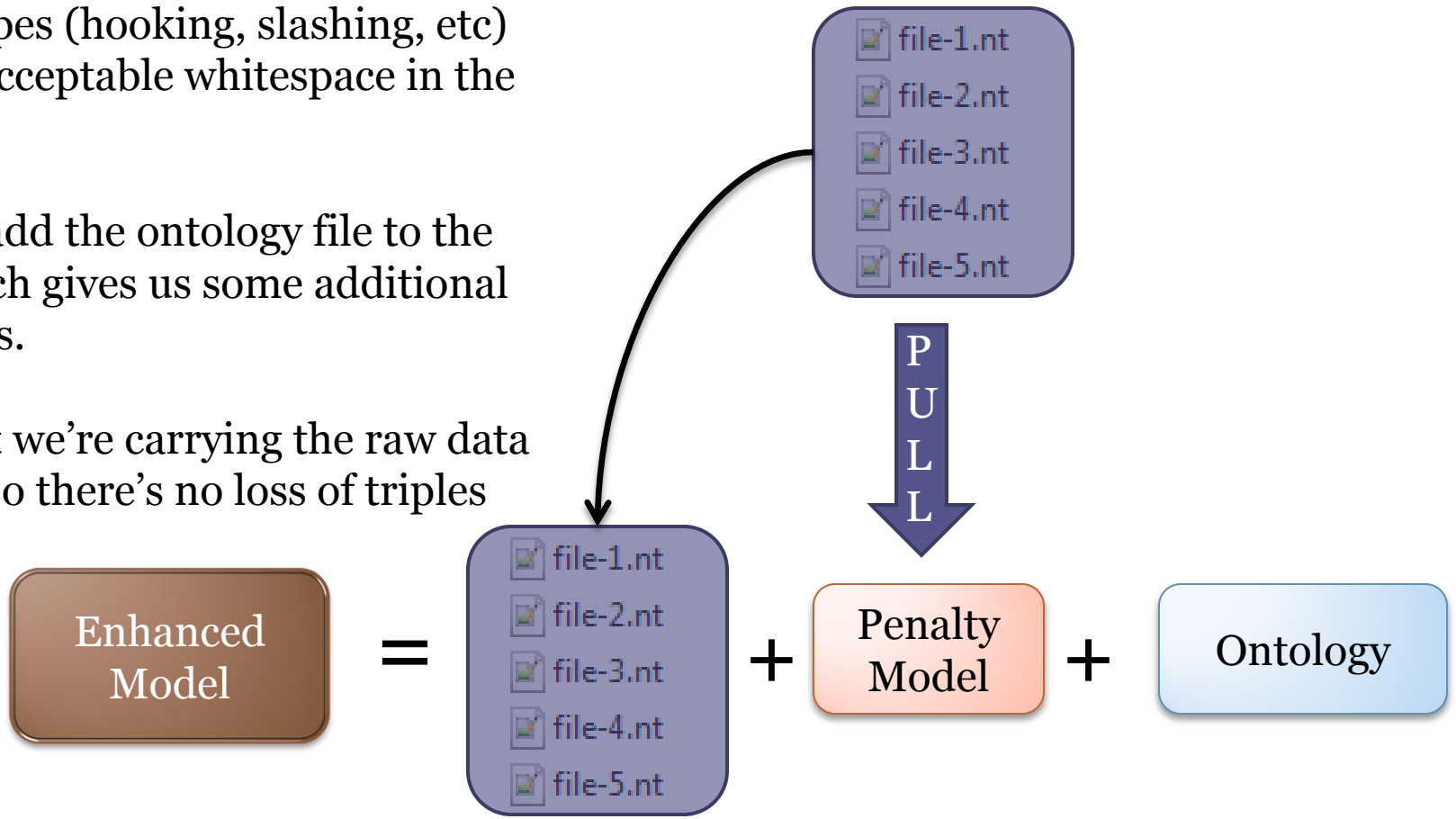


Workflow Details – step 2

We pull out facts about penalties, whose types (hooking, slashing, etc) have unacceptable whitespace in the raw data.

We also add the ontology file to the mix, which gives us some additional semantics.

Note that we're carrying the raw data with us, so there's no loss of triples



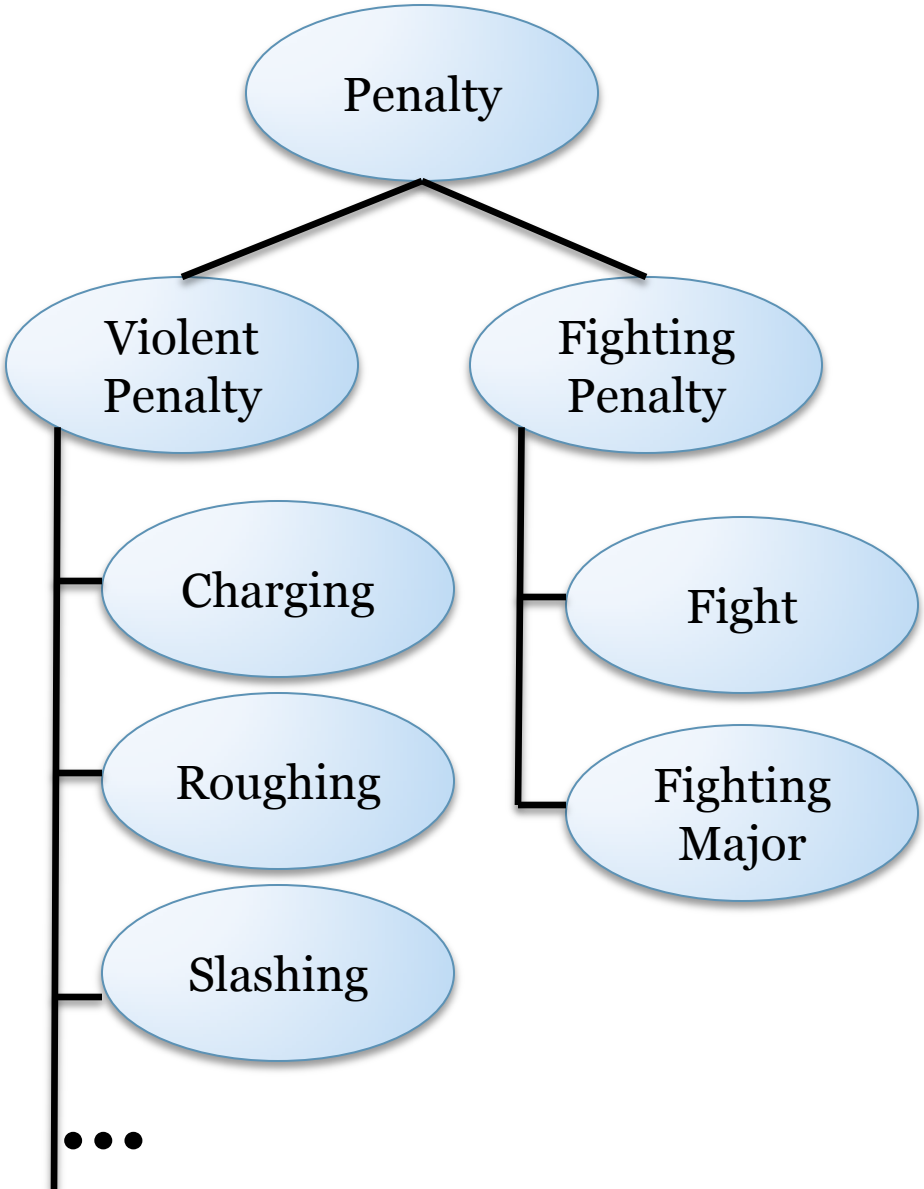
Wait, ontology?

The ontology does three things:

1. Defines nhl:hometeam
   nhl:awayteam
   as relations holding between games and teams

2. Defines a small Penalty taxonomy (more of a categorization)

3. Defines the penalty minutes associated with each kind of penalty



And how about that semantic pull?

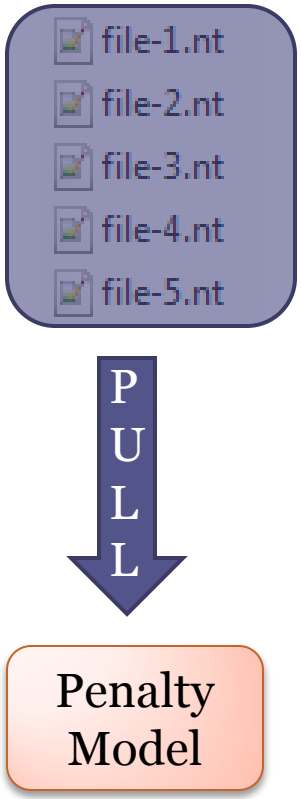
This is essentially a SPARQL CONSTRUCT query:

```
(def boarding "
   prefix : <http://www.nhl.com/>
   construct { ?e a :Boarding }
    { ?e a nhl:Penalty . ?e nhl:desc ?d . FILTER regex(?d, '.*(BOARDING).*') }
")
```

(def penaltyModel (**build**
(**pull** boarding M)
(**pull** slashing M)
(**pull** hooking M)))

Of course, we pull more penalty facts out than those associated with these three.

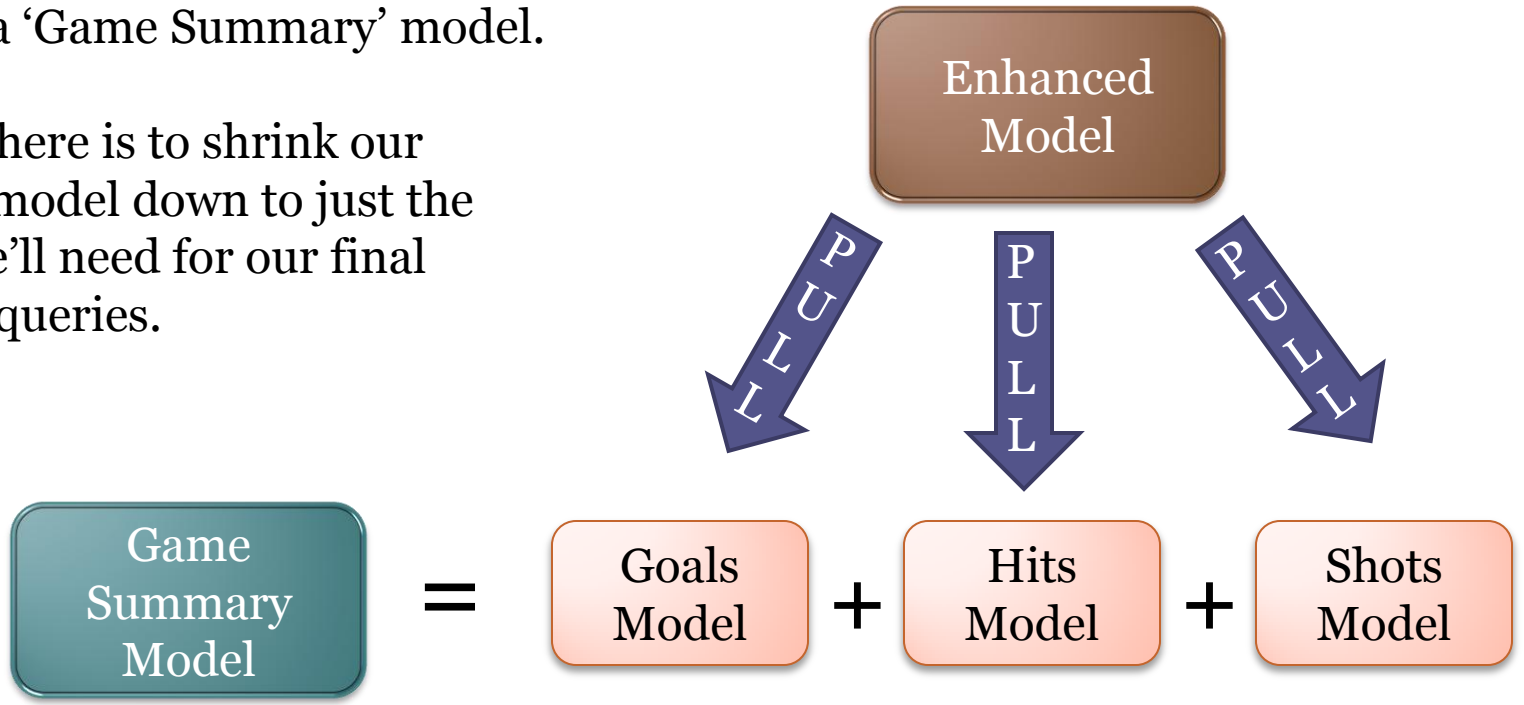
In fact, we use a Clojure function to take away a lot of the repetitiveness.



Workflow Details – step 3

Similar to step two, we're now creating a 'Game Summary' model.

The goal here is to shrink our working model down to just the triples we'll need for our final SELECT queries.

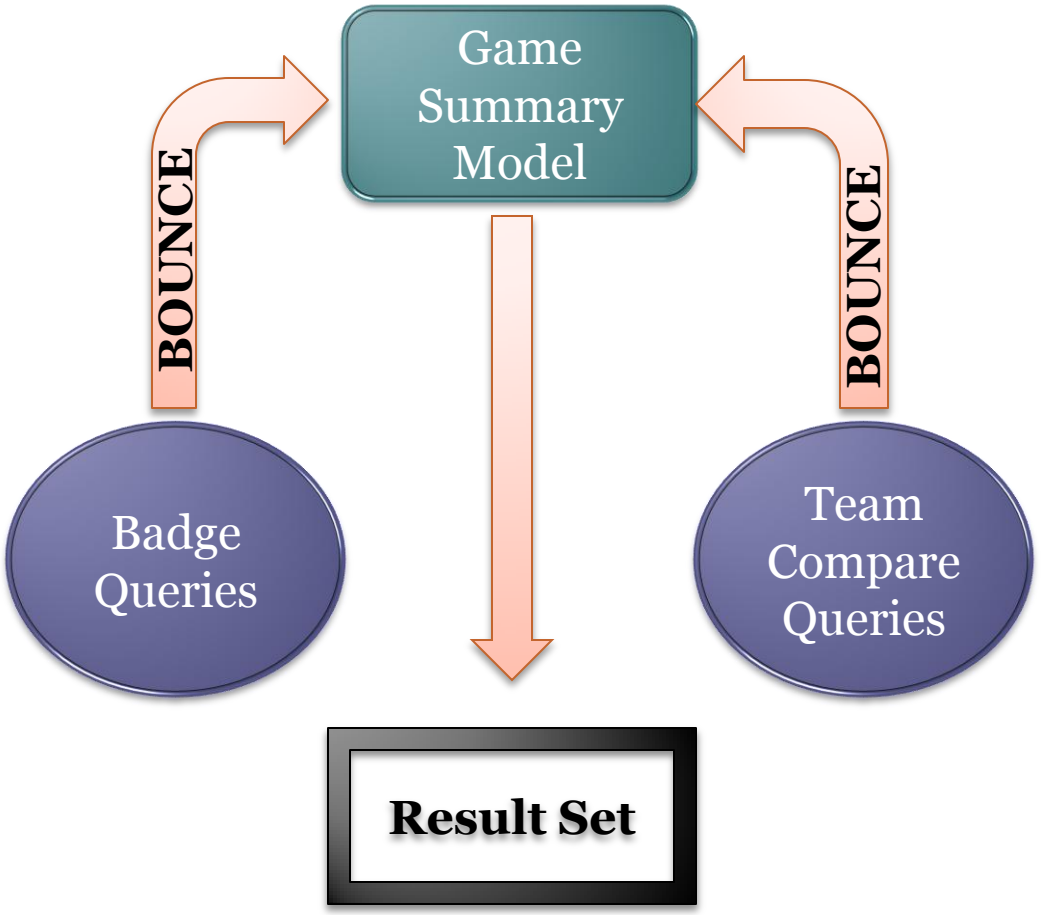


Workflow Details – step 4

We define a boatload of SELECT queries.

Those queries are **bounced** against our newly-created 'enhanced model'.

What comes back is gathered into a result set.



Badge Queries

Our badge queries, working off the Game Summary Model, are all similar in look:

We select on a **union** of two graph patterns, to make sure we check for a teams home games and away games.

```
;; Get hit 50 times or more in a game
(def check-magnets "
prefix : <http://www.nhl.com/>
select ?game ?team
{
 {?game :hometeam ?team .
  ?game :awayHits ?hits .
  filter (?hits >= 50 )}
 union
 {?game :awayteam ?team .
  ?game :homeHits ?hits .
  filter (?hits >= 50 )}
}
")
```

Team Comparison Queries

These are odd queries, because we want the **Top 5** matches for each team.

The way I do this is to parameterize my sparql query.

There are lots of triple-store-specific solutions for parameterization.

I chose for a vendor-neutral approach, since I was already in a programming environment.

```
;; Top 5 teams hit by the subject
(defn hits-outgoing [subj]
  (str "
prefix : <http://www.nhl.com/>
select ?team ?avg
{
 { select ?team (avg (?hits) as ?avg)
  {?game :hometeam " subj " . ?game :awayteam ?team .
   ?game :homeHits ?hits}
  group by ?team ?game
 }
 union
 { select ?team (avg (?hits) as ?avg)
  {?game :awayteam " subj " . ?game :hometeam ?team .
   ?game :awayHits ?hits}
  group by ?team ?game
 }
}
order by desc(?avg)
limit 5
"))
```

And OWL?

I didn't use any OWL

And it's true, I don't tend to like OWL

But OWL's not so bad (OWL 2 stinks, though)

Some of the inferences might have been done here with OWL-type reasoning. I just chose the SPARQL CONSTRUCT route.

Seabass is set up to do just RDF/S reasoning, using Jena's InfModel. Easy enough to use OntModel if you feel bad for this little guy.

Next Steps

- Finish out the web app
- Celebrity Semantics app!

- Ecological Semantics app!!

Questions?