



# RDFox Class



The first high-performance knowledge graph built from  
the ground up with semantic reasoning in mind

# Objectives



By the end of the class everyone will have:

- Set up RDFox
- Loaded data
- Run queries
- Written rules and understand reasoning in RDFox
- An overview of deployment options

## Setting up RDFox

- License and executable
- IDE
- REST endpoint and UI

## Loading data into RDFox

- Create datastore
- Import data

## Exploring and querying with SPARQL

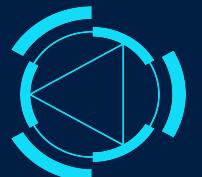
- Basics of SPARQL
- Useful queries to explore
- Aggregates
- Negation
- Filters
- Binds
- Optionals

## Reasoning

- Basic rules
- Aggregates
- Negation
- Filters
- Binds
- Incremental reasoning
- OWL

## Deployment

# Exercises



We will go through some exercises for each objective:

- Starting with a short explanation of each point.
- Everyone will then be able to do a short exercise. Advanced users will be able to also run through harder, bonus exercises.
- Then we regroup to check that everyone was able to complete the task

Please feel free to ask questions at any time.

Links are provided for reference.



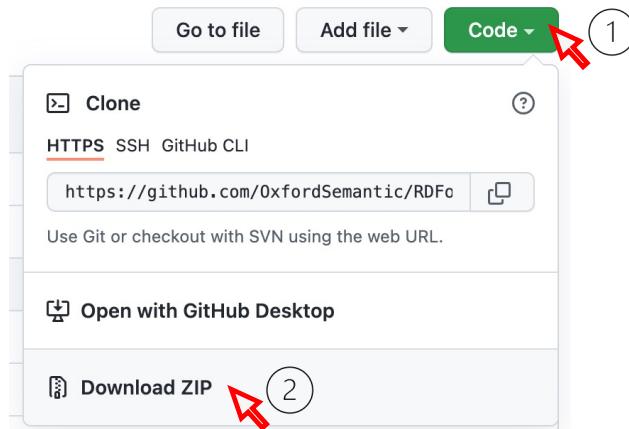
<https://docs.oxfordsemantic.tech/>



# Download the class materials

- Download or clone the class materials from Github:

<https://github.com/OxfordSemantic/RDFoxClass>

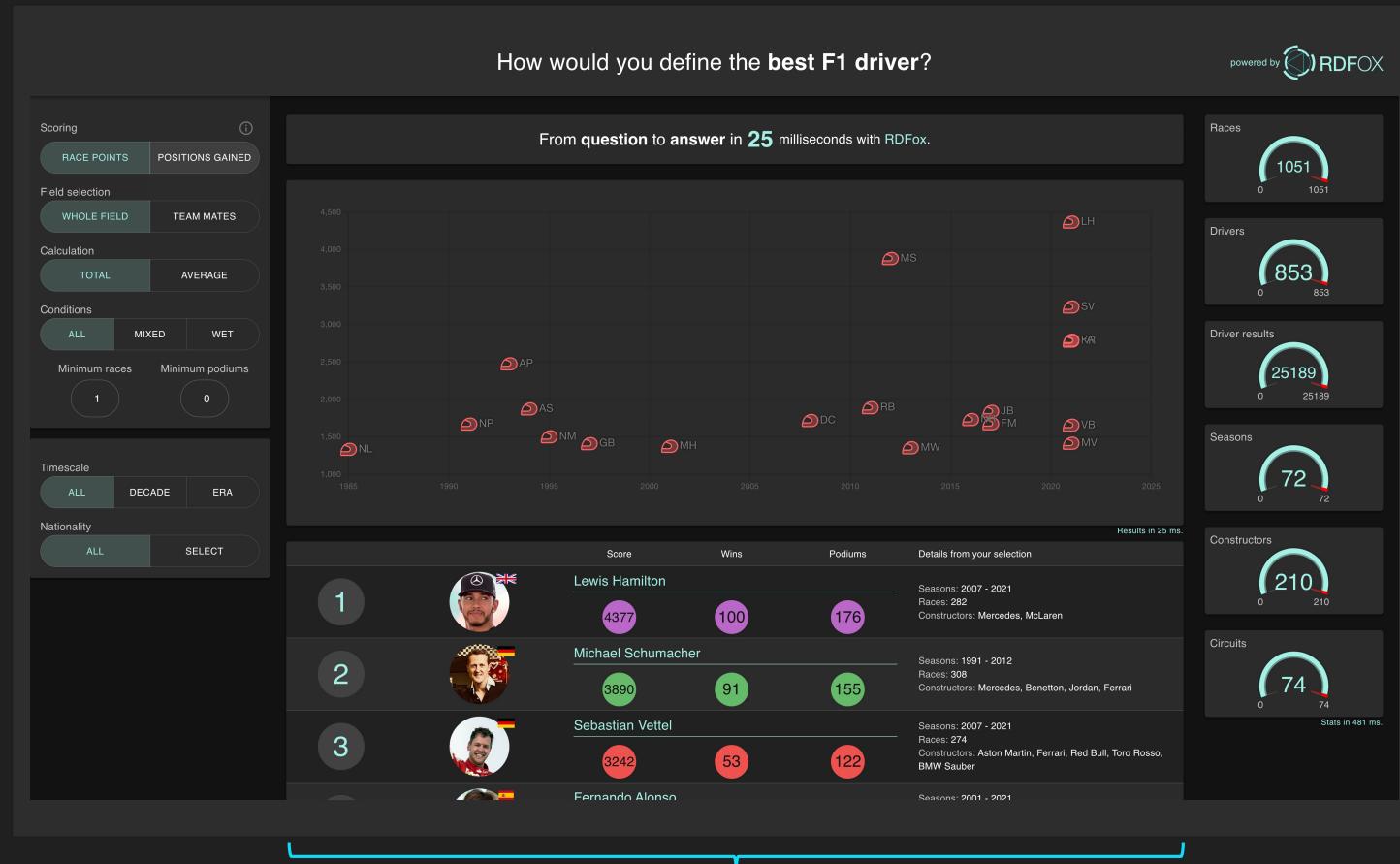


or

```
git clone https://github.com/OxfordSemantic/RDFoxClass.git
```

## RDFox Demo summary “Who is the Greatest Formula 1 Driver of all Time?”

Controls and filters for the scoring system.

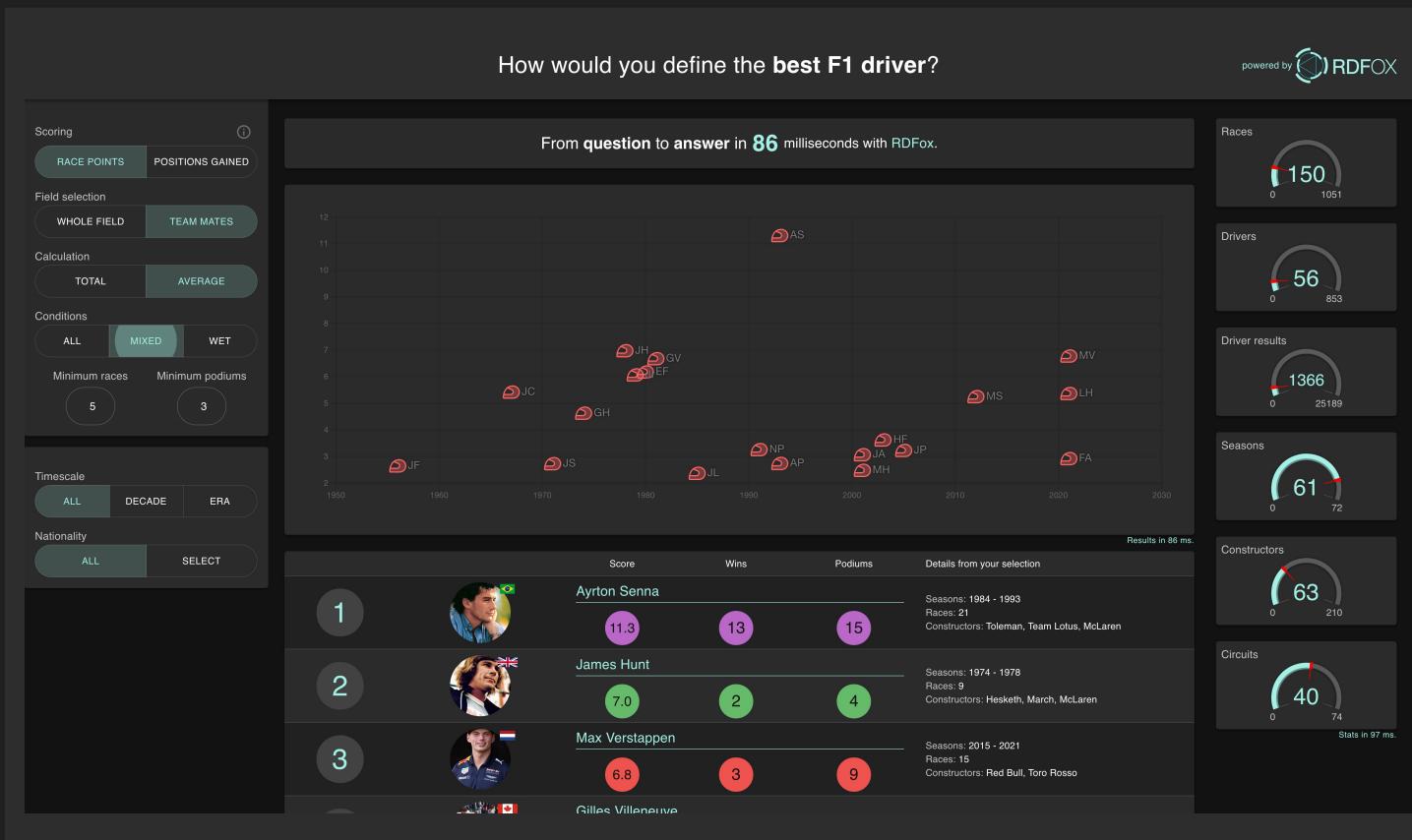


Statistics about the data used to form the results.

This matches the filters.

The graph displays the results, Driver Score vs Final Race Date. Below are the top 10 drivers and their details.

Changing the controls has a huge impact on the results.

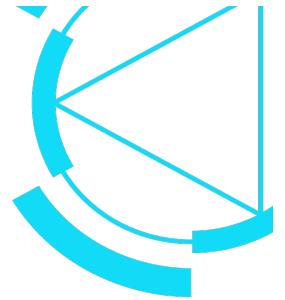


This computation takes milliseconds, despite the vast network of data involved.

Try it for yourself!

<http://f1.rdfbox.tech>

© Oxford Semantic Technologies 2022



# Setting Up RDFox

---

© Oxford Semantic Technologies 2022

## Setting up RDFox

- License and executable
- IDE
- REST endpoint and UI

## Loading data into RDFox

- Create datastore
- Import data

## Exploring and querying with SPAF

- Basics of SPARQL
- Useful queries to explore
- Aggregates
- Negation
- Filters
- Binds



# Setting up RDFox – License and executable

- You should have obtained a license from:  
<https://www.oxfordsemantic.tech/tryrdfoxforfree>
- You will also need to download the RDFox package from:  
<https://www.oxfordsemantic.tech/downloads>
- Unzip the package, then put the license into the unzipped folder.
- You might need to right-click on the RDFox executable to open it on Mac, for OS security reasons.
- We will be running the executable in the next step.

The screenshot shows the Oxford Semantic Technologies website with a dark header bar. The header includes the company logo, navigation links for Product, Resources, Company, and Contact, and a search icon. Below the header, the word "Downloads" is centered. Underneath, "RDFox v5.4" is displayed. Five download links are shown in blue boxes: "MAC (INTEL)" (macOS 10.14 or higher), "MAC (ARM)" (macOS 11 or higher), "LINUX (x86)" (Centos 7, Ubuntu 16.04 or higher), "LINUX (ARM)" (Centos 7, Ubuntu 16.04 or higher), and "WINDOWS" (Windows 8 or higher).



# Setting up RDFox – IDE

- Open your favourite IDE (I will use VS Code), and open the RDFoxClass folder
- Open a terminal in the IDE
- In the terminal, run:  
`<Path/to>/RDFox sandbox`
- The RDFox server is now running, but it's empty...

Source code for RDFox v1.0 Copyright 2013 Oxford University Innovation Limited and subsequent improvements Copyright 2017-2021 by Oxford Semantic Technologies Limited.

This copy of RDFox is licensed for Developer use to Tom Vout (tom.vout@oxfordsemantic.tech) of OST until 07-Jun-2022 16:01:44

This system is equipped with 16.9 GB of RAM, and RDFox is configured to use at most 15.2 GB (89.9% of the total).

Currently, 2.8 GB (18.4% of the amount allocated to RDFox) appear to be available on the system.

Since RDFox is a RAM-based system, its performance can suffer when other running processes use a lot of memory.

A new server connection was opened as role 'guest' and stored with name 'sc1'.

> █



<https://docs.oxfordsemantic.tech/command-line-reference.html#starting-the-rdfbox-process>



# The REST Endpoint and UI

- In the terminal, run:

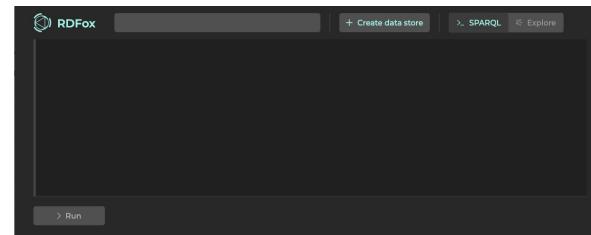
```
endpoint start
```

```
> endpoint start

WARNING: The RDFox endpoint is running with no transport layer security (TLS). This could allow attackers to
steal information including role passwords.
      See the endpoint.channel variable and related variables in the description of the RDFox endpoint for
details of how to set up TLS.

The REST endpoint was successfully started at port number/service name 12110 with 7 threads.
> |
```

- The RDFox endpoint is now running, so we can use the web UI
- Open a browser and go to:  
[localhost:12110/console](https://localhost:12110/console)
- This will show an empty console



 <https://docs.oxfordsemantic.tech/command-line-reference.html#endpoint>

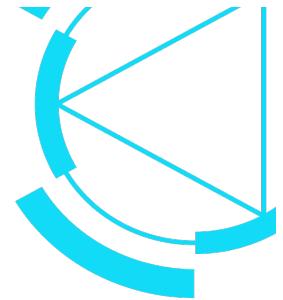
# Loading Data into RDFox

---

© Oxford Semantic Technologies 2022

## Setting up RDFox

- ✓ License and executable
- ✓ IDE
- ✓ REST endpoint and UI



## Loading data into RDFox

- Create datastore
- Import data

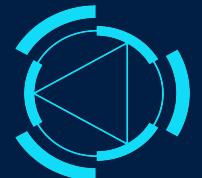
## Exploring and querying with SPARQL

- Basics of SPARQL
- Useful queries to explore
- Aggregates
- Negation
- Filters
- Binds
- Optionals

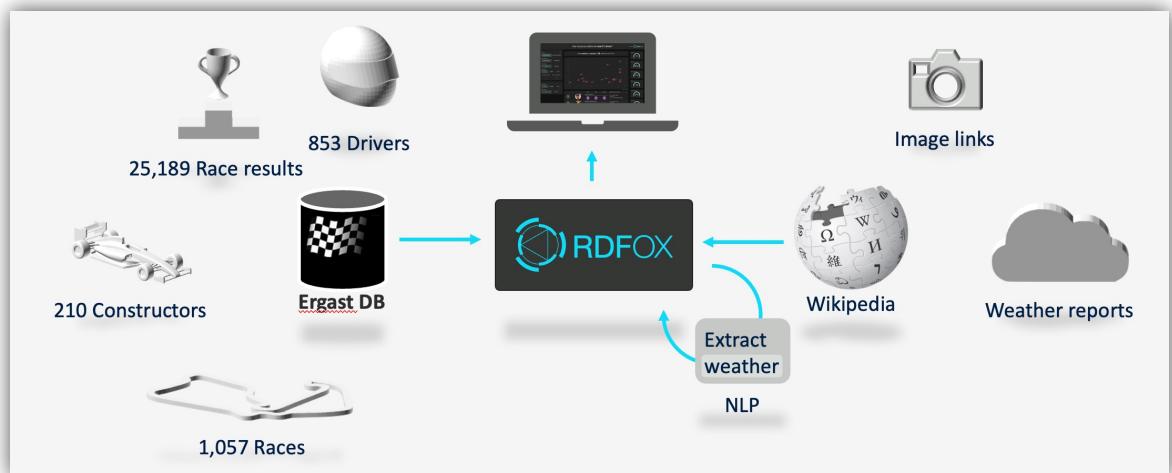
## Reasoning

- Basic rules
- ~~Aggregates~~

# The data for today's class



- Race data from the fan-run Ergast database
  - Drivers
  - Races
  - Race results
  - Constructors
- Combined with additional information from Wikidata
  - Driver images
  - Race weather reports



# Loading data into RDFox – Create datastore

A screenshot of the RDFox web interface. At the top, there's a navigation bar with the RDFox logo, a search bar, and buttons for '+ Create data store', 'SPARQL', and 'Explore'. Below the navigation is a dark sidebar with a 'Run' button. A modal window titled 'Create data store' is open in the center. It contains a brief description: 'A data store is a queryable container in RDFox that stores facts, rules, and OWL axioms.' Below this is a 'Name' input field containing 'f1'. Underneath is an 'Advanced settings' section with a 'Cancel' button and a red-bordered 'Create data store' button. To the right of the modal is a smaller, semi-transparent overlay window with a green checkmark and the text 'Data store has been created'. At the bottom of this overlay is a red-bordered 'Import content' button.

+ Create data store

SPARQL Explore

f1

Advanced settings

Create data store

Data store has been created

Import content

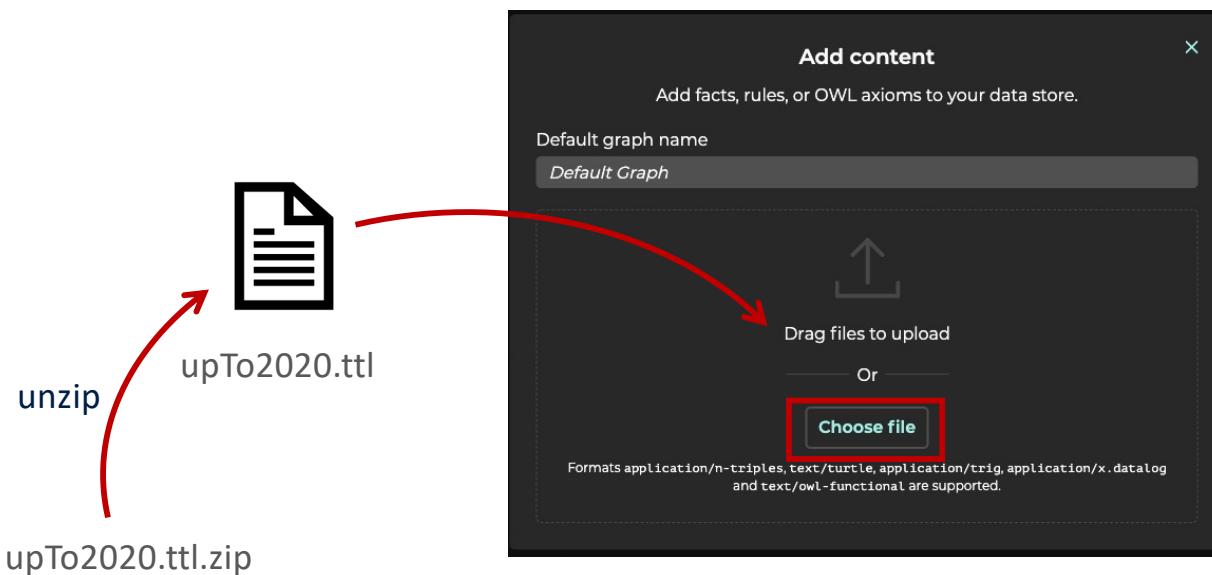
- i Make sure to call your data store f1 so that the links in this guide work.



<https://docs.oxfordsemantic.tech/getting-started.html#getting-started-with-the-web-console>



# Loading data into RDFox – Load files



# Exploring and Querying with SPARQL

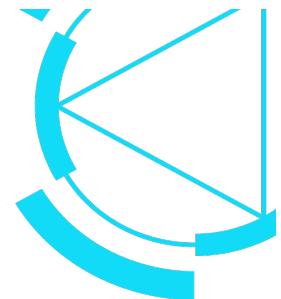
✓ IDE

✓ REST endpoint and UI

Loading data into RDFox

✓ Create datastore

✓ Import data



## Exploring and querying with SPARQL

❑ Basics of SPARQL

❑ Useful queries to explore

❑ Aggregates

❑ Negation

❑ Filters

❑ Binds

❑ Optionals

## Reasoning

❑ Basic rules

❑ Aggregates

❑ Negation

❑ Filters

❑ Binds

# From question to answer in milliseconds



Scoring

RACE POINTS POSITIONS GAINED

Field selection

WHOLE FIELD TEAM MATES

Calculation

TOTAL AVERAGE

Conditions

ALL MIXED WET

Minimum races Minimum podiums

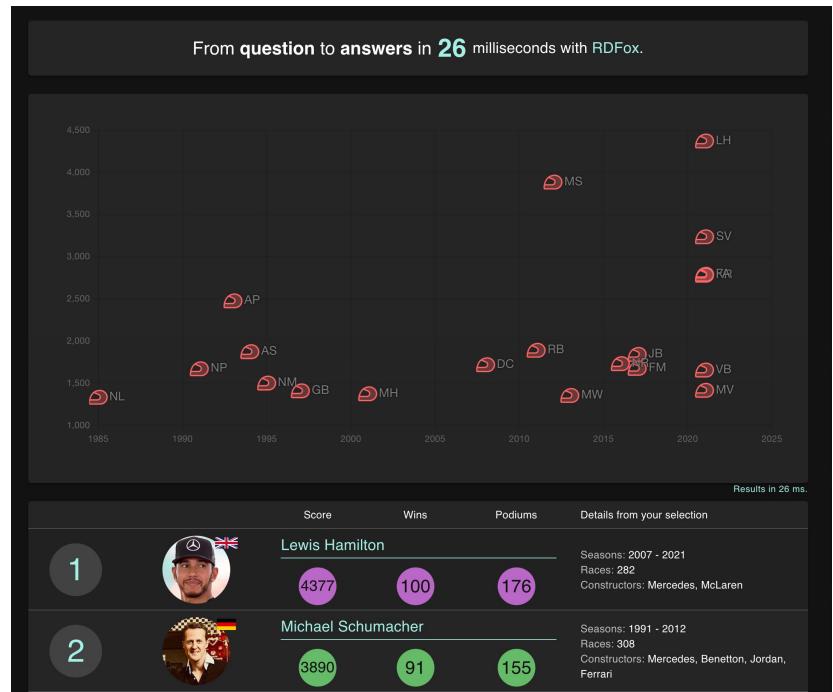
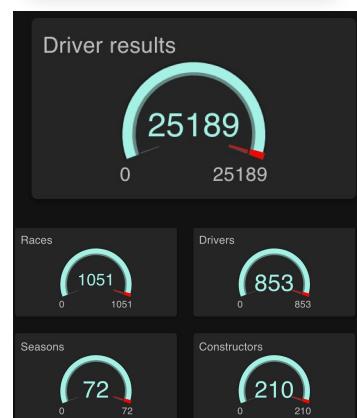
1 0

Timescale

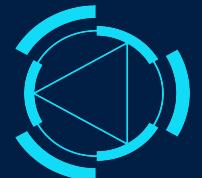
ALL DECADE ERA

Nationality

ALL SELECT



# SPARQL “used to express queries”



**SPARQL 1.1**



<https://www.w3.org/TR/sparql11-overview/>

W3C Recommendation

**W3C**

## SPARQL 1.1 Overview

W3C Recommendation 21 March 2013

### Abstract

RDF is a directed, labeled graph data format for representing information in the Web. This specification defines the syntax and semantics of the SPARQL query language for RDF. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs.

Copyright © 2013 W3C® (MIT, ERCIM, Keio, Beihang). All Rights Reserved. W3C liability, trademark and document use rules apply.

# Query 1 – Basic SELECT

[Console Query](#)

[Git Repository](#)



**SELECT** queries are used to extract results from the dataset and can be modified to match whatever pattern you desire.

This is the simplest **SELECT** query. It returns all the triples in your dataset.

The **SELECT** keyword determines the type of query. **SELECT** simply returns values.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # This query matches all the triples,
4 * and returns all the variables from the matched results
5 SELECT ?S ?P ?O
6 WHERE {
7   ?S ?P ?O
8 }
```

The **WHERE** clause describes what the query is looking for.

In this case we want all triples, so we write the variables **?S ?P** and **?O**, showing we want cases where a subject, predicate, and object exist, with any value in any position.

The output variables **?S ?P** and **?O** tell the **SELECT** query what to return from the variables that are found by the **WHERE** clause.

Here we're asking for everything.

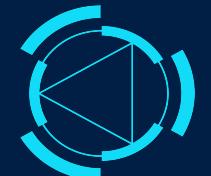
You can call your variables whatever you like but upper case **?S ?P** and **?O** have some special properties that we will make use of later.



## Query 2 – DISTINCT SELECT

[Console Query](#)

[Git Repository](#)



Often you'll want to be more specific.

This time, we don't want all the triples to be returned, just their predicates.

Using **SELECT** alone would return duplicates, but we don't want that either.

The **DISTINCT** modifier tells the query to disregard duplicate results.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # What kinds of edges are in the data
4 SELECT DISTINCT ?P
5 WHERE {
6   ?S ?P ?O
7 }
```

This time we only ask for **?P** to be returned.

We still have to tell the query that the variable **?P** should come from the predicate of a triple, and that we want to consider all triples.

Therefore, we still need **?S ?P ?O** inside the **WHERE** clause.

# Query 3 – Finding Classes

[Console Query](#)

[Git Repository](#)



We want to **SELECT** all the **DISTINCT** classes that exist in our data to find the all the types of things we're storing.

In SPARQL, the keyword '**a**' is used universal as the predicate for specifying the type of an entity.

We want to find all instances of triples **WHERE** nodes have the '**a**' property.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # What classes of nodes are there in the data?
4 SELECT DISTINCT ?type
5 WHERE {
6   ?S a ?type
7 }
```

We are only interested in the object of these triples – their class.

We have defined the variable **?type** to help us.

# Query 4 – Class Properties

[Console Query](#)  
[Git Repository](#)



This is one of the most important queries we'll cover today.

We **SELECT** a class of interest, and then return all its properties—not their values, just the properties it possesses.

We want to find all instances of the class **:drivers**.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # What types of edges are there specifically for drivers?
4 SELECT DISTINCT ?p
5 WHERE {
6   ?s a :drivers ;
7   ?p ?o .
8 }
```

And to consider all triples that share their subjects, **?s**.

This whole line declares a prefix. This creates a substitute, so every time you see ':', it is equivalent to <http://www.oxfordsemantic.tech/f1demo/>.

Subjects can be shared across multiple lines.

To denote this, all lines except the last must end with a ';'.

The last ends with a '.'.

# Exercise 1



- Run through **queries 1-4** in the **web console**.
- Write them out yourself. See how and what they do.
- Please **open up the class pdf within the RDFoxClass download** to follow along.
- You can click on **Console Query** at any time to see an editable version of the query for you to play around with. Sometimes there will be parts missing for you to fill in.
- You can click on **Git Repository** for the query in full for guidance.

# Query 5 – Finding Drivers

Incomplete Query

Answer



Find all drivers along with their forename and surname.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # What are the names of my drivers?
4 SELECT DISTINCT ?driver ?forename ?surname
5 WHERE {
6   ?driver a ___ ;
7   :drivers_forename ___ ;
8   :drivers_surname ___ .
9 }
```

We have introduced new variables to the output that must appear in the **WHERE** clause.

# Query 6 – A Driver’s Properties

[Incomplete Query](#)

[Answer](#)



To build on what we’ve got so far, let’s find all the associated properties and corresponding values of a specific driver. Say, Lewis Hamilton.

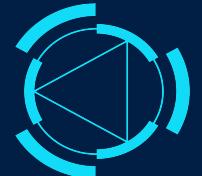
```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # Can you return everything about a specific driver, say Lewis Hamilton?
4 SELECT ?driver ?p ?o
5 WHERE {
6   ?driver a :drivers ;
7   --- "Lewis" ;
8   --- "Hamilton" ; ←
9   --- --- .
10 }
```

Literals, or the values attached to properties, can be specified in queries too.

Here we’re looking for a specific string so we use the “”’ characters to identify it.

# Query 7 – Hamilton’s Races

Incomplete Query  
Answer



```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # Number of races Lewis Hamilton raced
4 SELECT (COUNT(?race) AS ?raceCount) ←
5 WHERE {
6
7   ?driver :drivers_forename "Lewis" ;
8       :drivers_surname "Hamilton" .
9
10  ?result :results_driver ___ ;
11      :results_race ___ .
12
13 }
```

The **COUNT** function counts the number of instances of a variable, **?race**, and binds the resulting value as another variable, **?raceCount**.

## Exercise 2



Complete **Query 5** to find the forenames and surnames of all the drivers.

We've given you the query structure in the [Console](#) link. **Fill in the blanks** to make the query work and return the desired result.

If you need a clue, the [Answer](#) is available in the Git Repository (linked on each page).

### HINT

It might help you to have **several tabs open** with the previous queries in each one—they will help you fill in the gaps!

# Exercise 3



Complete **Query 6** to find out all of the properties associated with Lewis Hamilton, along with their values.

## HINT

Look back at **queries 4 & 5** to help with this exercise.

## Exercise 4



Complete **Query 7** to find the total number of races Hamilton has taken part in.

# Exercise 5



Complete **Query 8** to find the total number of races each driver individually has taken part in.

## BONUS Qs

Display the **forename** and **surname** of the driver (**instead of the IRI**) next to their race count.

[Answer](#)

Using the **GROUP\_CONCAT** aggregate function, find the list of teammates that each driver has ever had.

[Answer](#)

## HINTS:

Two people are teammates if they competed in the same **race** for the same **constructor**. Notice every result is associated with a constructor and race.

You can put **DISTINCT** inside of an aggregate function to deduplicate.

# Query 8 – All Drivers’ Races

Incomplete Query

Answer



```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # Number of races per driver, ordered
4 SELECT ?driver (COUNT(?race) AS ?raceCount)
5 WHERE {
6   ?result :results_driver ___ ;
7   :results_race ?race .
8 }
9 GROUP BY ___
10 ORDER BY DESC(____)
```

The **GROUP BY** clause states how the results should be grouped when returned. Aggregate functions, such as **COUNT**, will now be performed for each group.

The **ORDER BY** clause determines the order of the query results.

The **DESC** modifier tells the **ORDER BY** clause the variable indicated should be ranked in descending order.

# Query 9 – Drivers Without Podiums

[Incomplete Query](#)  
[Answer](#)



The **FILTER** function contains properties that are used to restrict the results that are returned to just those that match.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # Drivers who never got a podium
4 SELECT
5 WHERE {
6 # First, get all the drivers with their names.
7 ?driver a :drivers ;
8 :drivers_forename ?forename ;
9 :drivers_surname ?surname .
10
11 # Then make sure that they have never achieved a podium (i.e. positions 3, 2, or 1)
12 FILTER NOT EXISTS {
13 ?result ___ ?driver ;
14 :results_positionOrder ?positionOrder .
15 FILTER(?positionOrder IN(1, 2, 3))
16 }
17 }
```

The **IN** operator checks to see if the variable on the left-hand side appears in the list.

The **NOT EXIST** operator is our first glimpse of negation. It returns true if its conditions are **not** matched.

In effect, in this case the conditions of the **FILTER** are reversed, so results are limited to those that **do not** match what is stated.

# Query 10 – Most Races Without a Podium

Incomplete Query

Answer



Here we have used a query within a query—an ‘inner query’, or ‘subquery’.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # Most races run without ever getting a podium
4 SELECT ?forename ?surname ?raceCount
5 WHERE {
6 # First, get all the drivers with their names.
7 # The order of the query atoms is mostly not important, as RDFox will rearrange it
8 # internally in order to have the best performance automatically.
9     ?driver a :drivers ;
10    :drivers_forename ?forename ;
11    :drivers_surname ?surname .
12
13 # Then make sure that they have never achieved a podium (i.e. positions 3, 2, or 1)
14   FILTER NOT EXISTS {
15     ?result2 :results_driver ?driver ;
16     :results_positionOrder ?positionOrder .
17     FILTER(?positionOrder IN(1, 2, 3))
18   }
19
20 # Finally find out how many races they have raced in.
21 # This is done in an 'inner query'
22 {
23   SELECT ___ (COUNT(?___) AS ?raceCount)
24   WHERE {
25     ?result :results_driver ?driver ;
26     :results_race ?race .
27   }
28   GROUP BY ___
29 }
30 }
31 ORDER BY DESC(___) # Don't forget to order
```

# Query 11 – Driver Win Percentage

Incomplete Query

Answer



The **BIND** function sets a variable to a specific value.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # Drivers with their win percentage, ordered by win percentage
4 SELECT ?forename ?surname ?raceCount ?raceWins ?percentage
5 WHERE {
6 # First get the drivers
7 ?driver :drivers_forename ?forename ;
8 :drivers_surname ?surname .
9
10 # Then get the race count for each driver with an innery query...
11 {
12   SELECT ?driver (COUNT(?race) AS ?raceCount)
13   WHERE {
14     ?result :results_driver ?driver ;
15       :results_race ?race .
16   }
17   GROUP BY ?driver
18 }
19
20 # ... and get the *win* count for each driver with another inner query.
21 {
22   SELECT ___ (COUNT(?race) AS ___)
23   WHERE {
24     ?result :results_driver ___ ;
25       :results_race ___ ;
26       :results_positionOrder 1 .
27   }
28   GROUP BY ?driver
29 }
30
31 # Finally use the two aggregate variables to compute a percentage
32 # with the BIND keyword.
33 BIND(?raceWins/___ AS ?percentage)
34
35 }
36 ORDER BY DESC(?percentage)
```

# Query 12 – Correction: Win Percentage

Incomplete Query

Answer



The **OPTIONAL** keyword states that if, for a given case, the subsequent result cannot be evaluated, skip over it and leave it undefined.

```
1 PREFIX : <http://www.oxfordsemantic.tech/f1demo/>
2
3 # Can you spot a problem?
4
5 # We only return 'biased' results, i.e. those who have at least one win!
6 # We will solve this using the so called OPTIONAL keyword
7
8 SELECT ?forename ?surname ?raceCount ?raceWinsFinal ?percentage
9 WHERE {
10 # First get the drivers
11 ?driver :drivers_forename ?forename ;
12     :drivers_surname ?surname .
13
14 # First get the race count for each driver.
15 {SELECT ?driver (COUNT(?race) AS ?raceCount)
16 WHERE {
17     ?result :results_driver ?driver ;
18     :results_race ?race .
19 }
20 GROUP BY ?driver}
21
22 ▲ Then *optionally* get the win count for each driver.
23 OPTIONAL {SELECT ?driver (COUNT(?race) AS ?raceWins)
24 WHERE {
25     ?result :results_driver ?driver ;
26     :results_race ?race ;
27     :results_positionOrder 1 .
28 }
29 GROUP BY ?driver}
30
31 # Using the COALESCE keyword, we make sure that when the number of wins is undefined
32 # it is 'bound' as 0.
33 BIND(COALESCE(___,0) AS ?raceWinsFinal) ←
34 # And use the two aggregate variables to compute a percentage
35 BIND(___/?raceCount AS ?percentage)
36
37 }                                     © Oxford Semantic Technologies 2022
38 ORDER BY DESC(___)
```

The **COALESCE** function returns the first value in its list that does not throw up an error (including undefined).

It is bound to the stated variable.

# Exercise 6



Complete **Query 9** to find all of the drivers who have **never** finished on the podium.

## HINT:

To finish on the podium you must come in positions 1, 2, or 3.

Every result will be associated with a position number which tells us where they finished in the race.

## Exercise 7



Complete **Query 10** to rank all of the drivers who have **never** finished on the podium, from most to least races without a podium.

## Exercise 8



Complete **Query 11** to calculate the **win percentage** of each driver individually, ranking them from most to least successful.

# Exercise 9



Complete **Query 12** using the **OPTIONAL** keyword to correct the calculation of win percentages to include even those drivers who never won.

## BONUS

Use **FILTER** to restrict this list to only drivers who **won at least 5** races.

[Answer](#)



# Running SPARQL from the command line

- As well as running SPARQL queries from the web console queries can be run from the command line.
- Here are some of the commands that you will need to set this up:

`active <data store name>`

- The active command set the active data store for subsequent commands.

`set output out`

- The `set` command lets you set a range of variables used by RDFox.
- `set output out` sets the output variable so that query results are sent to the command prompt. An alternative would be to send the output to a file or null.

`answer <query file name>`

- Runs the query by that filename.



[https://docs.oxfordsemantic.tech/  
command-line-reference.html](https://docs.oxfordsemantic.tech/command-line-reference.html)



## Exercise 10

From the RDFox command line:

- a. Switch to make the `f1` data store active

```
active f1
```

- a. Set the output to the screen

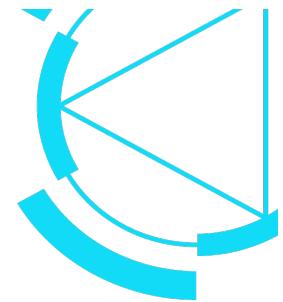
```
set output out
```

- b. Run **query 6** again for “Lewis Hamilton”. `queries/q6.rq`

If you are running RDFox from `RDFoxClass` directory then the command should be

```
answer Queries/q6.rq
```

5 min Break



# Reasoning with RDFox

---

## Basics of SPARQL

- ✓ Useful queries to explore
- ✓ Aggregates
- ✓ Negation
- ✓ Filters
- ✓ Binds
- ✓ Optionals



## Reasoning

- ❑ Basic rules
- ❑ Aggregates
- ❑ Negation
- ❑ Filters
- ❑ Binds
- ❑ Incremental reasoning
- ❑ OWL

## Deployment

# Reasoning



[*fact A...*] :- [*fact B...*]

*"A... is true if B... is true"*



<https://docs.oxfordsemantic.tech/reasoning.html>

## Rule 1 – Basic Rule

In the RDFoxClass folder:

Incomplete Query | [Rules/r1.dlog](#)

Answer | [RuleAnswers/r1.dlog](#)



This rule will add a direct link between drivers and the races they raced in

```
[?driver, :hasRacedIn, ?race] :-  
    [?result, :results_race, ?race],  
    [?result, :results_driver, ?driver] .
```

## Rule 1 – Basic Rule

In the RDFoxClass folder:

Incomplete Query | [Rules/r1.dlog](#)

Answer | [RuleAnswers/r1.dlog](#)



The same could be achieved with the following SPARQL update (i.e. a write query). As we'll explain, rules offer several advantages over SPARQL updates.

```
INSERT {?driver :hasRacedIn ?race}
WHERE {
  ?result :results_race ?race ;
  | :results_driver ?driver .
}
```



## Exercise 11 – Importing rule 1

- Run query 6 again from the CLI:  
`answer Queries/q6.rq`
- Import this rule (Rule 1) from the CLI:  
`import Rules/r1.dlog`
- Rerun query 6, again from the CLI:  
`answer Queries/q6.rq`

You should now see a lot more results: the rule has *materialised* them.



## Exercise 12 – Rule management

- Delete the rule (it's a useful thing to do when you make a mistake in the rule) with:

```
import - Rules/r1.dlog
```

- Rerun the query

```
answer Queries/q6.rq
```

The materialised results have disappeared...

- Finally, **reimport** the rule, as **you will need it later on.**

## Rule 2 – FILTERs in rules

In the RDFoxClass folder:

Incomplete Query | [Rules/r2.dlog](#)

Answer | [RuleAnswers/r2.dlog](#)



We can use FILTERs in rules too, in this case to find the races where drivers reached a podium.

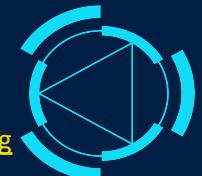
```
[?driver, :hasPodiumInRace, ?race] :-  
    [?result, :results_race, ?race],  
    [?result, :results_driver, ?driver],  
    [?result, :results_positionOrder, ?positionOrder],  
    FILTER(?positionOrder < 4) .
```

# Rule 3 – Aggregate Rule

In the RDFoxClass folder:

Incomplete Query | [Rules/r3.dlog](#)

Answer | [RuleAnswers/r3.dlog](#)



This rule will add a count of races every driver has entered.

Notice that we use the previously inferred :hasRacedIn property.

Rules can in fact be composed, and RDFox will automatically find the order in which to run them (so you don't have to).

```
[?driver, :hasRaceCount, ?raceCount] :-  
    AGGREGATE(  
        [?driver, :hasRacedIn, ?race]  
        ON ?driver  
        BIND COUNT(?race) AS ?raceCount  
    ) .
```



<https://docs.oxfordsemantic.tech/reasoning.html#aggregate>

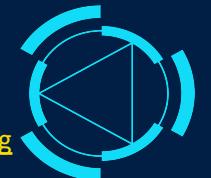
© Oxford Semantic Technologies 2022

## Rule 4 – Another aggregate Rule

In the RDFoxClass folder:

Incomplete Query | [Rules/r4.dlog](#)

Answer | [RuleAnswers/r4.dlog](#)



This rule will add a count of races every driver has *won* (if this is 1 or more). We can put more than one atom *inside* the AGGREGATE statement.

```
[?____, :hasRaceWinCount, ?____] :-  
    AGGREGATE(  
        [?____, :results_driver, ?____],  
        [?____, :results_positionOrder, 1]  
    ON ?driver  
    BIND COUNT(?____) AS ?____  
    ) .
```

## Exercise 13 – Filter and Aggregate rules



- Import `r2.dlog`, i.e. the FILTER rule.
- Import the aggregate rule (`r3`, to compute race counts).
- Then, fill in the blanks in the second aggregate rule (`r4.dlog`).
- Finally, import `r4.dlog`.
- Run query 13 (`q13.rq`) to see the results of these rules.
  - Note down the results for Lewis Hamilton as we will be referring to this later.

## Rule 5 – Negation as Failure

In the RDFoxClass folder:

Incomplete Query | [Rules/r5.dlog](#)

Answer | [RuleAnswers/r5.dlog](#)



This rule will tell us that if a driver does not have a race where they were on the podium, then he is a :DriverWithoutPodiums.

Here we are thus looking for something that is **not** in the data (i.e. no race where the driver was on the podium).

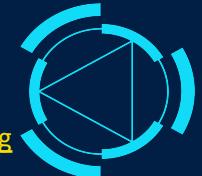
```
[?driver, a, :DriverWithoutPodiums] :-  
    [?driver, a, :drivers],  
    NOT EXISTS ?race IN (  
        |   [?driver, :hasPodiumInRace, ?race]) .
```



[https://docs.oxfordsemantic.tech/  
reasoning.html#negation](https://docs.oxfordsemantic.tech/reasoning.html#negation)

# Rule 5 –About Negation as Failure

In the RDFoxClass folder:  
Incomplete Query | [Rules/r5.dlog](#)  
Answer | [RuleAnswers/r5.dlog](#)



**Negation as failure** is particularly useful for many of our customers.

Having Negation as Failure can lead to ‘non-monotonic reasoning’—reasoning may need to retract some triples instead of just adding them.

This adds one more dimension to an already complicated problem.

Example:

What if a driver without any podiums *so far* gets one next year?

In 2021, for instance, George Russell got his first podium.

In our data, which is accurate up to and including 2020, George Russell would figure as a :DriverWithoutPodiums .

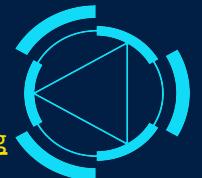
But add data from 2021, and we will need to retract this fact.

## Rule 6 – BIND rules

In the RDFoxClass folder:

Incomplete Query | [Rules/r6.dlog](#)

Answer | [RuleAnswers/r6.dlog](#)

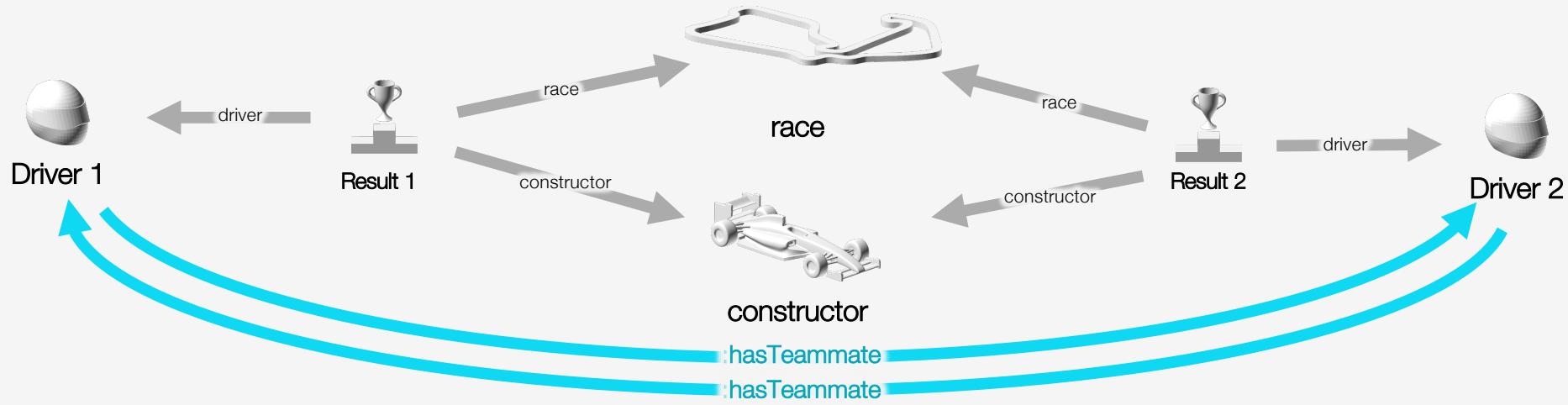
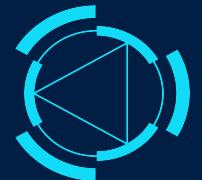


Using BIND in rules helps us to perform numerical reasoning.

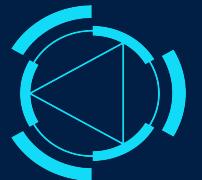
We can also do string manipulation.

```
[?driver, :hasWinPercentage, ?percentage] :-  
  | [?__, :hasRaceCount, ?__],  
  | [?__, :hasRaceWinCount, ?__],  
  | BIND(?raceWinCount/?raceCount AS ?percentage) .
```

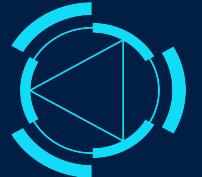
# Establishing teammate links with rules



## Exercise 14 – Negation as Failure rules



- Import r5.dlog, negation as failure
- Run query 14 (q14.rq) to see the results of these rules.



## Exercise 15 – BIND rules

- Import r6.dlog and use a query to check the triples were inferred.

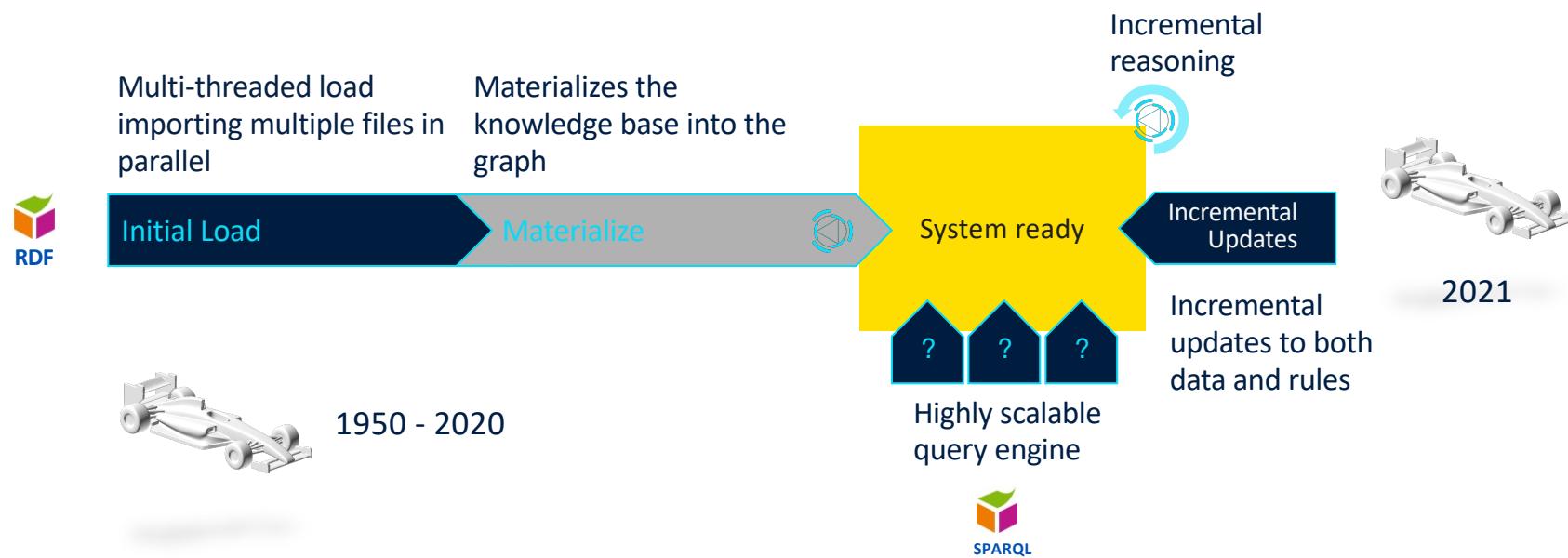
# Bonus Exercise

In the RDFoxClass folder:  
Answer | [RuleAnswers/teammate.dlog](#)



- Write and import a rule to find out if two drivers have been teammates, using the predicate :hasTeammate

# Incremental reasoning





## Incremental reasoning

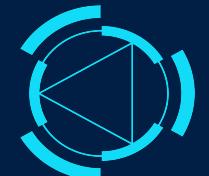
- This is one of the keys strengths of RDFox
- Data can be imported at any time, and the rules will automatically compute the necessary inferences
- This all happens *incrementally*, i.e. without needed to reboot
- Incremental reasoning open the door to many novel use cases previously impractical



## Exercise 17: Incremental reasoning

- Import the 2021-22 race data.
- From the command line run:  
`import 2021-22.ttl.zip`
- Note that from the command line we can import zip files.
- Now rerun some of your previous queries to see the updated results.

# OWL



- RDFox of course supports the OWL language, a W3C [standard](#).

Only the RL profile of OWL is supported

- This can be imported as triples, which can be parsed with the `importaxioms` command
- The parsing will translate triples into Datalog rules internally

```
@prefix : <http://www.oxfordsemantic.tech/f1demo/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

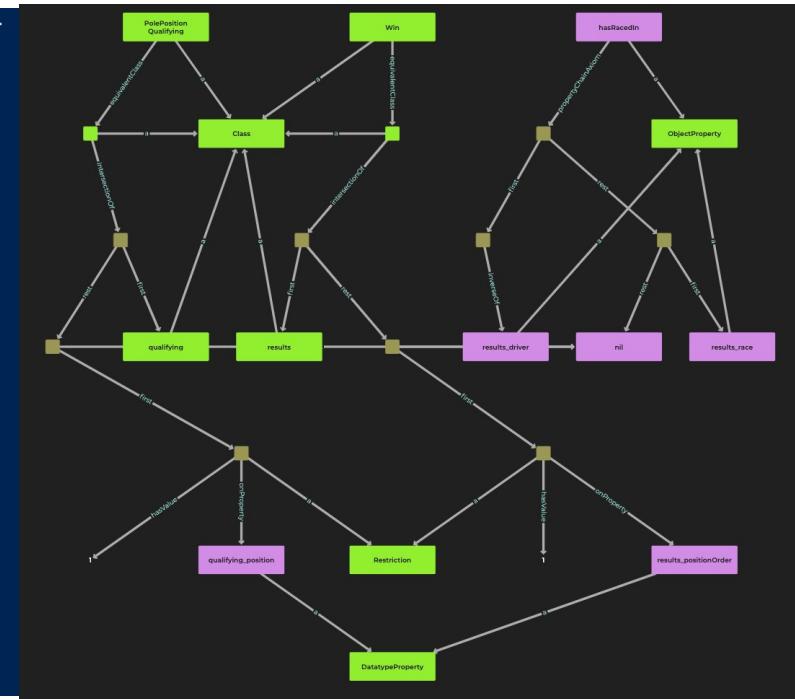
:PolePositionQualifying a owl:Class .
:qualifying a owl:Class .
:results a owl:Class .
:Win a owl:Class .

:results_driver a owl:ObjectProperty .
:results_race a owl:ObjectProperty .
:hasRacedIn a owl:ObjectProperty .
:results_positionOrder a owl:DatatypeProperty .
:qualifying_position a owl:DatatypeProperty .

:hasRacedIn owl:propertyChainAxiom (
  [owl:inverseOf :results_driver]
  :results_race
) .

:PolePositionQualifying owl:equivalentClass [
  a owl:Class ;
  owl:intersectionOf (
    :qualifying
    [
      a owl:Restriction ;
      owl:onProperty :qualifying_position ;
      owl:hasValue "1"^^xsd:integer
    ]
  )
] .

:Win owl:equivalentClass [
  a owl:Class ;
  owl:intersectionOf (
    :results
    [
      a owl:Restriction ;
      owl:onProperty :results_positionOrder ;
      owl:hasValue "1"^^xsd:integer
    ]
  )
] .
```





## Exercise 18: OWL

- Import the ontology into RDFox with:  
`import ontology.ttl`
- Parse the axioms with:  
`importaxioms`
- Check whether RDFox has inferred some new triples with:  
`answer Queries/q16.rq`



# RDFox reasoning vs. other solutions

Feature	Example	RDFox	Materialisation Competitors	Query Rewriting Competitors
	"List all financial instruments / derivative instruments / futures." [class inferencing]	✓	✓*	✓**
	"What is the nearest electrical switch for a circuit?" [negation + recursion]	✓		
	"What is the total volume and value of trades?" [aggregation + arithmetic]	✓		
	"No trades over a limit & no traders without trades!" [filters + negation]	✓	Partial (SHACL)	
	"Every component must be connected to a power source!" [negation + recursion]	✓		
	"What is the total value of assets owned through holdings?"	✓		

\* many times slower than RDFox

\*\* orders of magnitude slower than RDFox

# Deployment

---

## ▪ BASICS OF SPARQL

- ✓ Useful queries to explore
- ✓ Aggregates
- ✓ Negation
- ✓ Filters
- ✓ Binds
- ✓ Optionals

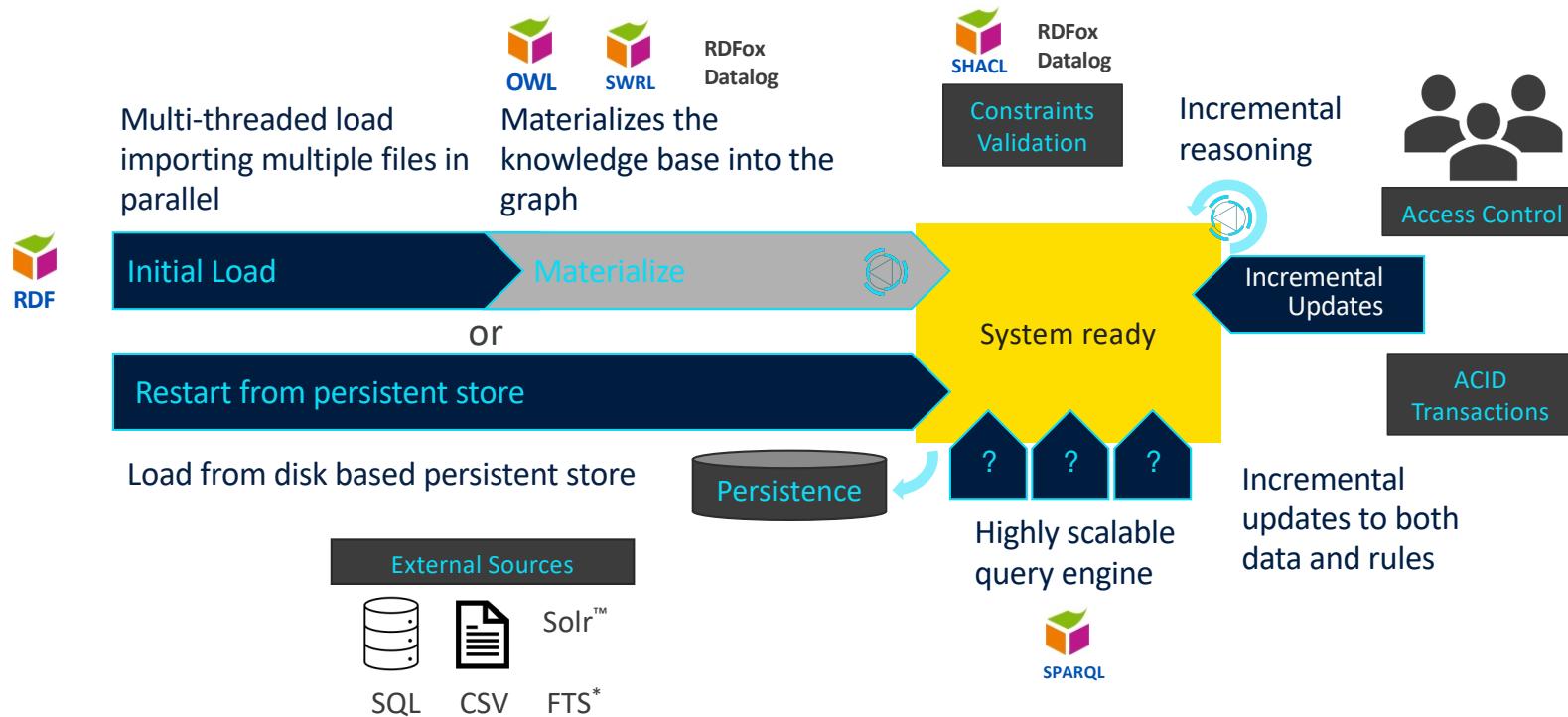


## Reasoning

- ✓ Basic rules
- ✓ Aggregates
- ✓ Negation
- ✓ Filters
- ✓ Binds
- ✓ Incremental reasoning

## Deployment

# RDFox: How it works



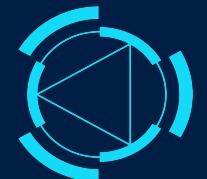


# Info command

```
> active f1
Data store connection 'f1' is active.
> info

::RDFStore
-----
Name          : f1
ID            : 0      ( 0      )
Unique ID     : 10367597292197999215
Data store version : 8      ( 8      )
End resource ID   : 1,932,735,283    ( 1.9 G)
Concurrent     : yes
Persistent     : no
Equality axiomatization mode : off
Requires incremental reasoning : yes
Aggregate size   : 262,783,450    (262.7 M)
Aggregate number of entries : 4,746,241    ( 4.7 M)
Bytes per entry   : 55.366
Aggregate number of EDB facts  : 4,708,612    ( 4.7 M)
Aggregate number of IDB facts  : 4,738,828    ( 4.7 M)
Dictionary size (%) : 18.702
Equality manager size (%) : 0      ( 0      )
Rule index size (%) : 0.140
Facts size (%)   : 81.297
> |
```

# Deployment options



Edge and mobile devices

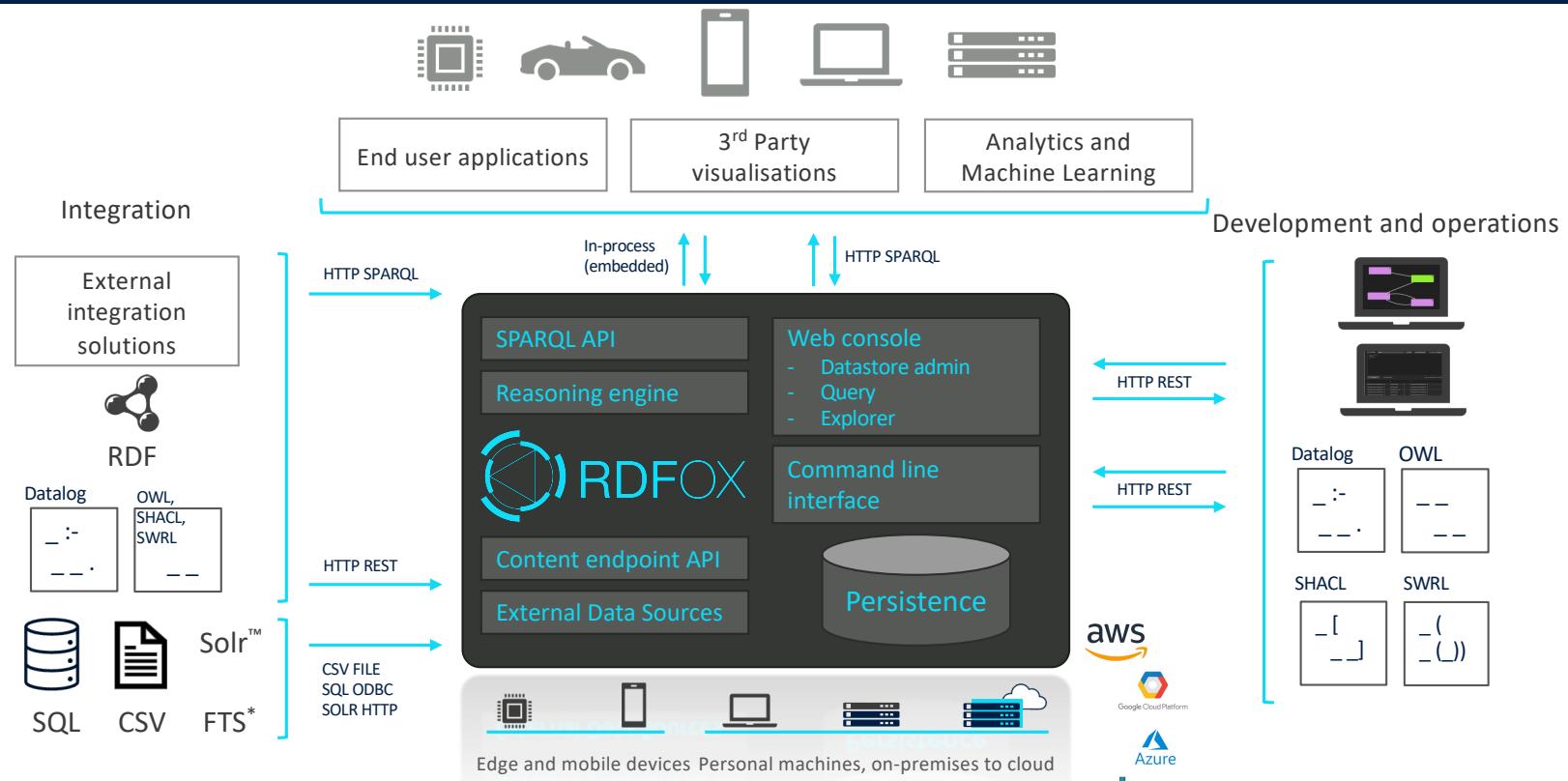
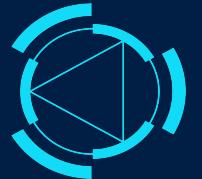


Personal machines, on-premises to cloud



# Architecture

## RDFox system context diagram



# Revisiting the objectives for today



- Setting up RDFox
- Loading data into RDFox
- Exploring and querying with SPARQL
- Reasoning
- Deployment

## Setting up RDFox

- ✓ License and executable
- ✓ IDE
- ✓ REST endpoint and UI

## Loading data into RDFox

- ✓ Create datastore
- ✓ Import data

## Exploring and querying with SPARQL

- ✓ Basics of SPARQL
- ✓ Useful queries to explore
- ✓ Aggregates
- ✓ Negation
- ✓ Filters
- ✓ Binds
- ✓ Optionals

## Reasoning

- ✓ Basic rules
- ✓ Aggregates
- ✓ Negation
- ✓ Filters
- ✓ Binds
- ✓ Incremental reasoning
- ✓ OWL

## Deployment

# The end



- Stop your RDFox server with the `quit` command

## Your final exercise...



- Before we close out with our last words of advice, we'd really appreciate it if you gave us your feedback.
- We want to keep improving this class for future students, so we really do want to hear everything you have to say—the good, the bad, and the ugly.
- Click on the link below to start filling out our quick survey. It won't take you more than 3 minutes!

<https://oxfordsemantictech.typeform.com/to/FDXQHTYJ>



# Further resources

Our website

<https://www.oxfordsemantic.tech>

Request an evaluation license

<https://www.oxfordsemantic.tech/tryrdfォrforfree>

Read the documentation

<https://docs.oxfordsemantic.tech/>

Our blog

<https://www.oxfordsemantic.tech/the-blog>

© Oxford Semantic Technologies 2022



RDFox's improved OWL implementation

RDFox V5: A short tutorial

Valerio Cocco  
Apr 19 - 1 min read



Assessing credit card risk with RDFox rules

Every year, credit card fraud causes massive losses for banks, businesses and their customers, and prevention is a constant race between...

Diana Marks  
Apr 6 - 8 min read



Getting started using the new web console

RDFox 5.0  
Diana Marks  
Mar 30 - 6 min read

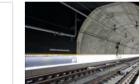


What is New in RDFox

Version 5?

Version 5 is now live...

Felicity Mulford  
Mar 26 - 1 min read



Transform Disparate Engineering Data into Structured Knowledge

Ensure your maintenance strategies are well-informed with RDFox

Caitlin Woods  
Mar 22 - 11 min read



Maintaining Market Integrity

Trade Surveillance using RDFox

Felicity Mulford  
Mar 9 - 9 min read



Humans learn using rules and relationships, so can computers

Artificial Intelligence and Semantic Reasoning

Felicity Mulford  
Jan 6 - 7 min read



Researching how human knowledge can be taught to machines

Professor Cuenca Grau — Reasoning Over Knowledge Graphs

Bernardo Cuenca Grau  
Dec 16, 2020 - 6 min read



Improving smartphone recommendation services, without data security risk

An on-device context-aware recommendation engine

Felicity Mulford  
Nov 25, 2020 - 4 min read



Smart solutions for determining compatibility with metaphactory and RDFox

Creating smart applications for configuration management

Felicity Mulford  
Nov 18, 2020 - 8 min read



Part Two: Music Streaming Services with RDFox

Validate and query large datasets without compromising performance

Felicity Mulford  
Nov 10, 2020 - 5 min read



Part One: Music Streaming Services with RDFox

Linking and enriching large datasets without compromising performance

Felicity Mulford  
Nov 6, 2020 - 6 min read