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:

- Setup 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

Deployment

Exercises

We will go through some exercises for each objective:

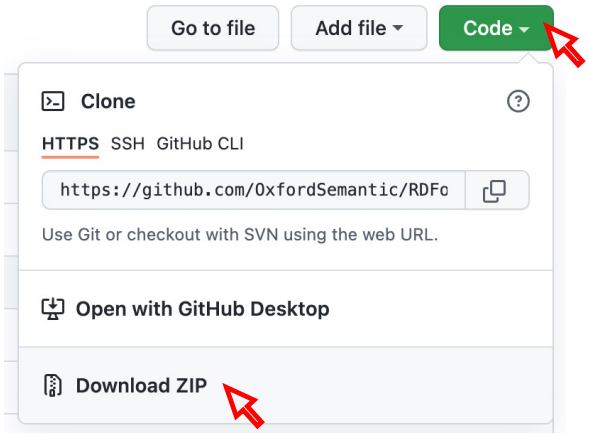
- Starting with a short explanation of each point.
- Everyone will then be able to do a short exercise. Advanced users will be able to also run through harder, bonus exercises.
- Then we regroup to check that everyone was able to complete the task

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

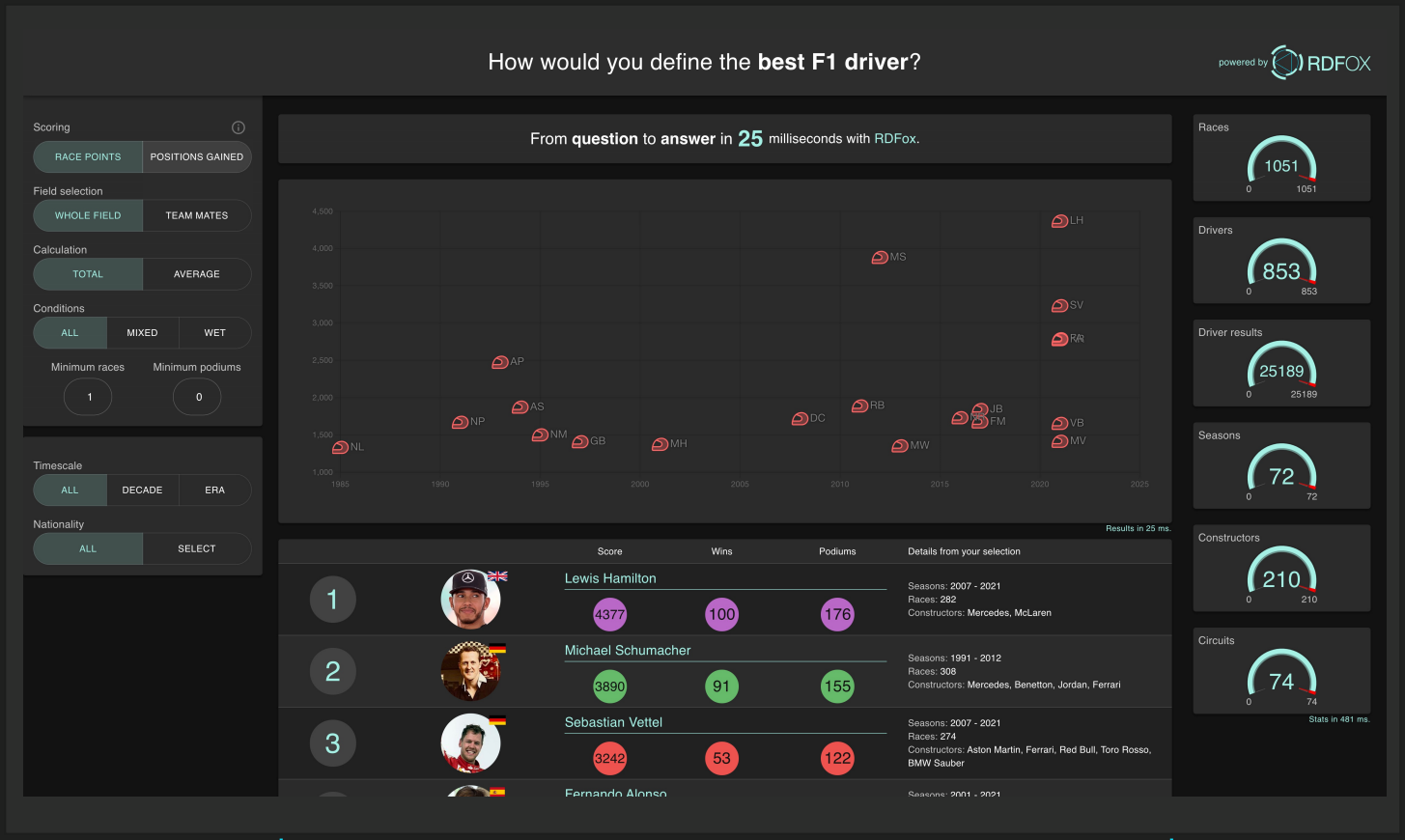


or

```
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.

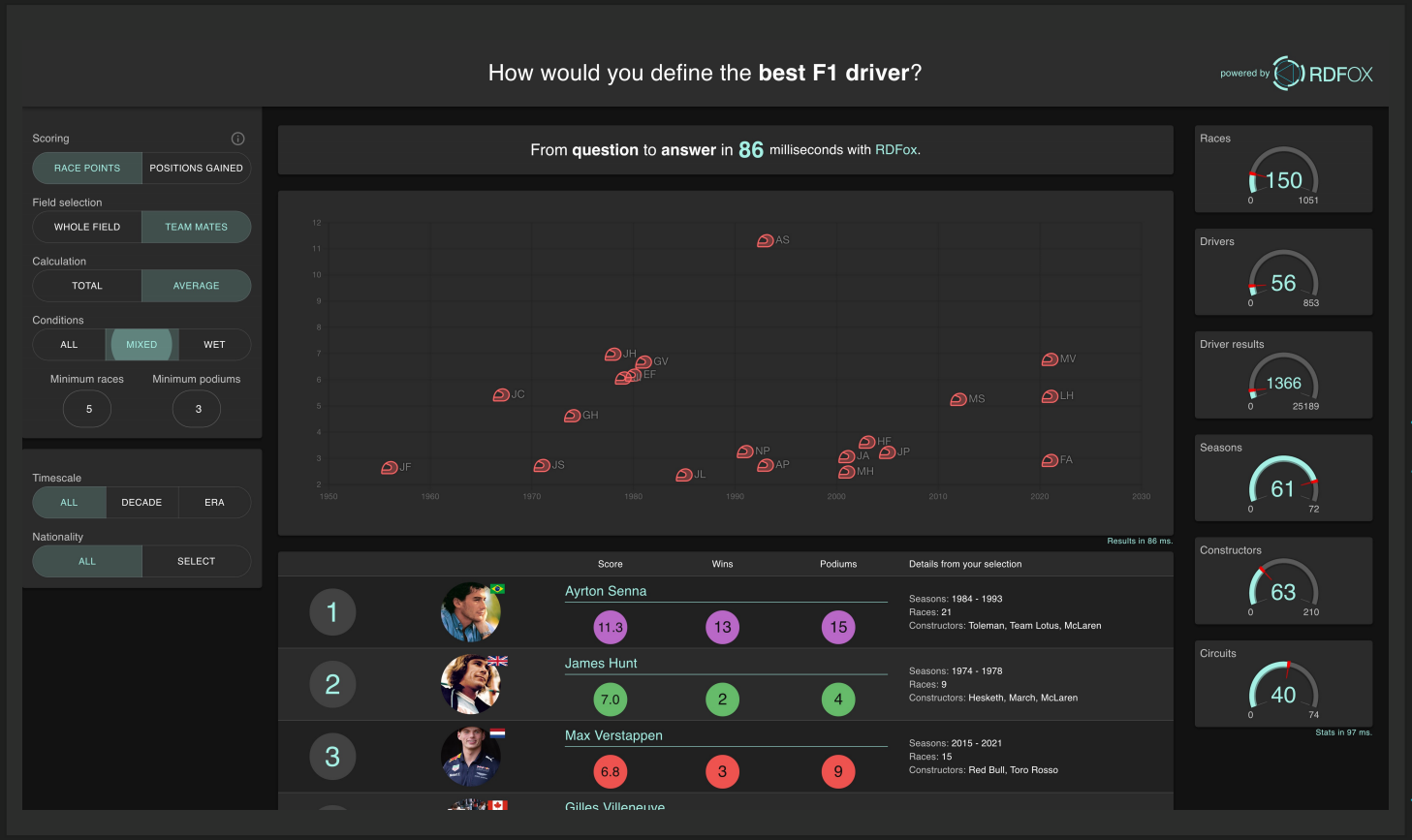


Statistics about the data used to form the results.

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.

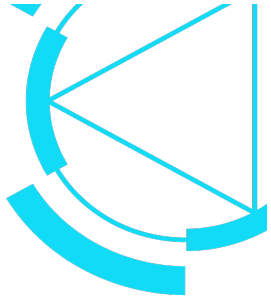


This computation takes milliseconds, despite the vast network of data involved.

Try it for yourself

http://f1.rdfox.tech

Setting up RDFox



Setting up RDFox

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

Loading data into RDFox

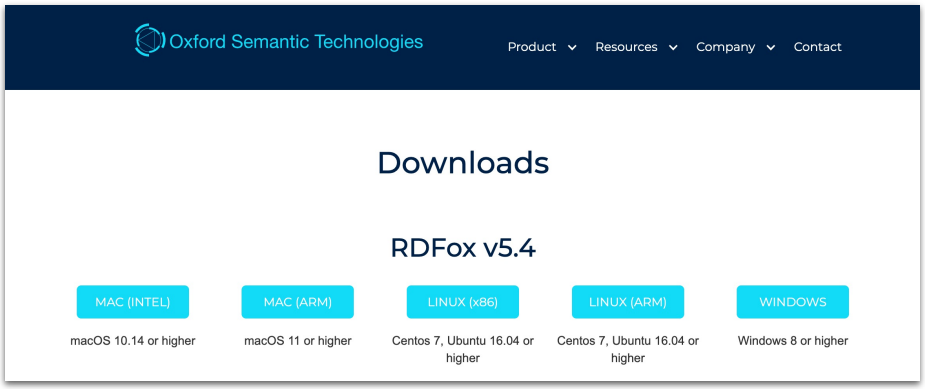
- ❑Create datastore
- ❑Import data

Exploring and querying with SPAR

- ❑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 the RDFox package from: https://www.oxfordsemantic.tech/downloads
- Unzip the package, and put the license in it.
- You might need to right-click on the RDFox executable to open it on Mac, for OS security reasons



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:

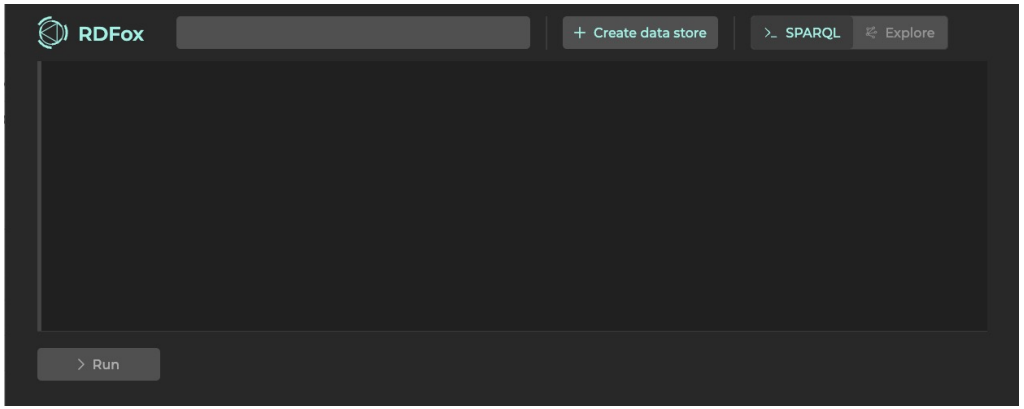
  `<Path/to>/RDFox sandbox`
- The RDFox server is now running, but it's empty…

```
> 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.
>
```

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

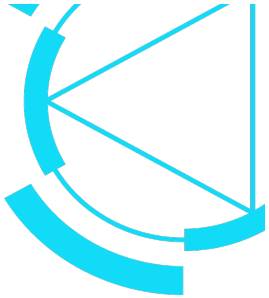
The REST Endpoint and UI

- In the terminal, run:
  `endpoint start`
- The RDFox endpoint is now running, so we can use the web UI
- Open a browser and go to:
  localhost:12110/console
- This will show an empty console



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

Loading data into RDFox



Setting up RDFox
- ✓ License and executable
- ✓ IDE
- ✓ REST endpoint and UI

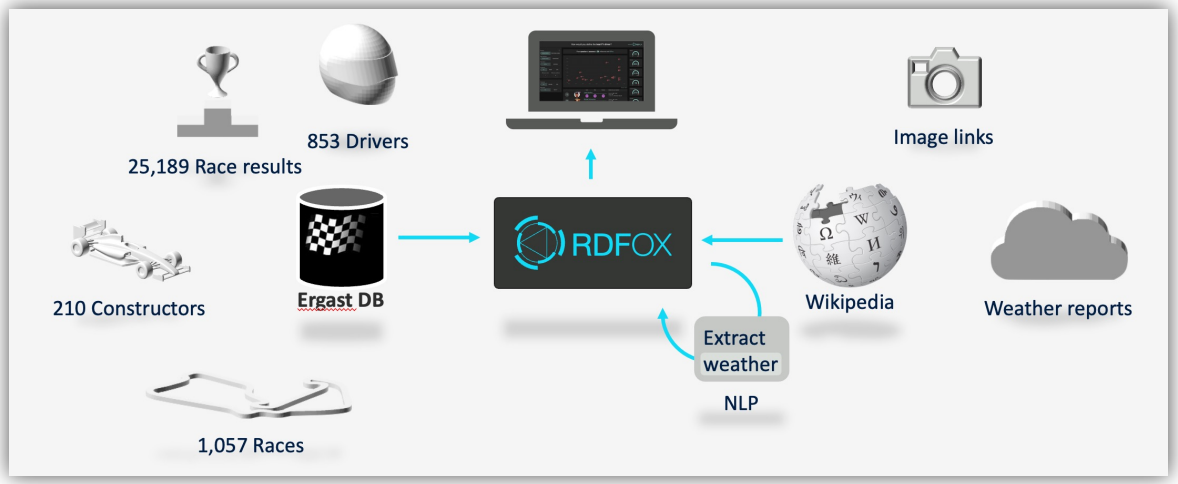
Loading data into RDFox
- ❑ Create datastore
- ❑ Import data

Exploring and querying with SPARQL
- ❑ Basics of SPARQL
- ❑ Useful queries to explore
- ❑ Aggregates
- ❑ Negation
- ❑ Filters
- ❑ Binds
- ❑ Optionals

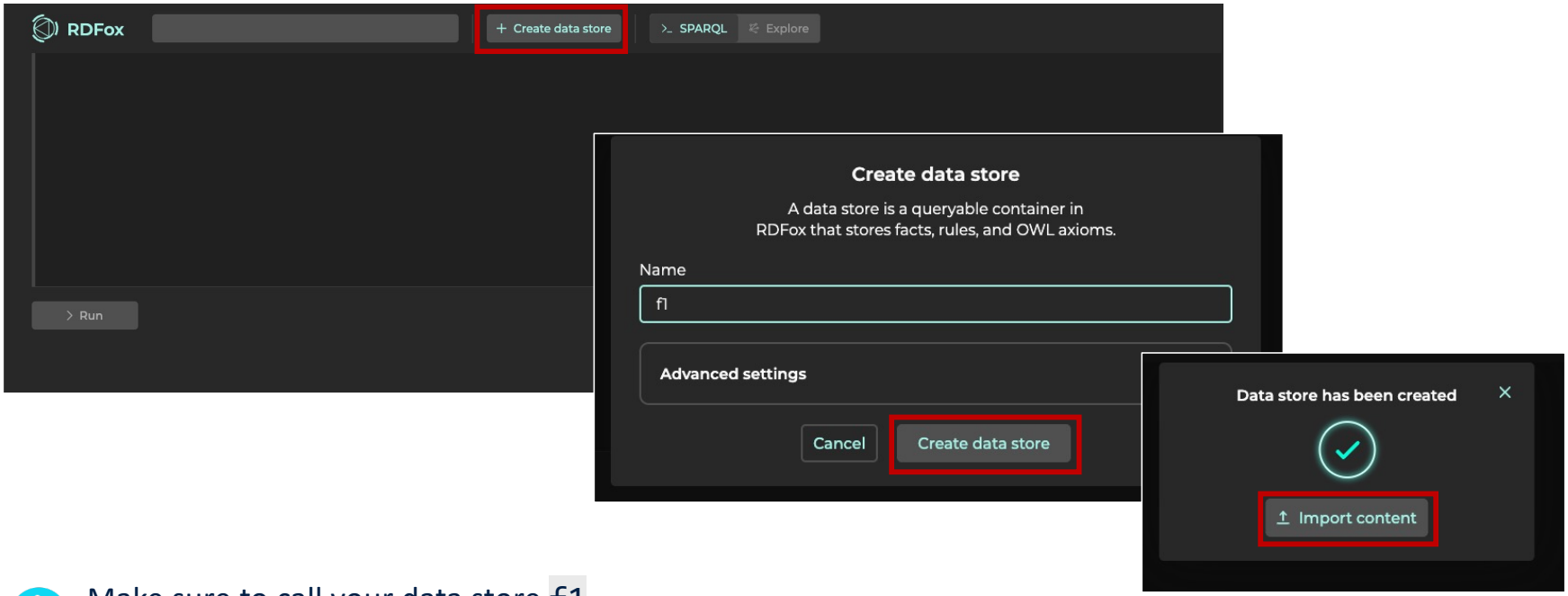
Reasoning
- ❑ Basic rules
- ❑ Aggregates

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



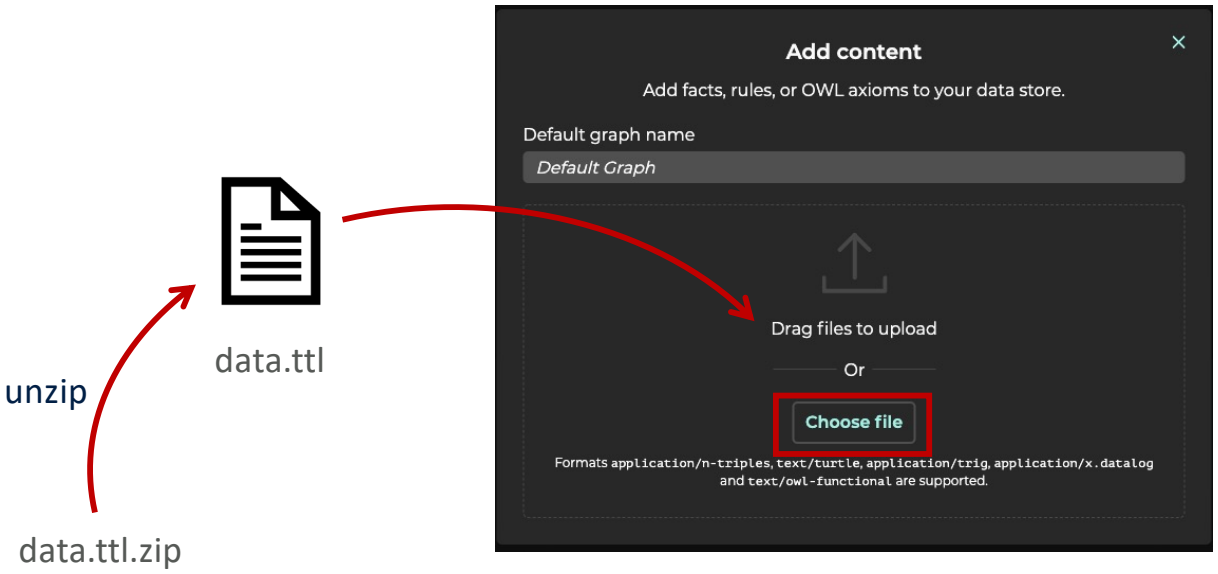
Loading data into RDFox – Create datastore



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

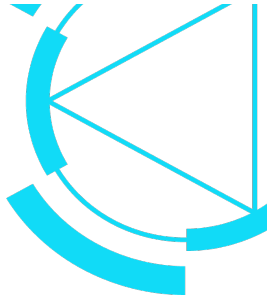
https://docs.oxfordsemantic.tech/getting-started.html#getting-started-with-the-web-console

Loading data into RDFox – Load files



RDF

Exploring and querying with SPARQL



- ✓ IDE
- ✓ REST endpoint and UI

Loading data into RDFox

- ✓ Create datastore
- ✓ Import data

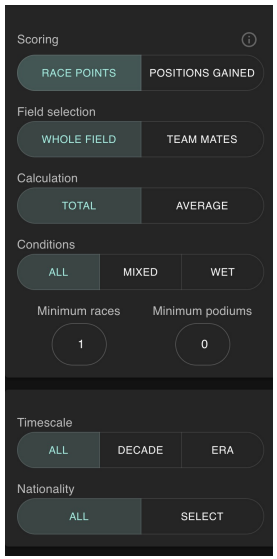
Exploring and querying with SPARQL

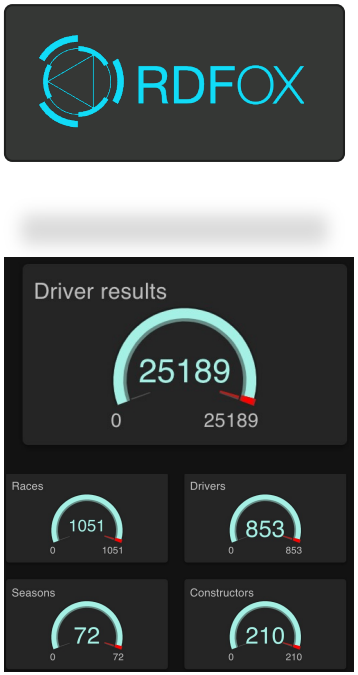
- ❑ Basics of SPARQL
- ❑ Useful queries to explore
- ❑ Aggregates
- ❑ Negation
- ❑ Filters
- ❑ Binds
- ❑ Optionals

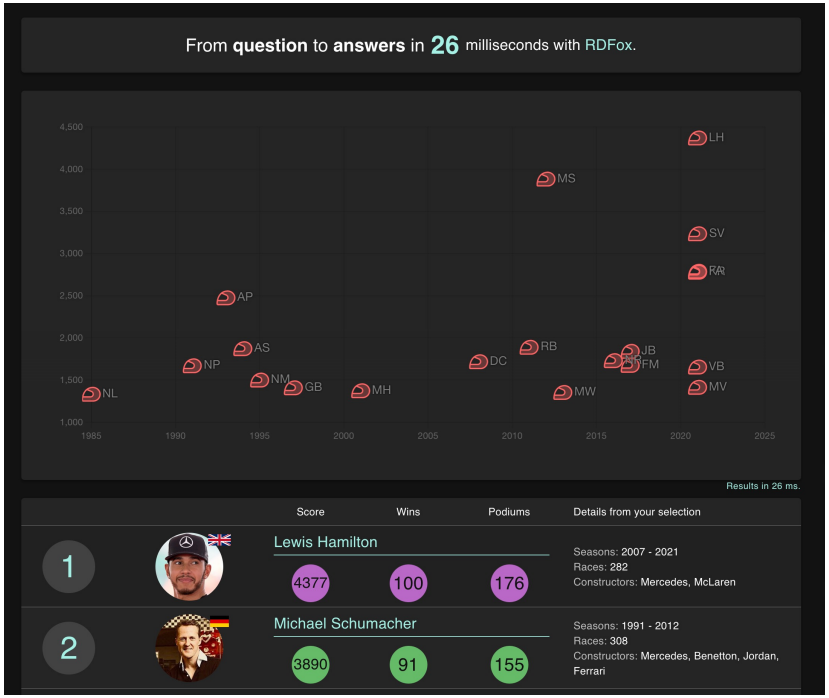
Reasoning

- ❑ Basic rules
- ❑ Aggregates
- ❑ Negation
- ❑ Filters
- ❑ Binds

From question to answer in milliseconds



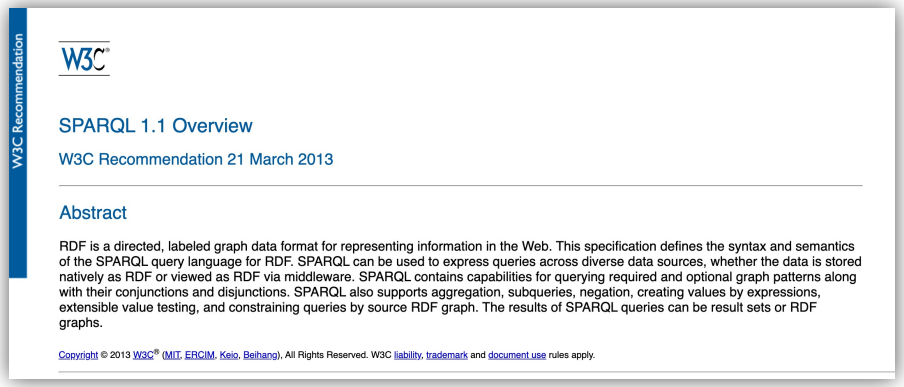




SPARQL “used to express queries”

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.

Exercise 1

Run through queries 1-4 in the web console.

Write them out yourself. See how and what they do.

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 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 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.

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.

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.

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

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 }
```

Subjects can be shared across multiple lines.

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

The last ends with a '.'.

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

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.

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.

Exercise 3

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

Query 6 – A Driver's Properties

Incomplete Query

Answer

To build on what we've got so far, now find all of the associated properties and corresponding values of a specific driver. Let's 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 ?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.

Exercise 4

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

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 instance of a variable, ?race, and binds the resulting value as another variable, ?raceCount.

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

HINT: 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.

HINT: 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.

Exercise 6

Complete Query 9 to find all of the drivers who have never finished on the podium. To finish on the podium you must come in positions 1, 2, or 3.

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

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.

```
PREFIX : <http://www.oxfordsemantic.tech/f1demo/>

# Drivers who never got a podium
SELECT ___ ___
WHERE {
# First, get all the drivers with their names.
    ?driver a :drivers ;
        :drivers_forename ?forename ;
        :drivers_surname ?surname .

# Then make sure that they have never achieved a podium (i.e. positions 3, 2, or 1)
    FILTER NOT EXISTS{
        ?result ___ ?driver ;
                :results_positionOrder ?positionOrder .
        FILTER(?positionOrder IN(1, 2, 3))
        }
    }
```

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.

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

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 win.

Query 10 – Most Races Without a Podium

Incomplete Query

Answer

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

```
PREFIX : <http://www.oxfordsemantic.tech/f1demo/>

# Most races run without ever getting a podium
SELECT ?forename ?surname ?raceCount
WHERE {
# First, get all the drivers with their names.
# The order of the query atoms is mostly not important, as RDFox will rearrange it
# internally in order to have the best performance automatically.
    ?driver a :drivers ;
        :drivers_forename ?forename ;
        :drivers_surname ?surname .

# Then make sure that they have never achieved a podium (i.e. positions 3, 2, or 1)
    FILTER NOT EXISTS{
        ?result2 :results_driver ?driver ;
                 :results_positionOrder ?positionOrder .
        FILTER(?positionOrder IN(1, 2, 3))
    }

# Finally find out how many races they have raced in.
# This is done in an 'inner query'
    {SELECT ___ (COUNT(?___) AS ?raceCount)
    WHERE {
        ?result :results_driver ?driver ;
            :results_race ?race .
    }
    GROUP BY ___
    }
}
ORDER BY DESC(___) # Don't forget to order
```

Exercise 8

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

Query 11 – Driver Win Percentage

Incomplete Query

Answer

The BIND function sets a variable to a specific value.

```
PREFIX : <http://www.oxfordsemantic.tech/f1demo/>

# Drivers with their win percentage, ordered by win percentage
SELECT ?forename ?surname ?raceCount ?raceWins ?percentage
WHERE {
# First get the drivers
?driver :drivers_forename ?forename ;
        :drivers_surname ?surname .

# Then get the race count for each driver with an innery query...
    {SELECT ?driver (COUNT(?race) AS ?raceCount)
    WHERE {
    ?result :results_driver ?driver ;
            :results_race ?race .
    }
    GROUP BY ?driver}

# ... and get the *win* count for each driver with another inner query.
    {SELECT ___ (COUNT(?race) AS ___)
    WHERE {
    ?result :results_driver ___ ;
            :results_race ___ ;
            :results_positionOrder 1 .
    }
    GROUP BY ?driver}

# Finally use the two aggregate variables to compute a percentage
# with the BIND keyword.
BIND(?raceWins/___ AS ?percentage)

}
ORDER BY DESC(?percentage)
```

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 Qs

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

Answer

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 aggregatee variables to compute a percentage
35 BIND(___/?raceCount AS ?percentage)
36
37 }
38 ORDER BY DESC(?percentage)
```

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.

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

Exercise 10

From the RDFox command line:

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

b. Set the output to the screen

c. 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`

BREAK

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

Reasoning

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

Deployment

Reasoning

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

Reasoning

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

Deployment

Reasoning

[*fact A…*] :- [*fact B…*]

“A… is true if B… is true”

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

Rule 1 – Basic Rule

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

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 will need it later on.

Rule 2 – FILTERs in rules

- 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

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
        ) .
```

Rule 4 – Another aggregate Rule

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.

- aggregate rule (r2, to compute race counts)
- Then, fill in the blanks in the second aggregate rule (`r3.dlog`)
- Finally, import `r3.dlog`

Rule 5 – Negation as Failure

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]) .
```

Rule 5 –About Negation as Failure

Negation as failure is particularly useful for many of our customers.
However it is also quite difficult to implement, which is why it is not in OWL.

This is because having it 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.

Exercise 15 – BIND rules

- Import r6.dlog and check its inferred triples with a query
- Write a new rule to find out if a constructor has never won a podium, using the `:ConstructorWithoutPodium` class
- Import the rule
- Run a query to check whether it added any triples.

Rule 6 – BIND rules

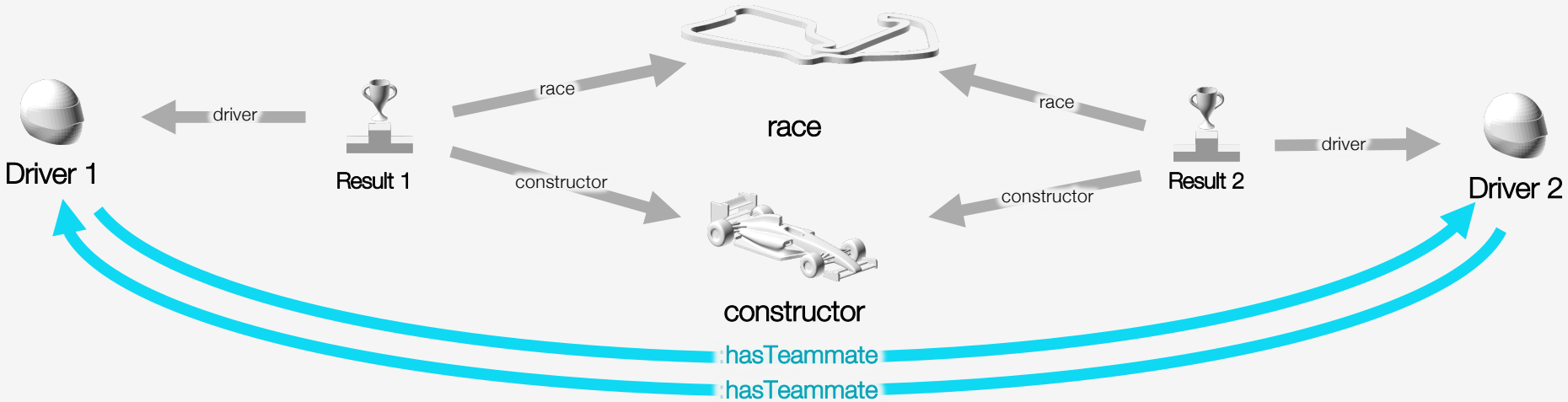
Using `BIND` in rules helps us to perform numerical reasoning. We can also do string manipulation.

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

Exercise 16 – BIND rules

- Fill in the blanks in Rule 6
- Import the rule
- Run a query to check whether it added any triples.

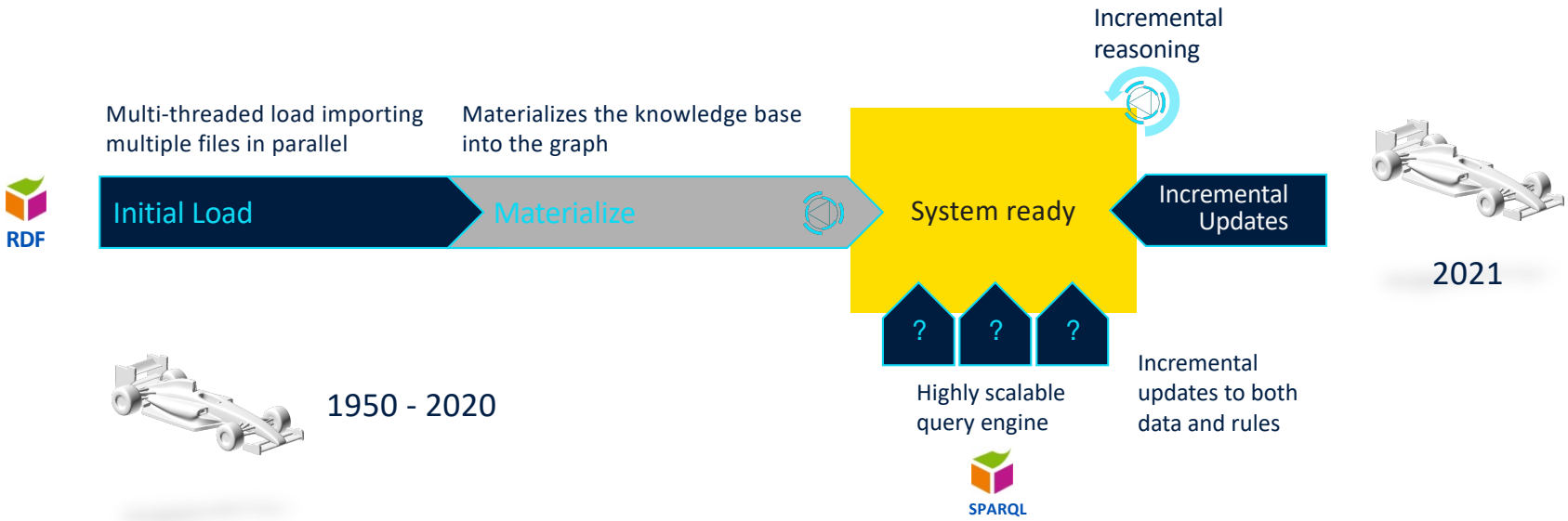
Establishing teammate links with rules



Bonus Exercise

- 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.

RDFox reasoning vs. other solutions

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

*** many times slower** than RDFox
**** orders of magnitude slower** than RDFox

Deployment

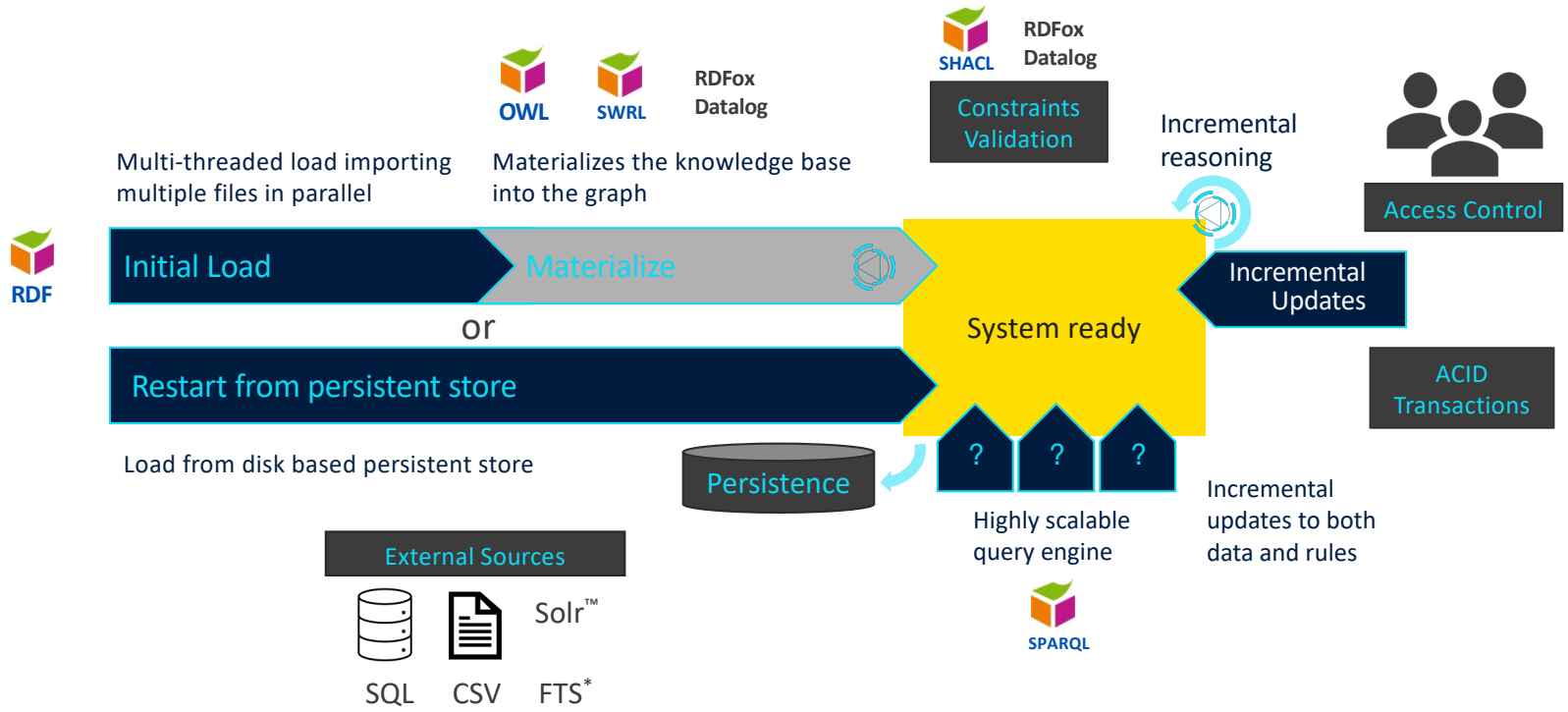
- ✓ Basics of SPARQL
- ✓ Useful queries to explore
- ✓ Aggregates
- ✓ Negation
- ✓ Filters
- ✓ Binds
- ✓ Optionals

Reasoning

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

Deployment

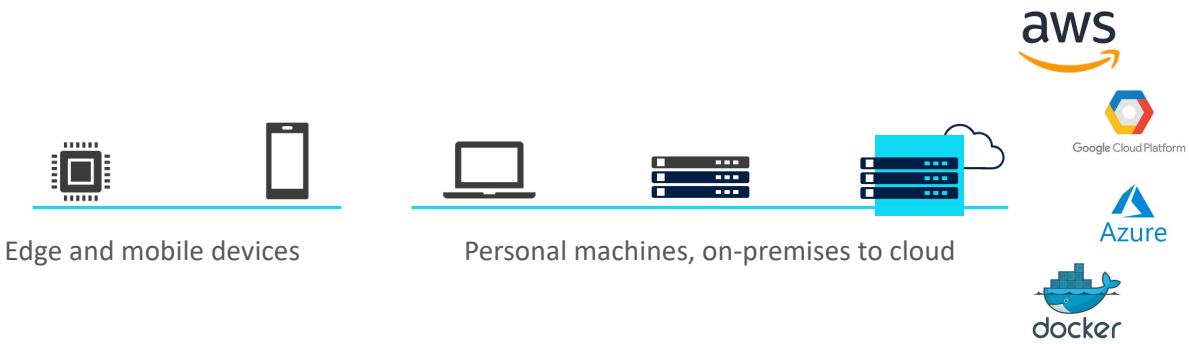
RDFox: How it works



Info command

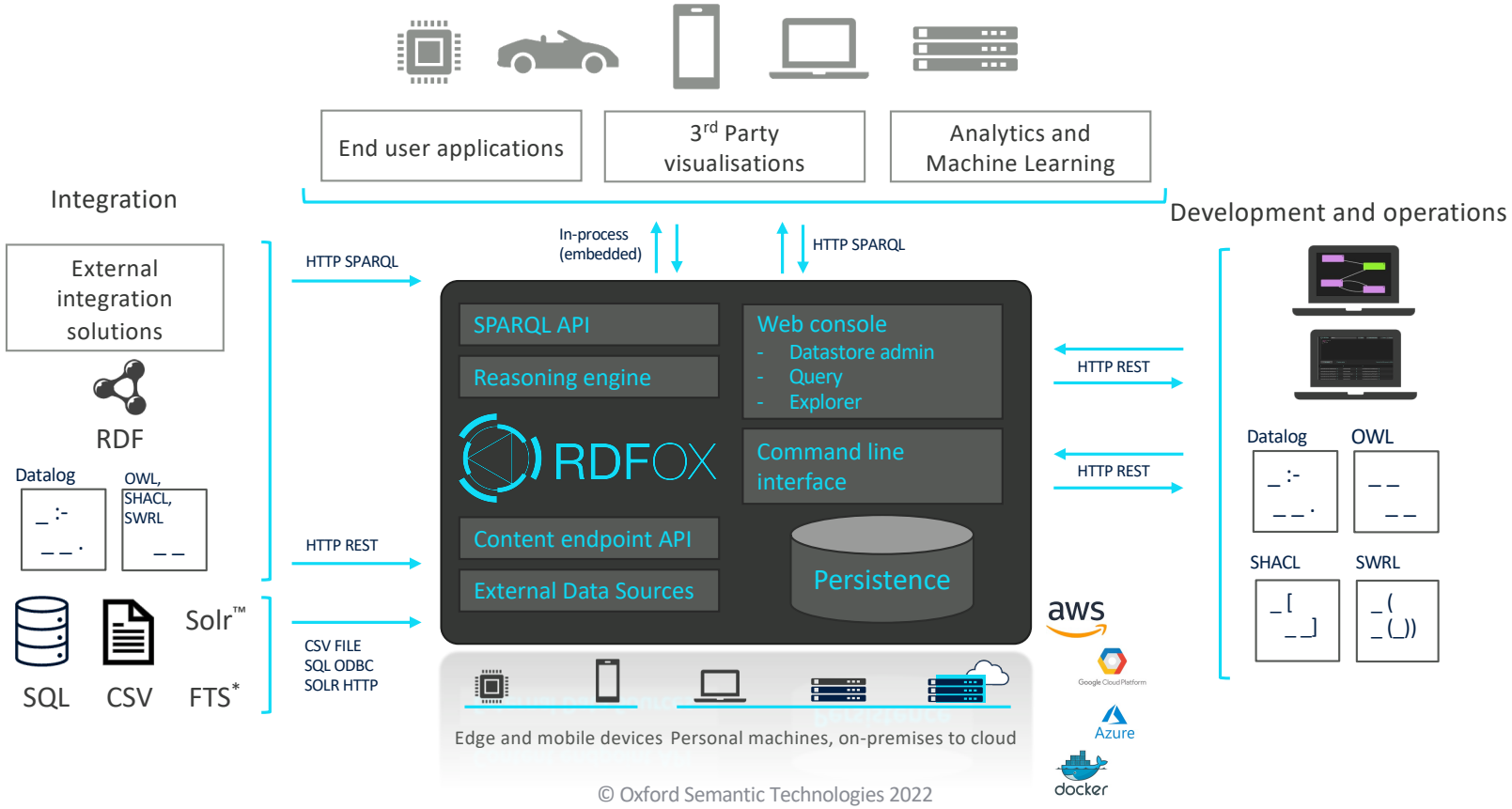
```
> active f1
Data store connection 'f1' is active.
> info


  ::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



Architecture

RDFox system context diagram



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

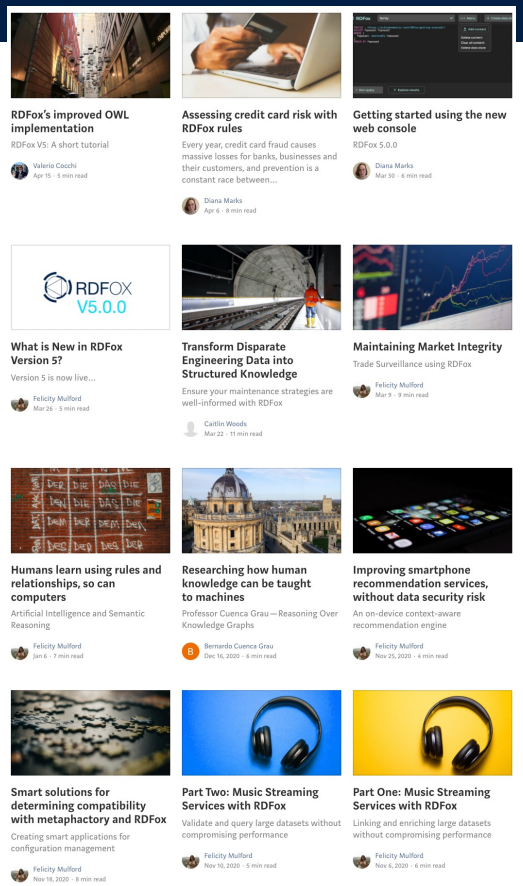
Further resources

Our website
https://www.oxfordsemantic.tech

Request an evaluation license
https://www.oxfordsemantic.tech/tryrdfoxforfree

Read the documentation
https://docs.oxfordsemantic.tech/

Our blog
https://www.oxfordsemantic.tech/the-blog



Active command

- Sets the active data store

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

When you start RDFox the active data store is called “default”

https://docs.oxfordsemantic.tech/command-line-reference.html