Querying APIs with SPARQL: language and worst case optimal algorithms

Matthieu Mosser, Fernando Pieressa, Juan Reutter, Adrián S., Domagoj Vrgoč

Pontificia Universidad Católica de Chile
Millenium Institute for Foundational Research on Data

Outline

Introduction

Preliminaries

SPARQL and JSON API calls

WCO Algorithm

Experiments

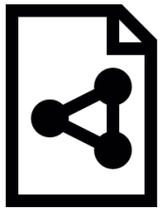
Conclusion

Introduction

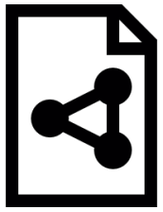
Introduction

Introduction

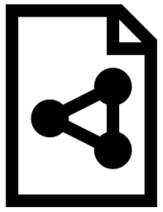
Introduction

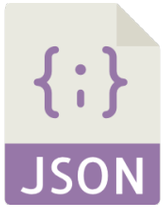


Introduction

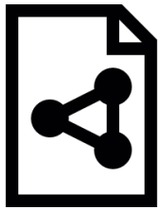


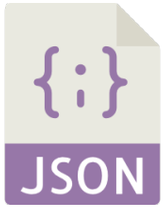
Introduction





Introduction





Introduction

The majority of the data available on the Web today is still not published in RDF

Introduction

The majority of the data available on the Web today is still not published in RDF

How can we integrate such data?

Introduction

The majority of the data available on the Web today is still not published in RDF

How can we integrate such data?

How can we do that as efficient as possible?

Introduction

In this talk

A way to enable SPARQL to integrate data from JSON Web APIs extending the SERVICE operator

Introduction

In this talk

A way to enable SPARQL to integrate data from JSON Web APIs extending the SERVICE operator

An algorithm to perform this task as efficient as possible

Introduction

In this talk

A way to enable SPARQL to integrate data from JSON Web APIs extending the SERVICE operator

An algorithm to perform this task as efficient as possible

Experimental results for the algorithm

Outline

Introduction

Preliminaries

SPARQL and JSON API calls

WCO Algorithm

Experiments

Conclusion

Preliminaries
SPARQL

We use the usual syntax and semantics of SPARQL, considering:

Tripple patterns
AND
OPTIONAL
UNION
FILTER
SERVICE

Preliminaries

SERVICE

Preliminaries

SERVICE

```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

SERVICE

```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

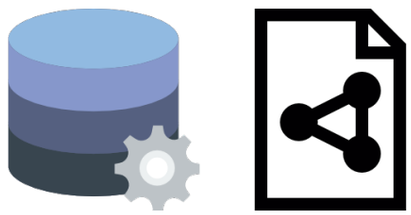
Preliminaries

SERVICE

```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

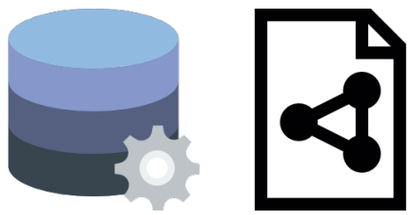
SERVICE



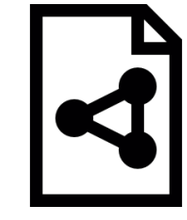
```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

SERVICE



```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```



Preliminaries

SERVICE

```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

SERVICE

```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

SERVICE

```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

SERVICE

```
SELECT ?name ?weather
WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

SERVICE

Accessing multiple *Endpoints* in a single query

```
SELECT ?city ?weather WHERE
```

Preliminaries

SERVICE

Accessing multiple *Endpoints* in a single query

```
SELECT ?city ?weather WHERE
{
  ?city :comuneOf :Chile .
```

Preliminaries

SERVICE

Accessing multiple *Endpoints* in a single query

```
SELECT ?city ?weather WHERE
{
  ?city :comuneOf :Chile .
  SERVICE <http://weather.org/sparql>
  {
    ?city weather:weatherNow ?weather
  }
}
```

Preliminaries

Web APIs

Interfaces that expose data through HTTP requests

Preliminaries

Web APIs

Interfaces that expose data through HTTP requests

```
http://weather.org/s?lat=-33.45&lon=-70.6
```

Preliminaries

Web APIs

Interfaces that expose data through HTTP requests

```
http://weather.org/s?lat=-33.45&lon=-70.6
```

```
{
  “city”: “Santiago”,
  “status”: “Sunny”,
  “temperature”: {
    “min”: “14ºC”,
    “max”: “29ºC”
  }
}
```

Preliminaries

JSON Documents

We assume that APIs responses are JSON Documents

We retrieve data from JSONs with *navigation conditions*

Preliminaries

Navigation Conditions

For the following JSON ***J***, the navigation condition `J["temperature"]["min"]` returns `"14ºC"`

Preliminaries

Navigation Conditions

For the following JSON ***J***, the navigation condition `J["temperature"]["min"]` returns `"14ºC"`

```
{
  "city": "Santiago",
  "status": "Sunny",
  "temperature": {
    "min": "14ºC",
    "max": "29ºC"
  }
}
```

Outline

Introduction

Preliminaries

SPARQL and JSON API calls

WCO Algorithm

Experiments

Conclusion

SERVICE Extension

Example

Suppose we want to do some *hiking* in Scotland, but we want to go to mountains located in places that are sunny

SERVICE Extension

Example

```
SELECT ?x ?l WHERE {
   ?x wdt:instanceOf wd:mountain .
   ?x wdt:locatedIn wd:Scotland .
   ?x rdfs:label ?l .
   SERVICE <http://weather.api/request?q={?l}>{
      (["description"]) AS (?d)
   }
   FILTER (?d = "Sunny")
}
```

SERVICE Extension

Example

We first resolve the *bgp*:

```
?x wdt:instanceOf wd:mountain .
?x wdt:locatedIn wd:Scotland .
?x rdfs:label ?l
```

?x	?l
wd:Q104674	“Ben Nevis”
wd:Q13130255	“Mount Blair”

SERVICE Extension

Example

And then we extend each mapping for the value provided by the API

```
SERVICE <http://weather.api/request?q={?l}>{
   (["description"]) AS (?d)
}
```

?x	?l	?d
wd:Q104674	“Ben Nevis”	“Sunny”
wd:Q13130255	“Mount Blair”	“Rain”

SERVICE Extension

Example

Then we apply the filter

```
FILTER (?d = “Sunny")
```

?x	?l	?d
wd:Q104674	“Ben Nevis”	“Sunny”

SERVICE Extension

Syntax

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

SERVICE Extension

Syntax

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

P_1 is a SPARQL Pattern

SERVICE Extension

Syntax

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

U is a URI template (a URI with variables)

SERVICE Extension

Syntax

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

U is a URI template (a URI with variables)

<http://weather.api/request?q={?l}>

SERVICE Extension

Syntax

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

N_1,…, N_m are JSON Navigation Conditions

SERVICE Extension

Syntax

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

N_1,…, N_m are JSON Navigation Conditions

```
["description"], ["temperature"]["min"]
```

SERVICE Extension

Syntax

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?x_1,…, ?x_m are variables not appearing in P_1 where the JSON Values will be bound

SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

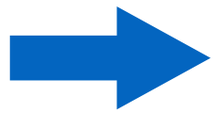
?v_1	…	?v_n
…	…	…
	μ	
…	…	…

SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?v_1	…	?v_n
…	…	…
	μ	
…	…	…

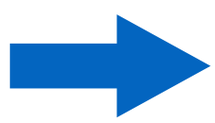


SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?v_1	…	?v_n
…	…	…
	μ	
…	…	…

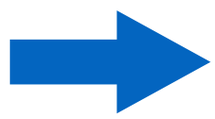


SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?v_1	…	?v_n
…	…	…
	μ	
…	…	…

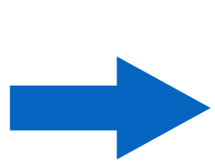


SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?v_1	…	?v_n
…	…	…
	μ	
…	…	…



SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?v_1	…	?v_n
…	…	…
	μ	
…	…	…

Call_API(U, μ)

SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?v_1	…	?v_n
…	…	…
	μ	
…	…	…

Call_API(U, μ)

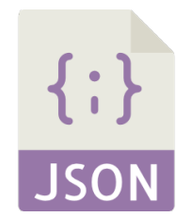
SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

?v_1	…	?v_n
…	…	…
	μ	
…	…	…

Call_API(U, μ)



SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS ($?x_1$, …, $?x_m$) }

SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS ($?x_1$, …, $?x_m$) }

SERVICE Extension

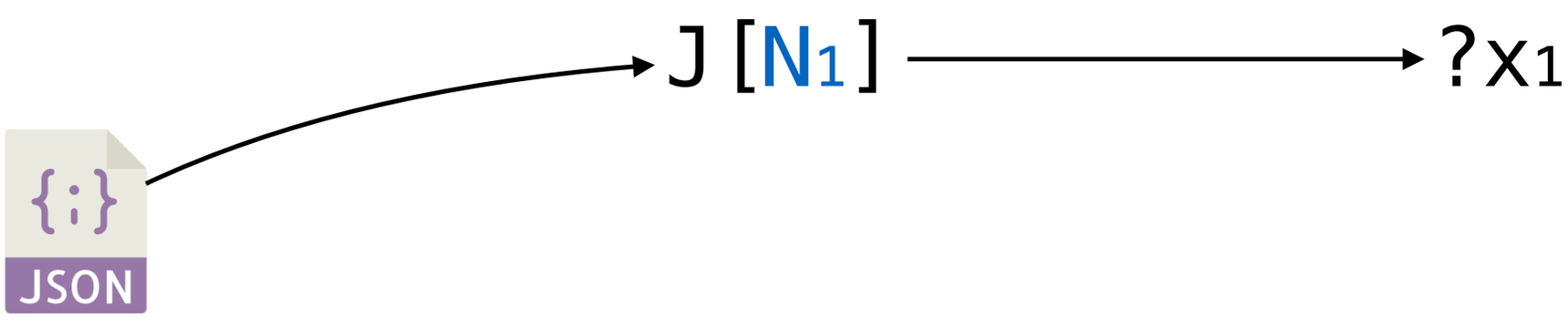
Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

SERVICE Extension

Semantics

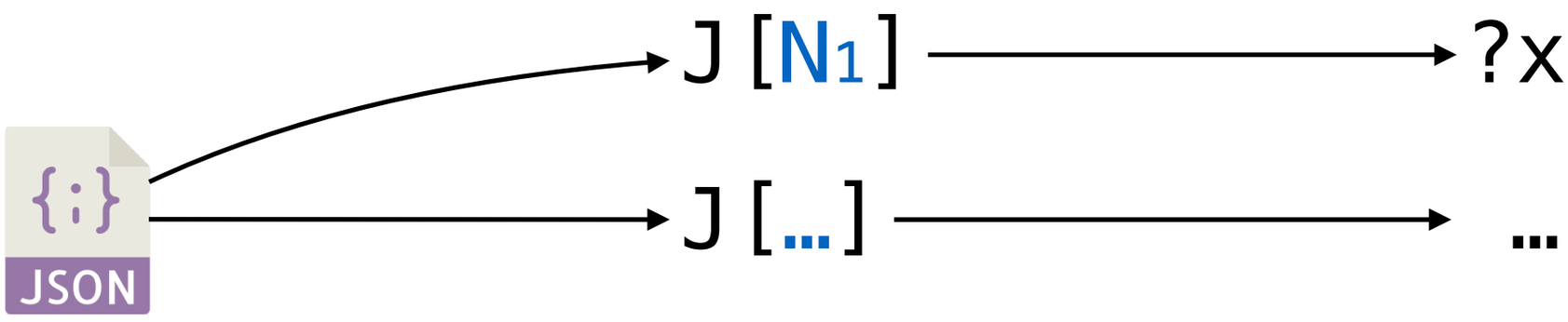
P_1 SERVICE U { (N_1,…, N_m) AS ($?x_1$, …, $?x_m$) }



SERVICE Extension

Semantics

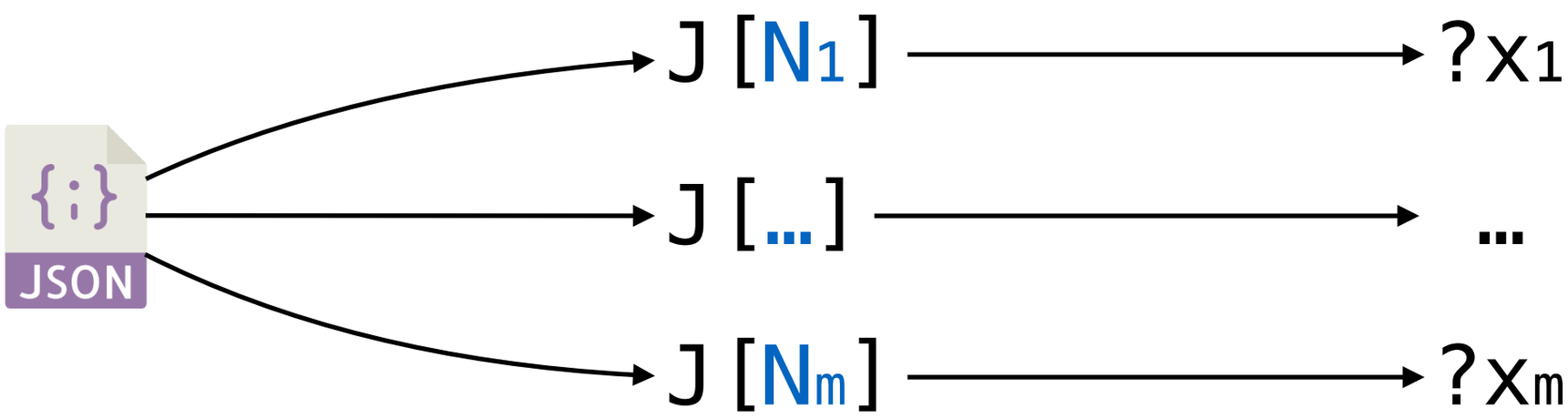
P_1 SERVICE U { (N_1,…, N_m) AS ($?x_1$, …, $?x_m$) }



SERVICE Extension

Semantics

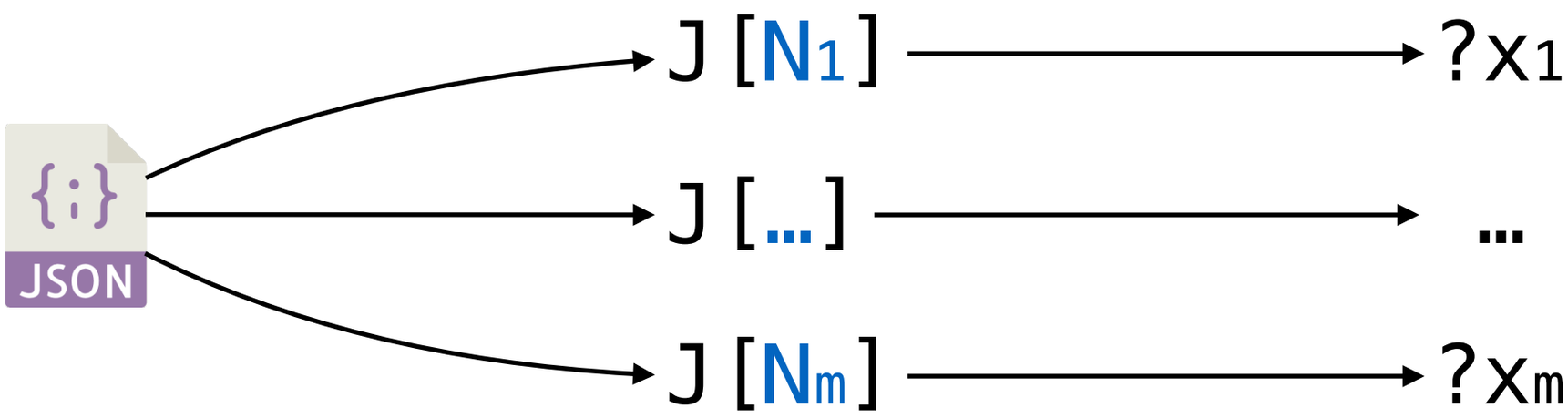
P_1 SERVICE U { (N_1,…, N_m) AS ($?x_1$, …, $?x_m$) }



SERVICE Extension

Semantics

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }



The mapping μ is extended with variables ?x_1, …, ?x_m

SERVICE Extension

Basic Implementation

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

SERVICE Extension

Basic Implementation

P_1 SERVICE U { (N_1,…, N_m) AS (?x_1, …, ?x_m) }

- Compute P_1 (recursively if it contains another SERVICE-to-API)
- Compute a set of Mappings **M**, where each mapping of P_1 is extended with the new variables
- Serialize each mapping of **M** with VALUES, and to allow them to be used by the next graph pattern

Outline

Introduction
Preliminaries
SPARQL and JSON API calls
WCO Algorithm
Experiments
Conclusion

Improving the algorithm

In a classical setting, we would like to minimize the calls to disk

Improving the algorithm

In a classical setting, we would like to minimize the calls to disk

But what is the main bottleneck for SERVICE-to-API queries?

Improving the algorithm

In a classical setting, we would like to minimize the calls to disk

But what is the main bottleneck for SERVICE-to-API queries?

We ran 4 queries with real use cases of SERVICE-to-API

Improving the algorithm

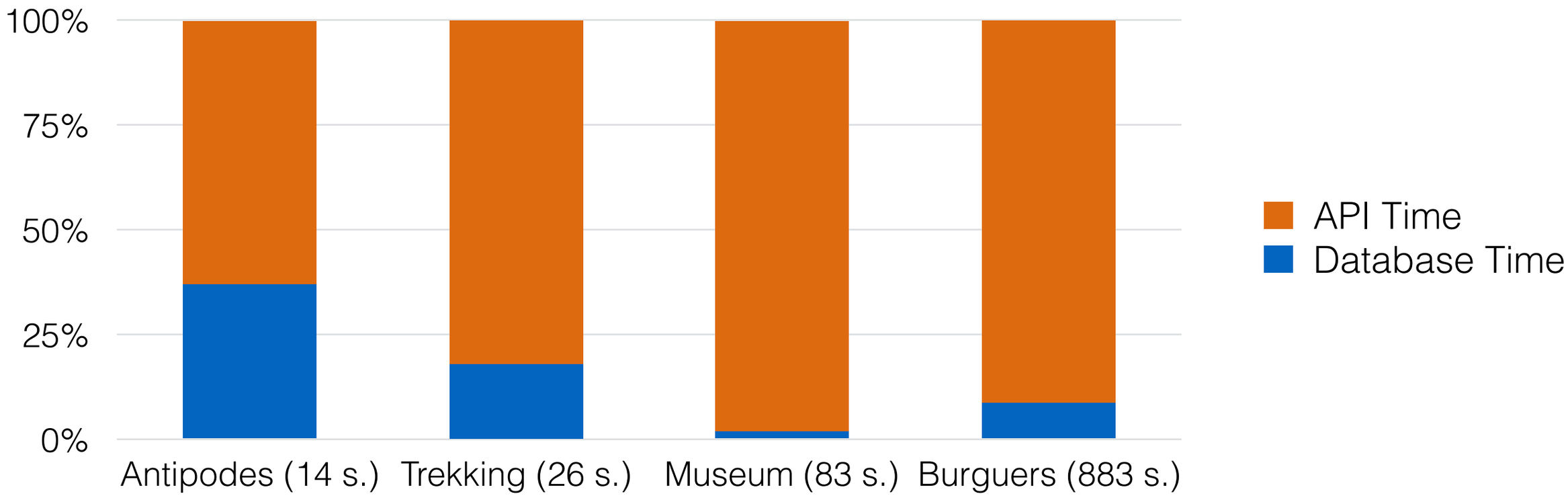


Figure 1: queries that used the APIs of OpenWeather, Yelp and Twitter

Worst Case Optimal Algorithm

In this setting, the main bottleneck is the number of calls done to the APIs

Worst Case Optimal Algorithm

In this setting, the main bottleneck is the number of calls done to the APIs

We developed a WCO Algorithm in terms of the number of API Calls

WCO Algorithm

Example

Suppose the following query:

```
SELECT ?x ?l ?d WHERE {
  ?x ex:prop1 ex:c1 .
  ?x rdfs:label ?l .
  SERVICE <http://weather.api/request?q={?l}>{
    (["description"]) AS (?d)
  }
  ?x ex:prop2 ex:c2
}
```

WCO Algorithm

Example

Where the following pattern:

WCO Algorithm

Example

Where the following pattern:

```
?x ex:prop1 ex:c1 .
?x rdfs:label ?l
```

WCO Algorithm

Example

Where the following pattern:

```
?x ex:prop1 ex:c1 .
?x rdfs:label ?l
```

has an output of size **n**, but the following pattern:

WCO Algorithm

Example

Where the following pattern:

```
?x ex:prop1 ex:c1 .
?x rdfs:label ?l
```

has an output of size **n**, but the following pattern:

```
?x ex:prop2 ex:c2
```

WCO Algorithm

Example

Where the following pattern:

```
?x ex:prop1 ex:c1 .
?x rdfs:label ?l
```

has an output of size **n**, but the following pattern:

```
?x ex:prop2 ex:c2
```

is empty

WCO Algorithm

Example

The basic algorithm would do **n** calls to the API, but the output of the query is empty!

WCO Algorithm

Example

The basic algorithm would do **n** calls to the API, but the output of the query is empty!

The WCO algorithm selects an order for the variables, and resolves the patterns in such order

WCO Algorithm

Example

Suppose the order of variables `?x`, `?l`, `?d`, then the algorithm evaluates the patterns in the following order

WCO Algorithm

Example

Suppose the order of variables `?x`, `?l`, `?d`, then the algorithm evaluates the patterns in the following order

?x

```
?x ex:prop1 ex:c1 .
?x ex:prop2 ex:c2
```

WCO Algorithm

Example

Suppose the order of variables `?x`, `?l`, `?d`, then the algorithm evaluates the patterns in the following order

```
?x, ?l
```

```
?x ex:prop1 ex:c1 .
?x ex:prop2 ex:c2 .
?x rdfs:label ?l
```

WCO Algorithm

Example

Suppose the order of variables `?x`, `?l`, `?d`, then the algorithm evaluates the patterns in the following order

```
?x, ?l, ?d
```

```
?x ex:prop1 ex:c1 .
?x ex:prop2 ex:c2 .
?x rdfs:label ?l
SERVICE <http://weather.api/request?q={?l}>{
    (["description"]) AS (?d)
  }
```

WCO Algorithm

Now the first pattern throws an empty output before evaluating the SERVICE, and the number of calls is 0

WCO Algorithm

Now the first pattern throws an empty output before evaluating the SERVICE, and the number of calls is 0

Recall that the order of appearance of variables is always a feasible one to execute the algorithm

WCO Algorithm

Now the first pattern throws an empty output before evaluating the SERVICE, and the number of calls is 0

Recall that the order of appearance of variables is always a feasible one to execute the algorithm

This algorithm stores in cache de JSONs provided by the API, in order to avoid duplicate calls

WCO Algorithm

The algorithm is inspired by the algorithm proposed by AGM, which evaluates a query variable by variable

WCO Algorithm

The algorithm is inspired by the algorithm proposed by AGM, which evaluates a query variable by variable

$$\varphi_1 = \pi_{A_1}(R_1) \bowtie \ldots \bowtie \pi_{A_1}(R_m)$$

$$\varphi_{j+1} = \varphi_j \bowtie \pi_{A_1,\ldots,A_{j+1}}(R_1) \bowtie \ldots \bowtie \pi_{A_1,\ldots,A_{j+1}}(R_m)$$

WCO Algorithm

Upper bound

Theorem 1. Any feasible query **Q** can be evaluated over any database **D** using a number of calls in

$$O(M_{Q,D} \times 2^{\rho^*(Q,D)})$$

WCO Algorithm

Upper bound

Theorem 1. Any feasible query **Q** can be evaluated over any database **D** using a number of calls in

$$O(M_{Q,D} \times 2^{\rho^*(Q,D)})$$

$2^{\rho^*(Q,D)}$: AGM Bound of the query

WCO Algorithm

Upper bound

Theorem 1. Any feasible query **Q** can be evaluated over any database **D** using a number of calls in

$$O(M_{Q,D} \times 2^{\rho^*(Q,D)})$$

$2^{\rho^*(Q,D)}$: AGM Bound of the query

$M_{Q,D}$: maximum size of the projection of a single attribute over the database

WCO Algorithm

Lower bound

We found a (infinite) family of instances where the bound is reached

WCO Algorithm

Lower bound

We found a (infinite) family of instances where the bound is reached

The algorithm is Worst Case Optimal in terms of API Calls

Outline

Introduction

Preliminaries

SPARQL and JSON API calls

WCO Algorithm

Experiments

Conclusion

Experiments

We did experiments to check that the algorithm is not only theoretically better, but in practice too

Experiments

We did experiments to check that the algorithm is not only theoretically better, but in practice too

We adapted the Berlin Benchmark transforming some sub-patterns into SERVICE-to-API queries

Experiments

Part of the data was exposed into REST APIs

Experiments

Part of the data was exposed into REST APIs

Three algorithms were tested:

- Vanilla: the basic algorithm
- W/O duplicates: the vanilla algorithm that avoid duplicated calls
- WCO Algorithm

WCO Algorithm

Results

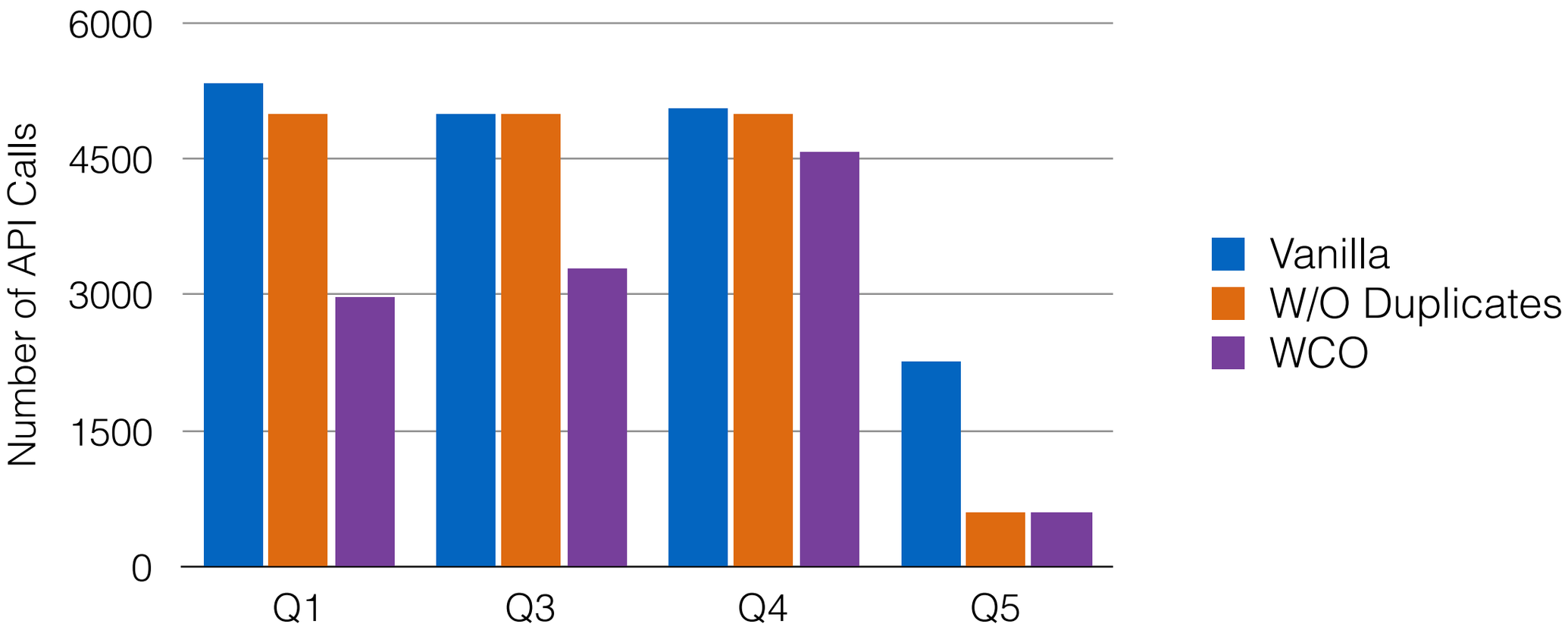


Figure 2: results for the bigger Berlin Benchmark queries

WCO Algorithm

Results

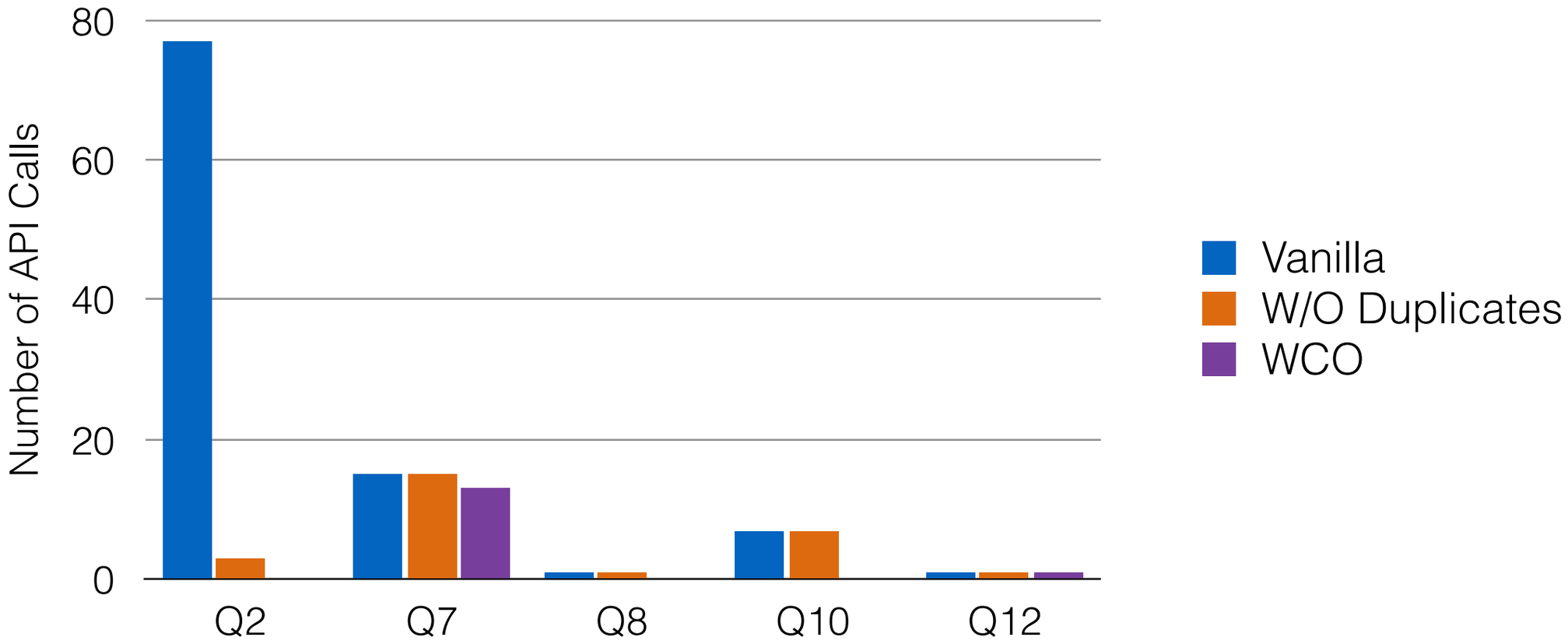


Figure 3: results for the smaller Berlin Benchmark queries

Experiments

The W/O Duplicates algorithm does 20% less calls

Experiments

The W/O Duplicates algorithm does 20% less calls

The WCO algorithm does 53% less calls

Outline

Introduction

Preliminaries

SPARQL and JSON API calls

WCO Algorithm

Experiments

Conclusion

Conclusion

The paper proposed a way to allow SPARQL queries to connect to HTTP APIs returning JSON

Conclusion

The paper proposed a way to allow SPARQL queries to connect to HTTP APIs returning JSON

We also proposed a Worst Case Optimal Algorithm to evaluate such queries

Conclusion

Future Work

As future work, we plan to support more formats than only JSON

Conclusion

Future Work

As future work, we plan to support more formats than only JSON

We also want to support automatic entity resolution based on an API answer, allowing us to transform API information back to IRIs

Querying APIs with SPARQL: language and worst case optimal algorithms

Matthieu Mosser, Fernando Pieressa, Juan Reutter, Adrián S., Domagoj Vrgoč

Pontificia Universidad Católica de Chile
Millenium Institute for Foundational Research on Data