Data Integration: Pizza Delivery Company

How many pizzas were delivered to Nuno’s home during his PhD?

From 2008 to 2012

1 / 24
Data Integration: Pizza Delivery Company

RDB

Clients / Orders

person   address
Nuno     Galway

How many pizzas were delivered to Nuno’s home during his PhD?

2012

From 2008 to 2012

1 / 24
Data Integration: Pizza Delivery Company

XML

Deliveries

RDB

Clients / Orders

<table>
<thead>
<tr>
<th>person</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno</td>
<td>Galway</td>
</tr>
</tbody>
</table>

<xml version="1.0"?>
<order>
  <customer>
    <name>Nuno</name>
    <address>Dublin</address>
  </customer>
</order>

How many pizzas were delivered to Nuno's home during his PhD?

From 2008 to 2012

1 / 24
Data Integration: Pizza Delivery Company

XML

```
<xml version="1.0"/>
<order>
  <name>Nuno</name>
  <address>Dublin</address>
</order>
```

RDB

<table>
<thead>
<tr>
<th>person</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno</td>
<td>Galway</td>
</tr>
</tbody>
</table>

How many pizzas were delivered to Nuno’s home during his PhD?

2012

From 2008 to 2012

1 / 24
Data Integration: Pizza Delivery Company

RDB

Clients / Orders

<table>
<thead>
<tr>
<th>person</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno</td>
<td>Galway</td>
</tr>
</tbody>
</table>

XML

<person>
  <name>Nuno</name>
  <address>Dublin</address>
</person>

RDF

How many pizzas were delivered to Nuno's home during his PhD?

2012

From 2008 to 2012

1 / 24
Data Integration: Pizza Delivery Company

XML

```xml
<person>
  <name>Nuno</name>
  <address>Dublin</address>
</person>
```

RDB

<table>
<thead>
<tr>
<th>person</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno</td>
<td>Galway</td>
</tr>
</tbody>
</table>

How many pizzas were delivered to Nuno’s home during his PhD?

2012

From 2008 to 2012

1 / 24
How many pizzas were delivered to Nuno’s home during his PhD?

From 2008 to 2012

1 / 24
Data Integration: Pizza Delivery Company

XML

```xml
<person>
  <name>Nuno</name>
  <address>Dublin</address>
</person>
```

RDF

How many pizzas were delivered to Nuno’s home during his PhD?

From 2008 to 2012

1 / 24
How many pizzas were delivered to Nuno’s home during his PhD?
Data Integration: Pizza Delivery Company

XML

```
<person>
  <name>Nuno</name>
  <address>Dublin</address>
</person>
```

RDF

How many pizzas were delivered to Nuno's home during his PhD?

From 2008 to 2012

1 / 24

RDB

<table>
<thead>
<tr>
<th>person</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno</td>
<td>Galway</td>
</tr>
</tbody>
</table>

Clients / Orders

<table>
<thead>
<tr>
<th>order to address date pizza</th>
<th>boy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pizza Nuno Galway 2008 Bob</td>
<td></td>
</tr>
<tr>
<td>Pizza Nuno Dublin 2008 Bart</td>
<td></td>
</tr>
<tr>
<td>Pizza Nuno Galway 2010 Charlie</td>
<td></td>
</tr>
<tr>
<td>Pizza Nuno Dublin 2012 Jack</td>
<td></td>
</tr>
</tbody>
</table>
Data Integration: Pizza Delivery Company

XML

```xml
<person>
  <name>Nuno</name>
  <address>Dublin</address>
</person>
```

RDF

How many pizzas were delivered to Nuno's home during his PhD?

From 2008 to 2012

<table>
<thead>
<tr>
<th>person</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno</td>
<td>Galway</td>
</tr>
</tbody>
</table>
Using Query Languages for Data Integration

- SQL: 
  ```sql
  select address from clients where person = "Nuno"
  ```

- SPARQL: 
  ```sparql
  select $lat $long from <geonames.org/..> where { $point geo:lat $lat; geo:long $long }
  ```

- XQuery: 
  ```xquery
  for $person in doc("eat.ie/..") return $person//name
  ```

**Mediator / Data Warehouse**

- How many pizzas were delivered to Nuno’s home during his PhD?
  - 2 / 24
Using Query Languages for Data Integration

**SQL**

```
select address from clients
where person = "Nuno"
```

**SPARQL**

```
select $lat $long
from <geonames.org/..>
where { $point geo:lat $lat;
geo:long $long }
```

**XQuery**

```
for $person in doc("eat.ie/..")
return $person//name
```
Using Query Languages for Data Integration

**SQL**

```sql
select address from clients
where person = "Nuno"
```

relation

**SPARQL**

```sparql
select $lat $long
from <geonames.org/..>
where { $point geo:lat $lat;
geo:long $long }
```

solution sequence

**XQuery**

```xml
for $person in doc("eat.ie/..")
return $person
```

sequence of items
Using Query Languages for Data Integration

SQL
select address from clients
where person = "Nuno"

SPARQL
select $lat $long
from <geonames.org/..>
where { $point geo:lat $lat;
geo:long $long } solution sequence

XQuery
for $person in doc("eat.ie/..")
return $person//name sequence of items

Mediator / Data Warehouse

How many pizzas were delivered to Nuno's home during his PhD?
2 / 24
Using Query Languages for Data Integration

SQL
select address from clients
where person = "Nuno"

SPARQL
select $lat $long
from <geonames.org/..>
where { $point geo:lat $lat;
     geo:long $long }

XQuery
for $person in doc("eat.ie/..")
return $person

Mediator / Data Warehouse

How many pizzas were delivered
to Nuno's home during his PhD?
2 / 24
Using Query Languages for Data Integration

SQL
select address from clients
where person = "Nuno"

SPARQL
select $lat $long
from <geonames.org/..>
where { $point geo:lat $lat;

XQuery
for $person in doc("eat.ie/..")
return $person//name

How many pizzas were delivered to Nuno’s home during his PhD?

2 / 24
How to represent context information in RDF?

- RDF triples

```turtle
:nuno :address :Galway .
:nuno :address :Dublin .
```
How to represent context information in RDF?

- **RDF triples**

  ```
  :nuno :address :Galway .
  :nuno :address :Dublin .
  ```

  Not enough!
How to represent context information in RDF?

- RDF triples

\[
\begin{align*}
&:nuno \text{ :address :Galway} . \\
&:nuno \text{ :address :Dublin} . \\
\end{align*}
\]

Not enough!

- Domain vocabulary/ontology

\[
\begin{align*}
&:address1 \text{ a :AddressRecord; } \\
&\quad :person \text{ :nuno; } \\
&\quad :address \text{ :Galway; } \\
&\quad :start \text{ "2008" ; } \\
&\quad :end \text{ "2012" . }
\end{align*}
\]
How to represent context information in RDF?

- **RDF triples**
  
  \[
  \text{:nuno :address :Galway .}
  \text{:nuno :address :Dublin .}
  \]
  
  Not enough!

- **Domain vocabulary/ontology**
  
  \[
  \text{:address1 a :AddressRecord;}
  \text{:person :nuno;}
  \text{:address :Galway;}
  \text{:start "2008" ;}
  \text{:end "2012" .}
  \]

- **Reification**
  
  \[
  \text{:address1 rdf:type rdf:Statement}
  \text{rdf:subject :nuno;}
  \text{rdf:predicate :address ;}
  \text{rdf:object :Dublin ;}
  \text{:loc <http://eat.ie/> .}
  \]
How to represent context information in RDF?

- **RDF triples**

  ```
  :nuno :address :Galway .
  :nuno :address :Dublin .
  ```

  *Not enough!*

- **Domain vocabulary/ontology**

  ```
  :address1 a :AddressRecord;
  :person :nuno;
  :address :Galway;
  :start "2008" ;
  :end "2012" .
  ```

- **Reification**

  ```
  :address1 rdf:type rdf:Statement
  rdf:subject :nuno;
  rdf:predicate :address ;
  rdf:object :Dublin ;
  ```

- **Named Graphs**
How to represent context information in RDF?

- **RDF triples**

```
:nuno :address :Galway .
nuno :address :Dublin .
```

Not enough!

- **Domain vocabulary/ontology**

```
:address1 a :AddressRecord;
  :person :nuno;
  :address :Galway;
  :start "2008" ;
  :end "2012" .
```

No defined semantics!

- **Reification**

```
:address1 rdf:type rdf:Statement
  rdf:subject :nuno;
  rdf:predicate :address ;
  rdf:object :Dublin ;
```

No defined semantics!

- **Named Graphs**
Hypothesis and Research Questions

Hypothesis

Efficient data integration over heterogeneous data sources can be achieved by

Research Questions

- How to design a query language that bridges the different formats?
- Can we reuse existing optimizations from other query languages?
- How to adapt RDF and SPARQL for context information?
Hypothesis

Efficient data integration over heterogenous data sources can be achieved by

- a combined query language that accesses heterogenous data in its original sources
Hypothesis and Research Questions

Hypothesis

Efficient data integration over heterogeneous data sources can be achieved by

- a combined query language that accesses heterogeneous data in its original sources
- optimisations for efficient query evaluation for this language

Research Questions

- How to design a query language that bridges the different formats?
- Can we reuse existing optimisations from other query languages?
- How to adapt RDF and SPARQL for context information?
Hypothesis and Research Questions

Hypothesis

Efficient data integration over heterogenous data sources can be achieved by

- a combined query language that accesses heterogenous data in its original sources
- optimisations for efficient query evaluation for this language
- an RDF-based format with support for context information

Research Questions

- How to design a query language that bridges the different formats?
- Can we reuse existing optimisations from other query languages?
- How to adapt RDF and SPARQL for context information?
Hypothesis and Research Questions

Hypothesis
Efficient data integration over heterogeneous data sources can be achieved by
- a combined query language that accesses heterogeneous data in its original sources
- optimisations for efficient query evaluation for this language
- an RDF-based format with support for context information

Research Questions
- How to design a query language that bridges the different formats?
Hypothesis and Research Questions

Hypothesis

Efficient data integration over heterogeneous data sources can be achieved by

- a combined query language that accesses heterogeneous data in its original sources
- optimisations for efficient query evaluation for this language
- an RDF-based format with support for context information

Research Questions

- How to design a query language that bridges the different formats?
- Can we reuse existing optimisations from other query languages?
### Hypothesis

Efficient data integration over heterogeneous data sources can be achieved by

- a combined query language that accesses heterogeneous data in its original sources
- optimisations for efficient query evaluation for this language
- an RDF-based format with support for context information

### Research Questions

- How to design a query language that bridges the different formats?
- Can we reuse existing optimisations from other query languages?
- How to adapt RDF and SPARQL for context information?
Outline and Contributions

- **XSPARQL**
  - Syntax & Semantics
  - Implementation
  - Evaluation

- **Annotated RDF(S)**
  - Annotated RDF(S)
  - AnQL: Annotated SPARQL
  - Architecture
Outline and Contributions

- XSPARQL
  - Syntax & Semantics
  - Implementation
  - Evaluation

- Annotated RDF(S)
  - Annotated RDF(S)
  - AnQL: Annotated SPARQL
  - Architecture

[JoDS2012], [EPIA2011]
Outline and Contributions

- **XSPARQL**
  - Syntax & Semantics
  - Implementation
  - Evaluation

- **Annotated RDF(S)**
  - Annotated RDF(S)
  - AnQL: Annotated SPARQL
  - Architecture

[AAAI2010], [ISWC2010], [JWS2012]
**XSPARQL**

- Transformation language between RDB, XML, and RDF
Transformation language between RDB, XML, and RDF

Syntactic extension of XQuery
XSPARQL

- Transformation language between RDB, XML, and RDF
- Syntactic extension of XQuery
- Semantics based on XQuery’s semantics
XSPARQL

- Transformation language between RDB, XML, and RDF
- Syntactic extension of XQuery
- Semantics based on XQuery’s semantics

Why based on XQuery?
- Expressive language
- Use as scripting language
XSPARQL

- Transformation language between RDB, XML, and RDF
- Syntactic extension of XQuery
- Semantics based on XQuery’s semantics

Why based on XQuery?
- Expressive language
- Use as scripting language
- Arbitrary Nesting of expressions
Same Language for each Format

```sql
XSPARQL
for address as $address from clients
where person = "Nuno"
return $address

XSPARQL
for $lat $long from <geonames.org/..>
where { $point geo:lat $lat; geo:long $long }
return $lat

for $person in doc("eat.ie/..")
return $person//name
```
for var in Expr
let var := Expr
where Expr
order by Expr
return Expr

XQuery
for $person in doc("eat.ie/..")
return $person//name
Same Language for each Format

XSPARQL

for $person in doc("eat.ie/..")
return $person//name
for SelectSpec from RelationList where WhereSpecList return Expr

XSPARQL

for address as $address from clients where person = "Nuno" return $address
for varlist
from DatasetClause
where { pattern }
return Expr

XSPARQL
for $lat $long from <geonames.org/...>
where { $point geo:lat $lat;
        geo:long $long }
return $lat
Create RDF with XSPARQL

Convert online orders into RDF

```xml
prefix : <http://pizza-vocab.ie/>

for $person in doc("eat.ie/..")//name
construct { :meal a :FoodOrder; :author {$person} }
```
Creating RDF with XSPARQL

Convert online orders into RDF

```xquery
prefix : <http://pizza-vocab.ie/>

for $person in doc("eat.ie/..")//name
  construct { :meal a :FoodOrder; :author {$person} }
```

construct clause generates RDF
Creating RDF with XSPARQL

Convert online orders into RDF

prefix : <http://pizza-vocab.ie/>

for $person in doc("eat.ie/..")//name
construct { :meal a :FoodOrder; :author {$person} }
Creating RDF with XSPARQL

Convert online orders into RDF

```xml
(prefix: <http://pizza-vocab.ie/>)

for $person in doc("eat.ie/..")//name
construct { :meal a :FoodOrder; :author {$person} }
```

Query result

```xml
@prefix : <http://pizza-vocab.ie/> .

:meal a :FoodOrder .
:meal :author "Nuno" .
```
Integration Query Example

“Display pizza deliveries in Google Maps using KML”

for $person in doc("eat.ie/...")//name
  for address as $address from clients
  where person = $person
    let $uri := fn:concat(
      "api.geonames.org/search?q=", $address)
    for $lat $long from $uri
    where { $point geo:lat $lat; geo:long $long }
    return <kml>
      <lat>{$lat}</lat>
      <long>{$long}</long>
    </kml>
Integration Query Example

“Display pizza deliveries in Google Maps using KML”

```xml
for $person in doc("eat.ie/...")//name
    for address as $address from clients
        where person = $person
        let $uri := fn:concat(
            "api.geonames.org/search?q=", $address)
        for $lat $long from $uri
            where { $point geo:lat $lat; geo:long $long }
        return <kml>
            <lat>{$lat}</lat>
            <long>{$long}</long>
        </kml>
```
“Display pizza deliveries in Google Maps using KML”

```xml
for $person in doc("eat.ie/...")//name
    for address as $address from clients
        where person = $person
            let $uri := fn:concat(
                "api.geonames.org/search?q=", $address)
            for $lat $long from $uri
                where { $point geo:lat $lat; geo:long $long }
            return <kml>
                <lat>{$lat}</lat>
                <long>{$long}</long>
            </kml>
```

Integration Query Example

“Display pizza deliveries in Google Maps using KML”

```xml
for $person in doc("eat.ie/...")//name
    for address as $address from clients
        where person = $person
            let $uri := fn:concat(
                "api.geonames.org/search?q=", $address)
                for $lat $long from $uri
                    where { $point geo:lat $lat; geo:long $long }
                return <kml>
                <lat>{$lat}<lat>
                <long>{$long}</long>
            </kml>
```

More involved XSPARQL queries: RDB2RDF
Direct Mapping:
∼ 130 LOC
R2RML:
∼ 290 LOC
Integration Query Example

“Display pizza deliveries in Google Maps using KML”

for $person in doc("eat.ie/...")//name
  for address as $address from clients
    where person = $person
    let $uri := fn:concat(
      "api.geonames.org/search?q=", $address)
    for $lat $long from $uri
      where { $point geo:lat $lat; geo:long $long }
  return <kml>
    <lat>{$lat}<lat>
    <long>{$long}</long>
  </kml>
Integration Query Example

“Display pizza deliveries in Google Maps using KML”

for $person in doc("eat.ie/...")//name
    for address as $address from clients
        where person = $person
        let $uri := fn:concat(
            "api.geonames.org/search?q=", $address)
        for $lat $long from $uri
            where { $point geo:lat $lat; geo:long $long }
        return <kml>
            <lat>{$lat}</lat>
            <long>{$long}</long>
        </kml>

More involved XSPARQL queries: RDB2RDF

- Direct Mapping: ~130 LOC
- R2RML: ~290 LOC
Extension of the XQuery Evaluation Semantics

for $person in doc("eat.ie/...")//name
  for address as $address from clients
  where person = $person
    let $uri := fn:concat(
      "api.geonames.org/search?q=", $address)
  for $lat $long from $uri
  where { $point geo:lat $lat; geo:long $long }
return <kml>
  <lat>{$lat}</lat>
  <long>{$long}</long>
</kml>
Language Semantics

- Extension of the XQuery Evaluation Semantics
- for add variables and values to the dynamic environment

```xml
for $person in doc("eat.ie/...")//name
for address as $address from clients
where person = $person
  let $uri := fn:concat(
    "api.geonames.org/search?q=", $address)
  for $lat $long from $uri
where { $point geo:lat $lat; geo:long $long }
return <kml>
  <lat>{$lat}</lat>
  <long>{$long}</long>
</kml>
```
Language Semantics

- Extension of the XQuery Evaluation Semantics
- For add variables and values to the dynamic environment
- Reuse semantics of original languages for the new expressions

```xml
for $person in doc("eat.ie/...")//name
  for address as $address from clients
    where person = $person
    let $uri := fn:concat(
      "api.geonames.org/search?q=", $address)
    for $lat $long from $uri
    where { $point geo:lat $lat; geo:long $long }
    return <kml>
      <lat>{$lat}</lat>
      <long>{$long}</long>
    </kml>
```
Language Semantics

- Extension of the XQuery Evaluation Semantics
- For add variables and values to the dynamic environment
- Reuse semantics of original languages for the new expressions

```xml
for $person in doc("eat.ie/...")//name
  for address as $address from clients
    where person = $person
    let $uri := fn:concat("api.geonames.org/search?q=", $address)
    for $lat $long from $uri
      where { $point geo:lat $lat; geo:long $long }
    return <kml>
      <lat>{$lat}<lat>
      <long>{$long}</long>
    </kml>
```

dynEnv.varValue ⇒ relation

```
person
  "nuno"
```

```
dynEnv + globalPosition
  =
  ((Pos 1, ..., Pos j))
```

```
dynEnv ⊢ fs : dataset
```

```
dynEnv ⊢ fs : sparql
```

```
dynEnv + globalPosition + activeDataset
```

```
dynEnv + varValue
```

```
for $Var 1 ··· $Var n DatasetClause WhereClause SolutionModifier
  return ExprSingle
```

```
dynEnv ⊢ fs : value
```

```
9 / 24
```
Language Semantics

- Extension of the XQuery Evaluation Semantics
- for add variables and values to the dynamic environment
- Reuse semantics of original languages for the new expressions

```xml
for $person in doc("eat.ie/...")//name
for address as $address from clients
where person = $person
let $uri := fn:concat(
    "api.geonames.org/search?q=", $address)
for $lat $long from $uri
where { $point geo:lat $lat; geo:long $long }
return <kml>
    <lat>{$lat}</lat>
    <long>{$long}</long>
</kml>
```
Language Semantics

- Extension of the XQuery Evaluation Semantics
- For add variables and values to the dynamic environment
- Reuse semantics of original languages for the new expressions

```
for $person in doc("eat.ie/...")//name
    for address as $address from clients
        where person = $person
        let $uri := fn:concat(
            "api.geonames.org/search?q=", $address)
        for $lat $long from $uri
            where { $point geo:lat $lat; geo:long $long }
        return <kml>
            <lat>{$lat}</lat>
            <long>{$long}</long>
        </kml>
```

dynEnv.varValue ⇒ relation
person "nuno"

eval(SQL SELECT) ⋊ ⋉ 
dynEnv.varValue

```
{ { person ⇒ "nuno" } }
```

dynEnv ⊢ fs : dataset (DatasetClause)
dynEnv ⊢ fs : sparql (Dataset, WhereClause, SolutionModifier)
dynEnv + globalPosition ((Pos1, ···, Posj)) + activeDataset (Dataset) + varValue

```
for $Var1 ··· $Varn DatasetClause WhereClause SolutionModifier
return ExprSingle ⇒ Value1, ..., Valm
```

dynEnv + globalPosition ((Pos1, ···, Posj, m)) + activeDataset (Dataset) + varValue

```
{ { person ⇒ "nuno" } }
```
Extension of the XQuery Evaluation Semantics
- for add variables and values to the dynamic environment
- Reuse semantics of original languages for the new expressions

```xml
for $person in doc("eat.ie/...")//name
  for address as $address from clients
  where person = $person
  let $uri := fn:concat(
    "api.geonames.org/search?q=", $address
  )
  for $lat $long from $uri
  where { $point geo:lat $lat; geo:long $long }
return <kml>
  <lat>{$lat}</lat>
  <long>{$long}</long>
</kml>
```
Language Semantics

- Extension of the XQuery Evaluation Semantics
- For add variables and values to the dynamic environment
- Reuse semantics of original languages for the new expressions

```plaintext
dynEnv.globalPosition = (Pos₁, ⋯, Posₗ)
dynEnv ⊢ fs:dataset(DatasetClause) ⇒ Dataset

dynEnv ⊢ fs:sparql(Dataset, WhereClause, SolutionModifier) ⇒ µ₁, ⋯, µₘ

dynEnv + globalPosition((Pos₁, ⋯, Posₗ, 1)) + activeDataset(Dataset)

+ varValue(Var₁ ⇒ fs:value(µ₁, Var₁); ⋯)
+ varValue(Varₙ ⇒ fs:value(µ₁, Varₙ))

⇒ ExprSingle ⇒ Value₁

⋮

dynEnv + globalPosition((Pos₁, ⋯, Posₗ, m)) + activeDataset(Dataset)

+ varValue(Var₁ ⇒ fs:value(µₘ, Var₁); ⋯)
+ varValue(Varₙ ⇒ fs:value(µₘ, Varₙ))

⇒ ExprSingle ⇒ Valueₘ

for $Var₁ · · · $Varₙ DatasetClause
WhereClause SolutionModifier ⇒ Value₁, ⋯, Valueₘ

return ExprSingle
```

9 / 24
Extension of the XQuery Evaluation Semantics

- For add variables and values to the dynamic environment
- Reuse semantics of original languages for the new expressions

\[
\begin{align*}
\text{dynEnv.globalPosition} &= (Pos_1, \ldots, Pos_j) \\
\text{dynEnv} &\vdash \text{fs:dataset}(\text{DatasetClause}) \Rightarrow \text{Dataset} \\
\text{dynEnv} &\vdash \text{fs:sparql}(\text{Dataset, WhereClause, SolutionModifier}) \Rightarrow \mu_1, \ldots, \mu_m \\
\text{dynEnv} \oplus \text{globalPosition}((Pos_1, \ldots, Pos_j, 1)) &\oplus \text{activeDataset}(\text{Dataset}) \\
&+ \text{varValue}\left(\begin{array}{c}
\text{Var}_1 \Rightarrow \text{fs:value}(\mu_1, \text{Var}_1) \\
\vdots \\
\text{Var}_n \Rightarrow \text{fs:value}(\mu_1, \text{Var}_n)
\end{array}\right) \vdash \text{ExprSingle} \Rightarrow \text{Value}_1 \\
\vdots \\
\text{dynEnv} \oplus \text{globalPosition}((Pos_1, \ldots, Pos_j, m)) &\oplus \text{activeDataset}(\text{Dataset}) \\
&+ \text{varValue}\left(\begin{array}{c}
\text{Var}_1 \Rightarrow \text{fs:value}(\mu_m, \text{Var}_1) \\
\vdots \\
\text{Var}_n \Rightarrow \text{fs:value}(\mu_m, \text{Var}_n)
\end{array}\right) \vdash \text{ExprSingle} \Rightarrow \text{Value}_m \\
\text{for} \ \$\text{Var}_1 \ldots \$\text{Var}_n \ \text{DatasetClause} \\
\text{WhereClause} \ \text{SolutionModifier} &\Rightarrow \text{Value}_1, \ldots, \text{Value}_m \\
\text{return} \ \text{ExprSingle}
\end{align*}
\]
Language Implementation

- Reuse components (SPARQL engine, Relational Database)
Language Implementation

- Reuse components (SPARQL engine, Relational Database)
- Implemented XSPARQL by rewriting to XQuery
Language Implementation

- Reuse components (SPARQL engine, Relational Database)
- Implemented XSPARQL by rewriting to XQuery
- Semantics implemented by substitution of bound variables
Language Implementation

- Reuse components (SPARQL engine, Relational Database)
- Implemented XSPARQL by rewriting to XQuery
- Semantics implemented by substitution of bound variables

Bound Variable Substitution

- *Bound* variables are replaced by their *value* at runtime

```xml
for $person in doc("eat.ie/...")//name
 for address as $address from clients
 where person = $person
```
Language Implementation

- Reuse components (SPARQL engine, Relational Database)
- Implemented XSPARQL by rewriting to XQuery
- Semantics implemented by substitution of bound variables

Bound Variable Substitution

- **Bound** variables are replaced by their **value** at runtime
- Implemented in the generated XQuery

```xml
for $person in doc("eat.ie/...")//name
  for address as $address from clients
    where person = $person
  fn:concat(
    "SELECT address from clients
    where person = ", $person))
```
Language Implementation

- Reuse components (SPARQL engine, Relational Database)
- Implemented XSPARQL by rewriting to XQuery
- Semantics implemented by substitution of bound variables

Bound Variable Substitution

- **Bound** variables are replaced by their **value** at runtime
- Implemented in the generated XQuery
- Pushing the variable bindings into the respective query engine

```xml
for $person in doc("eat.ie/...")//name
  for address as $address from clients
    where person = $person
      fn:concat("SELECT address from clients
        where person = ", $person)
```
Implementation

The diagram shows the flow of a query rewritten using the Query Rewriter, transformed into an XQuery query, and then executed by the Enhanced XQuery engine. Custom XQuery functions implemented as Java methods are also part of this process.
Implementation

XSPARQL query

Query Rewriter

XSPARQL

XQuery query

Enhanced XQuery engine
Implementation

Custom XQuery functions implemented as Java methods

11 / 24
Implementation

XSPARQL query

Query Rewriter

XSPARQL

XQuery query

Enhanced XQuery engine

XML / RDF

RDB

XML

RDF

SQL

XSPARQL

SPARQL

Custom XQuery functions implemented as Java methods.
Experimental Results

Compared to native XQuery (XMark)

RDB and RDF: same order of magnitude for most queries

Except on RDF nested queries (self-joining data), several orders of magnitude slower due to the number of calls to the SPARQL engine.

XMark
XSPARQL Evaluation

Comparing XMark and XMark\_RDF:
- RDB and RDF: same order of magnitude for most queries
- RDF nested queries (self joining data): several orders of magnitude slower due to the number of calls to the SPARQL engine.
XSPARQL Evaluation

Compared to native XQuery (XMark)
RDB and RDF: same order of magnitude for most queries
Except on RDF nested queries (self joining data)
due to the number of calls to the SPARQL engine
XSPARQL Evaluation

Experimental Results

- Compared to native XQuery (XMark)
  - RDB and RDF: same order of magnitude for most queries

Except on RDF nested queries (self joining data), several orders of magnitude slower due to the number of calls to the SPARQL engine.
Experimental Results

- Compared to native XQuery (XMark)
  - **RDB and RDF**: same order of magnitude for most queries
  - Except on RDF nested queries (self joining data)
    - several orders of magnitude slower
    - due to the number of calls to the SPARQL engine
Q₈: “List the names of persons and the number of items they bought.”

```
for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}"/>
  {count(
    for * from <input.rdf>
    where { $ca :buyer $person .}
    return $ca
  )}
</item>
```
Q₈: “List the names of persons and the number of items they bought.”

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}">
   {count(
      for * from <input.rdf>
      where { $ca :buyer $person .}
      return $ca
   )}
</item>

Returns ( nuno, 
         axel, 
         ... )
Rewriting techniques for Nested Queries

Q₈: “List the names of persons and the number of items they bought.”

```xml
for $person $name from <input.rdf>
  where { $person foaf:name $name }
return <item person="$name">
  {count(
    for * from <input.rdf>
    where { $ca :buyer $person .}
    return $ca
  )}
</item>
```

Returns

```
nuno,
axel,
```

sparqlQuery(
  fn:concat(
    "SELECT * from <input.rdf>
    where { $ca :buyer ", $person, " .}"))
)
Rewriting techniques for Nested Queries

Q8: “List the names of persons and the number of items they bought.”

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}">
  {count(
    for * from <input.rdf>
    where { $ca :buyer $person .}
    return $ca
  )}
</item>

`sparqlQuery(`
  fn:concat(`
    "SELECT * from <input.rdf>
Rewriting techniques for Nested Queries

Q₈: “List the names of persons and the number of items they bought.”

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}"/>
{count(
  for * from <input.rdf>
  where { $ca :buyer $person .}
  return $ca
)}
</item>

Returns

(sp) nuno, axel, ...

sparqlQuery(
  fn:concat(
    "SELECT * from <input.rdf>
    where { $ca :buyer ", axel
    ", ".}"
  )
)

Join in XQuery

Nested Loop rewriting

Applied to related language: SPARQL2XQuery [SAC2008]

Other optimisations:

Join in SPARQL
Q₈: “List the names of persons and the number of items they bought.”

```xml
let $aux := sparqlQuery(
  fn:concat(
    "SELECT * from <input.rdf>
    where { $ca :buyer $person .}" )
)

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}">
  {count(
    for * from <input.rdf>
    where { $ca :buyer $person .}
    return $ca
  )}
</item>
```
Rewriting techniques for Nested Queries

Q8: “List the names of persons and the number of items they bought.”

```xml
let $aux := sparqlQuery( 
    fn:concat( 
        "SELECT * from <input.rdf>
        where { $ca :buyer $person .}" )
)
for $person $name from <input.rdf> 
where { $person foaf:name $name }
return <item person="{$name}">
  {count( 
    return $ca 
  )}
</item>
```

Join in XQuery

```xml
return $ca
```
Rewriting techniques for Nested Queries

Q8: “List the names of persons and the number of items they bought.”

let $aux := sparqlQuery(
    fn:concat(
        "SELECT * from <input.rdf>
        where { $ca :buyer $person .}"
    )
)

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}">
  {count(
    for $ca from $aux
    where { $ca :buyer $person .}
    return $ca
  )}
</item>

Join in XQuery

let $aux := sparqlQuery(
    fn:concat(
        "SELECT * from <input.rdf>
        where { $ca :buyer $person .}"
    )
)

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}">
  {count(
    return $ca
  )}
</item>

Nested Loop rewriting

Applied to related language: SPARQL2XQuery [SAC2008]
Rewriting techniques for Nested Queries

Q8: “List the names of persons and the number of items they bought.”

let $aux := sparqlQuery(
  fn:concat(
  "SELECT * from <input.rdf>
  where { $ca :buyer $person .}" )
)

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}">
  {count(

  Join in XQuery
  return $ca
  )}
</item>

Applied to related language: SPARQL2XQuery [SAC2008]
Rewriting techniques for Nested Queries

Q₈: “List the names of persons and the number of items they bought.”

let $aux := sparqlQuery(
    fn:concat(
        "SELECT * from <input.rdf>
        where { $ca :buyer $person .}" )
)

for $person $name from <input.rdf>
where { $person foaf:name $name }
return <item person="{$name}">
    {count(
        Join in XQuery
        return $ca
    )}
</item>

Other optimisations:
Join in SPARQL

Applied to related language:
SPARQL2XQuery [SAC2008]
Evaluation of different rewritings - $XMark_{RDF} Q_8$

Optimised rewritings show promising results

Best Results: SPARQL-based

Results included in [JoDS2012]
Evaluation of different rewritings - $XMark_{RDB}Q_8$

- Not so effective for RDB, requires different optimisations
- Due to schema representation?
- Additional Results for this thesis
Data Integration: Pizza Delivery Company

XML

```
<person>
  <name>Nuno</name>
  <address>Dublin</address>
</person>
```

2012

From 2008 to 2012

<table>
<thead>
<tr>
<th>person</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno</td>
<td>Galway</td>
</tr>
</tbody>
</table>
Annotations refer to a specific domain
Use Annotated RDF(S)!

Annotations refer to a specific domain

Temporal:

Annotations refer to a specific **domain**

**Temporal:**


**Fuzzy:**

`:nuno :address :Dublin . 0.9`
Use Annotated RDF(S)!

Annotations refer to a specific **domain**

**Temporal:**


**Fuzzy:**

:nuno :address :Dublin . 0.9

**Provenance:**

Use Annotated RDF(S)!

Annotations refer to a specific domain

Temporal:


Fuzzy:

:nuno :address :Dublin . 0.9

Provenance:


Representation for the values of each annotation domain
Annotated RDF(S) Inference example

Inference rules are **independent** of the annotation domain
Annotated RDF(S) Inference example

Inference rules are **independent** of the annotation domain

**RDFS subPropertyOf ("sp") rule:**

```
```

```
:address sp foaf:based_near .
:nuno :address :Galway .
⇒ :nuno foaf:based_near :Galway .
```
Annotated RDF(S) Inference example

Inference rules are **independent** of the annotation domain

<table>
<thead>
<tr>
<th>Annotated RDFS subPropertyOf (“sp”) rule:</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Prop1 sp ?Prop2 . ?v1</td>
</tr>
<tr>
<td>?x ?Prop1 ?y . ?v2</td>
</tr>
</tbody>
</table>

```
:address sp foaf:based_near .          [2009, +∞]  
⇒ :nuno foaf:based_near :Galway .      
```
Inference rules are **independent** of the annotation domain

### Annotated RDFS subPropertyOf ("sp") rule:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Prop1 sp ?Prop2 . ?v1</td>
<td></td>
</tr>
<tr>
<td>?x ?Prop1 ?y . ?v2</td>
<td></td>
</tr>
</tbody>
</table>

### Example:

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>:address</td>
<td>sp foaf:based_near</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:nuno :address</td>
<td>:Galway</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Annotated RDF(S) Inference example

Inference rules are independent of the annotation domain

**Annotated RDFS subPropertyOf (“sp”) rule:**

\[
\text{?Prop1} \text{ sp } \text{?Prop2} . \ ?v1 \\
\text{?x } \text{?Prop1 } \text{?y} . \ ?v2 \\
\Rightarrow \text{?x } \text{?Prop2} \ ?y . \ ?v1 \otimes ?v2
\]

\[
\text{:address sp foaf:based\_near} . \ [2009,\infty] \\
\text{:nuno :address :Galway} . \ [2008,2012] \\
\Rightarrow \text{:nuno foaf:based\_near :Galway} . \ [2009,\infty] \otimes [2008,2012]
\]

**Extra rule to group annotations triples (⊕):**

\[
\text{:nuno :address :Galway} . \ [2009,\infty] \\
\text{:nuno :address :Galway} . \ [2008,2012]
\]
Annotated RDF(S) Inference example

Inference rules are **independent** of the annotation domain

**Annotated RDFS subPropertyOf (“sp”) rule:**

\[
?Prop1 \text{ sp } ?Prop2 . \ ?v1 \\
?x \ ?Prop1 \ ?y . \ ?v2 \\
\Rightarrow \ ?x \ ?Prop2 \ ?y . \ ?v1 \odot ?v2
\]

\[
:address \text{ sp foaf:based\_near } . \ [2009, +\infty] \\
\Rightarrow \ :nuno \foaf:based\_near :Galway . \ [2009, +\infty] \odot [2008, 2012]
\]

**Extra rule to group annotations triples (⊕):**

\[
:nuno :address :Galway . \ [2009, +\infty] \\
\Rightarrow \ :nuno :address :Galway . \ [2009, +\infty] \oplus [2008, 2012]
\]
Inference rules are independent of the annotation domain

Annotated RDFS subPropertyOf ("sp") rule:

\[
?\text{Prop1} \text{ sp } ?\text{Prop2} . \ ?v1 \\
?x \ ?\text{Prop1} \ ?y . \ ?v2 \\
\Rightarrow ?x \ ?\text{Prop2} \ ?y . \ ?v1 \otimes ?v2
\]

:address sp foaf:based_near . \ [2009, +\infty] \\

Extra rule to group annotations triples (\(\oplus\)):

\[
\Rightarrow :nuno :address :Galway . \ [2009, +\infty] \oplus [2008,2012]
\]
Operations on Annotations Elements

\[ [2009, +\infty] \quad [2008, 2012] \]
Operations on Annotations Elements

\[ [2009, +\infty] \quad \otimes \quad [2008, 2012] \]

\[ [2009, 2012] \]

\[ [2008, +\infty] \]
Operations on Annotations Elements

\[
\begin{align*}
[2009, +\infty] \quad &\quad \otimes \\ &\quad [2008, 2012] \\
\otimes &\quad [2009, 2012] \\
&\quad [] \\
\end{align*}
\]
Operations on Annotations Elements

\[
\begin{array}{c}
[2008, +\infty] \\
[2009, +\infty] \\
[2009, 2012] \\
\emptyset
\end{array}
\]
Operations on Annotations Elements

\[-\infty, +\infty\]

\[2008, +\infty\]

\[2009, +\infty\]

\[2008, 2012\]

\[2009, 2012\]

\[]
Operations on Annotations Elements

Operations on Annotations Elements

Operations on Annotations Elements


\[ [2000, 2005] \]

\[ [2008, 2012] \]

\[ [2000, 2005] \]

\[ [2008, 2012] \]

\[ [] \]
Operations on Annotations Elements


\[\{ [] \} \]

\[\oplus\]

\[\otimes\]
AnQL: Annotated SPARQL

Extension of SPARQL Syntax

- triple pattern

Example

?person a foaf:Person .
AnQL: Annotated SPARQL

Extension of SPARQL Syntax

- triple pattern
- annotated triple pattern is a triple pattern plus
  - annotation term

Example

?person a foaf:Person . [−∞, +∞]
Extension of SPARQL Syntax

- triple pattern
- annotated triple pattern is a triple pattern plus
  - annotation term; or
  - annotation variable

Example

```
?person :address ?address . ?l
```
AnQL: Annotated SPARQL

Extension of SPARQL Syntax

- triple pattern
- **annotated triple pattern** is a triple pattern plus
  - annotation term; or
  - annotation variable
- **Basic Annotated Patterns (BAP)** are sets of annotated triple patterns

Example

```sparql
{ ?person a foaf:Person . [−∞, +∞]
  ?person :address ?address . ?l
}
```
AnQL: Annotated SPARQL

Extension of SPARQL Syntax

- triple pattern
- annotated triple pattern is a triple pattern plus
  - annotation term; or
  - annotation variable
- Basic Annotated Patterns (BAP) are sets of annotated triple patterns

Example

```sparql
{  ?person a foaf:Person . [−∞, +∞]
   ?person :address ?address . ?l
}
```

Combine BAPs using AND(.), OPTIONAL, UNION, FILTER
“List my address, time interval and optionally people living in the same city at the same time.”

```
SELECT ?city ?t ?person
WHERE { :nuno :address ?city . ?t
    OPTIONAL { ?person :address ?city . ?t } }
```

Sample input:

```
```
“List my address, time interval and optionally people living in the same city at the same time.”

```
SELECT ?city ?t ?person
WHERE { :nuno :address ?city . ?t
   OPTIONAL { ?person :address ?city . ?t }
}
```

Sample input:

```
```

Answers:

\[ S_1 = \{ ?city \rightarrow :\text{Galway}, ?t \rightarrow [2008, 2012]\} \]
Evaluation of SPARQL OPTIONALs

“List my address, time interval and optionally people living in the same city at the same time.”

```
SELECT ?city ?t ?person
WHERE { :nuno :address ?city . ?t
    OPTIONAL { ?person :address ?city . ?t } }
```

Sample input:

```
```

Answers:

```
```
"List my address, time interval and optionally people living in the same city at the same time."

```
SELECT   ?city ?t ?person
WHERE    { :nuno :address ?city . ?t
            OPTIONAL { ?person :address ?city . ?t } }
```

**Sample input:**


**Answers:**

$$\begin{align*}
S_1 &= \{ ?city \rightarrow :Galway, ?t \rightarrow [2008, 2012] \} \\
S_2 &= \{ ?city \rightarrow :Galway, ?t \rightarrow [2008, 2010], ?person \rightarrow :axel \} 
\end{align*}$$

**OPTIONAL** provide more information maybe restricting annotation values
Annotated RDF(S) Domains

Temporal:

Fuzzy:
:nuno :address :Galway . 0.9

Provenance:
Annotated RDF(S) Domains

Temporal:

Fuzzy:
:nuno :address :Galway .  0.9

Provenance:

Combining domains: Included in [JWS2012]
Annotated RDF(S) Domains

Temporal:

Fuzzy:
:nuno :address :Galway . 0.9

Provenance:

Combining domains: Included in [JWS2012]

Access Control: Presented at [ICLP2012]
:nuno :address :Galway . [nl]
Architecture combining XSPARQL and AnQL

Annotated RDF

Data converted to RDF using XSPARQL

XSPARQL
Architecture combining XSPARQL and AnQL

- **Annotated RDFS**
  - AnQL
    - Reasoner
  - Annotated RDF

- Implementation in SWI Prolog

Data converted to RDF using XSPARQL

Each domain is a different Prolog module

Rules specified as Prolog predicates
Architecture combining XSPARQL and AnQL

Each domain is a different Prolog module

**Domains**
- Temporal
- Fuzzy
- Provenance
- Access Control

**Annotated RDFS**

- AnQL
- Reasoner

**Annotated RDF**

**XSPARQL**

Data converted to RDF using XSPARQL

Implementation in SWI Prolog

Each domain is a different Prolog module

Rules specified as Prolog predicates

RDFS (\(\rho_{\text{df}}\))

Rules AnQL Annotated RDFS

Annotated RDFS
Architecture combining XSPARQL and AnQL

- Domains
  - Temporal
  - Fuzzy
  - Provenance
  - Access Control

- Annotated RDFS
  - AnQL
  - Reasoner
  - Annotated RDF

- Rules
  - RDFS ($\rho$df)
  - Custom Rules

Data converted to RDF using XSPARQL

Implementation in SWI Prolog

Each domain is a different Prolog module

Rules specified as Prolog predicates
Architecture combining XSPARQL and AnQL

- AnQL Query
  - Annotated RDFS
    - Annotated RDF
      - XSPARQL
          - Data converted to RDF using XSPARQL
          - Implementation in SWI Prolog

- Domains:
  - Temporal
  - Fuzzy
  - Provenance
  - Access Control

- AnQL
  - Reasoner
    - Annotated RDFS
      - Rules
        - RDFS (ρdf)
        - Custom Rules
Efficient data integration over heterogeneous data sources can be achieved by

1. Using a combined query language that accesses heterogeneous data in its original sources
2. Implementing optimizations for efficient query evaluation for this language
3. Using rewriting techniques for nested queries for our implementation of XSPARQL
4. Developing an RDF-based format with support for context information
   - Annotated RDFS: inferences and query over context information

Use XSPARQL to create Annotated RDF representing the integrated data.
Conclusions

Efficient data integration over heterogeneous data sources can be achieved by

1. a combined query language that accesses heterogeneous data in its original sources

2. optimisations for efficient query evaluation for this language

3. an RDF-based format with support for context information
Efficient data integration over heterogeneous data sources can be achieved by

1. a combined query language that accesses heterogeneous data in its original sources
   - XSPARQL can integrate heterogeneous sources

2. optimisations for efficient query evaluation for this language

3. an RDF-based format with support for context information
Conclusions

Efficient data integration over heterogenous data sources can be achieved by

1. a combined query language that accesses heterogenous data in its original sources
   - XSPARQL can integrate heterogeneous sources

2. optimisations for efficient query evaluation for this language
   - rewriting techniques for nested queries for our implementation of XSPARQL

3. an RDF-based format with support for context information
Conclusions

Efficient data integration over heterogenous data sources can be achieved by

1. a combined query language that accesses heterogenous data in its original sources
   - XSPARQL can integrate heterogeneous sources

2. optimisations for efficient query evaluation for this language
   - rewriting techniques for nested queries for our implementation of XSPARQL

3. an RDF-based format with support for context information
   - Annotated RDFS: inferences and query over context information
   - Use XSPARQL to create Annotated RDF representing the integrated data
Conclusions

Efficient data integration over heterogenous data sources can be achieved by

1. a combined query language that accesses heterogenous data in its original sources
   - XSPARQL can integrate heterogeneous sources

2. optimisations for efficient query evaluation for this language
   - rewriting techniques for nested queries for our implementation of XSPARQL

3. an RDF-based format with support for context information
   - Annotated RDFS: inferences and query over context information
   - Use XSPARQL to create Annotated RDF representing the integrated data

Thank you! Questions?


Nuno Lopes, Axel Polleres, Umberto Straccia, and Antoine Zimmermann.

AnQL: SPARQLing Up Annotated RDFS.

Umberto Straccia, Nuno Lopes, Gergely Lukácsy, and Axel Polleres.

Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia.
A General Framework for Representing, Reasoning and Querying with Annotated Semantic Web Data.