Mining RDF
A SPARQL Tutorial

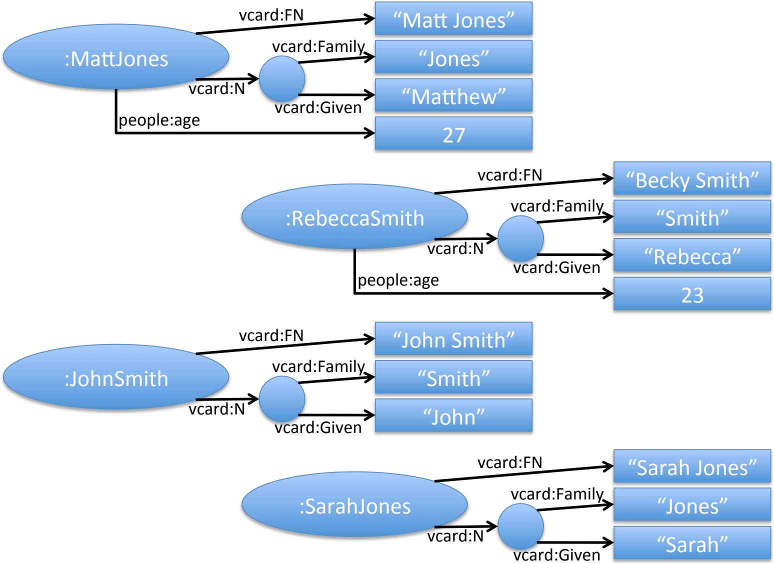
Ian Emmons
iemmons@bbn.com

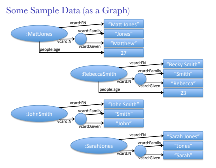
November 18, 2015

Raytheon
BBN Technologies

Outline

Some Sample Data (as a Graph)



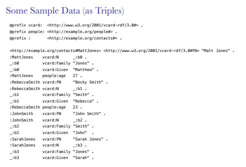


Some Sample Data (as a Graph)

1. Introduce the predicates we're using here.
2. Discuss the distinction between URIs and strings.
3. Blank nodes have identity, but only within this particular graph.
4. In other words, if we merge this graph with another, the blank nodes from the two graphs will never fold together.
5. Some nodes have ages, others don't. The "Open World Assumption" says that nodes without ages have unknown ages, i.e., we cannot assume they don't have ages. (Contrast with relational.)

Some Sample Data (as Triples)

```
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .

<http://example.org/contacts#MattJones> <http://www.w3.org/2001/vcard-rdf/3.0#FN> "Matt Jones" .
:MattJones    vcard:N      _:b0 .
_:b0          vcard:Family "Jones" .
_:b0          vcard:Given  "Matthew" .
:MattJones    people:age   27 .
:RebeccaSmith vcard:FN     "Becky Smith" .
:RebeccaSmith vcard:N      _:b1 .
_:b1          vcard:Family "Smith" .
_:b1          vcard:Given  "Rebecca" .
:RebeccaSmith people:age   23 .
:JohnSmith    vcard:FN     "John Smith" .
:JohnSmith    vcard:N      _:b2 .
_:b2          vcard:Family "Smith" .
_:b2          vcard:Given  "John"  .
:SarahJones   vcard:FN     "Sarah Jones" .
:SarahJones   vcard:N      _:b3 .
_:b3          vcard:Family "Jones" .
_:b3          vcard:Given  "Sarah" .
```



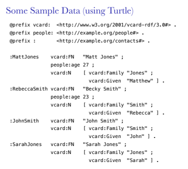
Some Sample Data (as Triples)

```
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .

<http://example.org/contacts#MattJones> <http://www.w3.org/2001/vcard-rdf/3.0#FN> "Matt Jones" .
:MattJones    vcard:N      _:b0 .
_:b0          vcard:Family "Jones" .
_:b0          vcard:Given  "Matthew" .
:MattJones    people:age   27 .
:RebeccaSmith vcard:FN     "Becky Smith" .
:RebeccaSmith vcard:N      _:b1 .
_:b1          vcard:Family "Smith" .
_:b1          vcard:Given  "Rebecca" .
:RebeccaSmith people:age   23 .
:JohnSmith    vcard:FN     "John Smith" .
:JohnSmith    vcard:N      _:b2 .
_:b2          vcard:Family "Smith" .
_:b2          vcard:Given  "John"  .
:SarahJones   vcard:FN     "Sarah Jones" .
:SarahJones   vcard:N      _:b3 .
_:b3          vcard:Family "Jones" .
_:b3          vcard:Given  "Sarah" .
```

1. Discuss prefixes and how we form URIs with them. Also note the abscense of angle brackets when we use them.
2. Special prefix for blank nodes
3. URIs versus strings
4. Blank node labels have meaning only within this graph.
5. Note the periods to terminate statements.

Some Sample Data (using Turtle)

```
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .

:MattJones     vcard:FN   "Matt Jones" ;
               people:age 27 ;
               vcard:N    [ vcard:Family "Jones" ;
                            vcard:Given  "Matthew" ] .
:RebeccaSmith vcard:FN   "Becky Smith" ;
               people:age 23 ;
               vcard:N    [ vcard:Family "Smith" ;
                            vcard:Given  "Rebecca" ] .
:JohnSmith     vcard:FN   "John Smith" ;
               vcard:N    [ vcard:Family "Smith" ;
                            vcard:Given  "John" ] .
:SarahJones    vcard:FN   "Sarah Jones" ;
               vcard:N    [ vcard:Family "Jones" ;
                            vcard:Given  "Sarah" ] .
```



Some Sample Data (using Turtle)

```
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .

:MattJones    vcard:FN   "Matt Jones" ;
              people:age 27 ;
              vcard:N    [ vcard:Family "Jones" ;
                           vcard:Given  "Matthew" ] .
:RebeccaSmith vcard:FN   "Becky Smith" ;
              people:age 23 ;
              vcard:N    [ vcard:Family "Smith" ;
                           vcard:Given  "Rebecca" ] .
:JohnSmith    vcard:FN   "John Smith" ;
              vcard:N    [ vcard:Family "Smith" ;
                           vcard:Given  "John" ] .
:SarahJones   vcard:FN   "Sarah Jones" ;
              vcard:N    [ vcard:Family "Jones" ;
                           vcard:Given  "Sarah" ] .
```

Turtle conveniences:

1. Semi-colons versus periods
2. Blank nodes

Other Turtle-isms

```
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .

:MattJones    a people:Person ;
              rdf:type people:Person ;
              vcard:FN "Matt Jones" ;
              vcard:FN "Matt Jones"^^xsd:string ;
              vcard:FN "Matt Jones"@en ;
              vcard:NICKNAME "Matt", "Jonesy", "M", "Человек-паук"@ru ;
              people:age 27 ;
              people:height 6.25 ;
              people:isMale true .
```

Other Turtle-isms

```
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .

:MattJones    a people:Person ;
              rdf:type people:Person ;
              vcard:FN "Matt Jones" ;
              vcard:FN "Matt Jones"^^xsd:string ;
              vcard:FN "Matt Jones"@en ;
              vcard:NICKNAME "Matt", "Jonesy", "M", "Человек-паук"@ru ;
              people:age 27 ;
              people:height 6.25 ;
              people:isMale true .
```

More Turtle conveniences:

1. “a” versus rdf:type
2. Type-less versus typed strings
3. Language-tagged strings (rdf:langString)
4. Comma-separated lists
5. UTF-8 character encoding
6. Shortcuts for xsd:integer, xsd:decimal, and xsd:boolean
7. Note that xsd:decimal is preferred to xsd:double and xsd:float

Some Sample Data (using Turtle)

```
@prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .

:MattJones    vcard:FN   "Matt Jones" ;
              people:age 27 ;
              vcard:N    [ vcard:Family "Jones" ;
                           vcard:Given  "Matthew" ] .
:RebeccaSmith vcard:FN   "Becky Smith" ;
              people:age 23 ;
              vcard:N    [ vcard:Family "Smith" ;
                           vcard:Given  "Rebecca" ] .
:JohnSmith    vcard:FN   "John Smith" ;
              vcard:N    [ vcard:Family "Smith" ;
                           vcard:Given  "John" ] .
:SarahJones   vcard:FN   "Sarah Jones" ;
              vcard:N    [ vcard:Family "Jones" ;
                           vcard:Given  "Sarah" ] .
```

Outline

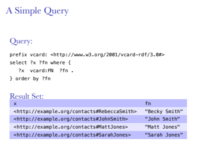
A Simple Query

Query:

```
prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
select ?x ?fn where {
   ?x  vcard:FN  ?fn .
} order by ?fn
```

Result Set:

x	fn
`<http://example.org/contacts#RebeccaSmith>`	`"Becky Smith"`
`<http://example.org/contacts#JohnSmith>`	`"John Smith"`
`<http://example.org/contacts#MattJones>`	`"Matt Jones"`
`<http://example.org/contacts#SarahJones>`	`"Sarah Jones"`



A Simple Query

Query:

```
prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
select ?x ?fn where {
   ?x  vcard:FN  ?fn .
} order by ?fn
```

Result Set:

x	fn
<http://example.org/contacts#RebeccaSmith>	"Becky Smith"
<http://example.org/contacts#JohnSmith>	"John Smith"
<http://example.org/contacts#MattJones>	"Matt Jones"
<http://example.org/contacts#SarahJones>	"Sarah Jones"

1. Here the where clause consists of a single “triple pattern,” i.e., a Turtle-formatted triple that may have variables in any of the three positions.
2. Variables are introduced with a question mark, which is not considered to be part of the variable’s name.
3. The select line lists the variables that should be returned in the result set.
4. The where clause specifies a graph pattern that the query processor searches for in the triple store.
5. Each row in the result set represents a subgraph matching the pattern of the where clause.
6. A row in the result set is also called a set of “bindings” for the variables.
7. I also added an ordering. Note that this will disable streaming of the result set, so it is best avoided when the results may be numerous.

A (Slightly) More Interesting Query

Query:

```
prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?x ?givenName where {
   ?x vcard:N ?y .
   ?y vcard:Family "Smith" ;
      vcard:Given  ?givenName .
}
```

Result Set:

x	givenName
`<http://example.org/contacts#JohnSmith>`	`"John"`
`<http://example.org/contacts#RebeccaSmith>`	`"Rebecca"`

A (Slightly) More Interesting Query

Query:

```
prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?x ?givenName where {
   ?x vcard:N ?y .
   ?y vcard:Family "Smith" ;
      vcard:Given  ?givenName .
}
```

Result Set:

x	givenName
<http://example.org/contacts#JohnSmith>	"John"
<http://example.org/contacts#RebeccaSmith>	"Rebecca"

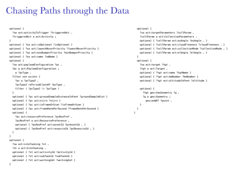
1. Shows Turtle-like syntax of the where clause, i.e., semi-colons, commas, and all the other syntactic sugar work here as well.
2. Shows typical pattern of SPARQL queries, where we chase paths through the data and pick up property values along the way.
3. The `vcard:N` and `vcard:Given` triple patterns represent such a path.
4. The `vcard:Family` triple pattern forms a restriction on the paths that match.
5. Note that one of the variables (`y`) is not returned in the result set.
6. Here the where clause consists of a Basic Graph Pattern (BGP), which is a simple set of triple patterns. It matches a sub-graph in the triple store when every triple pattern matches and the same value is used for every occurrence of each variable.

Chasing Paths through the Data

```
select distinct ?act ?cr ?actName ?actType ?priority ?sas ?sasName ?schedRestriction
  ?tp ?maximumSeparation ?minimumDuration ?minimumSeparation ?nominalDuration
  ?timesPerDay ?dayOfTheWeek ?wd ?laxity ?priorityWeight ?resourceCost ?rsrcMoveMinimize
  ?sensitivity ?sa ?isOptional ?lowestMoverPriority ?minBumperPriority ?saName
  ?pc ?pcType ?groundSampleDist ?niirs ?isFrameOrScan ?frameRatePerSecond
  ?sasResPref ?sasAssetId ?sasResourceId ?st ?activityId ?subTaskId ?workingSet
  ?triggeredAct ?triggeringSa ?pcResPref ?pcAssetId ?pcResourceId ?timeInt
  ?timeIntStart ?timeIntEnd ?timeOfDayInt ?timeOfDayIntStart ?timeOfDayIntEnd
  ?collParam ?azAngle ?cloudFreeness ?collectionMode ?elAngle ?tgt ?tgtName
  ?beNumber ?altitude ?g ?point
where {
  bind (<%1$s> as ?crUri)
  ?act a act:Activity ;
    act:collectionRequest ?crUri ;
    act:collectionRequest ?cr ;
    act:subActivity ?sas ;
    a ?actType .
  filter not exists {
    ?act a ?actType2 .
    ?actType2 rdfs:subClassOf ?actType .
    filter ( ?actType2 != ?actType )
  }
  optional { ?act act:name ?actName }
  optional { ?act act:priority ?priority }
  optional {
    ?triggeringSa a act:SubActivity ;
      act:activityToTrigger ?act .
  }
  ?sas a act:SubActivitySet .
  optional { ?sas act:name ?sasName }
  optional { ?sas act:schedulingRestriction ?schedRestriction }
  optional {
    ?sas act:timeParameters ?tp .
    ?tp a act:TimeParameters .
    optional { ?tp act:maximumSeparationInSeconds ?maximumSeparation }
    optional { ?tp act:minimumDurationInSeconds ?minimumDuration }
    optional { ?tp act:minimumSeparationInSeconds ?minimumSeparation }
    optional { ?tp act:nominalDurationInSeconds ?nominalDuration }
    optional { ?tp act:timesPerDay ?timesPerDay }
    optional { ?tp act:dayOfTheWeek ?dayOfTheWeek }
    optional {
      ?tp act:desiredInterval ?timeInt .
      ?timeInt a act:TimeInterval .
      optional { ?timeInt act:start ?timeIntStart . }
      optional { ?timeInt act:end ?timeIntEnd . }
    }
    optional {
      ?tp act:timeOfDay ?timeOfDayInt .
      ?timeOfDayInt a act:TimeOfDayInterval .
      optional { ?timeOfDayInt act:start ?timeOfDayIntStart . }
      optional { ?timeOfDayInt act:end ?timeOfDayIntEnd . }
    }
  }
  optional {
    ?sas act:worthDefinition ?wd .
    ?wd a act:WorthDefinition .
    optional { ?wd act:laxityWeight ?laxity }
    optional { ?wd act:priorityWeight ?priorityWeight }
    optional { ?wd act:resourceCostWeight ?resourceCost }
    optional { ?wd act:resourceMoveMinimizeWeight ?rsrcMoveMinimize }
    optional { ?wd act:sensitivityWeight ?sensitivity }
  }
  optional {
    ?sas act:resourcePreference ?sasResPref .
    ?sasResPref a act:ResourcePreference .
    optional { ?sasResPref act:assetId ?sasAssetId . }
    optional { ?sasResPref act:resourceId ?sasResourceId . }
  }
  ?sas act:subActivity ?sa .
  ?sa a act:SubActivity .
```

Chasing Paths through the Data

```
  optional {
    ?sa act:activityToTrigger ?triggeredAct .
    ?triggeredAct a act:Activity ;
  }
  optional { ?sa act:isOptional ?isOptional }
  optional { ?sa act:lowestMoverPriority ?lowestMoverPriority }
  optional { ?sa act:minBumperPriority ?minBumperPriority }
  optional { ?sa act:name ?saName }
  optional {
    ?sa act:payloadConfiguration ?pc .
    ?pc a act:PayloadConfiguration ;
      a ?pcType .
    filter not exists {
      ?pc a ?pcType2 .
      ?pcType2 rdfs:subClassOf ?pcType .
      filter ( ?pcType2 != ?pcType )
    }
    optional { ?pc act:groundSampleDistanceInFeet ?groundSampleDist }
    optional { ?pc act:niirs ?niirs }
    optional { ?pc act:isFrameOrScan ?isFrameOrScan }
    optional { ?pc act:frameRatePerSecond ?frameRatePerSecond }
    optional {
      ?pc act:resourcePreference ?pcResPref .
      ?pcResPref a act:ResourcePreference .
      optional { ?pcResPref act:assetId ?pcAssetId . }
      optional { ?pcResPref act:resourceId ?pcResourceId . }
    }
  }
  optional {
    ?sa act:siteTasking ?st .
    ?st a act:SiteTasking .
    optional { ?st act:activityId ?activityId }
    optional { ?st act:subTaskId ?subTaskId }
    optional { ?st act:workingSet ?workingSet }
  }
  optional {
    ?sa act:targetParameters ?collParam .
    ?collParam a act:CollectionParameters .
    optional { ?collParam act:azAngle ?azAngle . }
    optional { ?collParam act:cloudFreeness ?cloudFreeness . }
    optional { ?collParam act:collectionMode ?collectionMode . }
    optional { ?collParam act:elAngle ?elAngle . }
  }
  optional {
    ?sa act:target ?tgt .
    ?tgt a act:Target .
    optional { ?tgt act:name ?tgtName }
    optional { ?tgt act:beNumber ?beNumber }
    optional { ?tgt act:altitudeInFeet ?altitude }

    optional {
      ?tgt geo:hasGeometry ?g .
      ?g a geo:Geometry ;
        geo:asWKT ?point .
    }
  }
}
```



Chasing Paths through the Data

```
optional {
  ?sa act:activityToTrigger ?triggeredAct .
  ?triggeredAct a act:Activity ;
}
optional { ?sa act:isOptional ?isOptional }
optional { ?sa act:lowestMoverPriority ?lowestMoverPriority }
optional { ?sa act:minBumperPriority ?minBumperPriority }
optional { ?sa act:name ?saName }
optional {
  ?sa act:payloadConfiguration ?pc .
  ?pc a act:PayloadConfiguration ;
    a ?pcType .
  filter not exists {
    ?pc a ?pcType2 .
    ?pcType2 rdfs:subClassOf ?pcType .
    filter ( ?pcType2 != ?pcType )
  }
  optional { ?pc act:groundSampleDistanceInFeet ?groundSampleDist }
  optional { ?pc act:niirs ?niirs }
  optional { ?pc act:isFrameOrScan ?isFrameOrScan }
  optional { ?pc act:frameRatePerSecond ?frameRatePerSecond }
  optional {
    ?pc act:resourcePreference ?pcResPref .
    ?pcResPref a act:ResourcePreference .
    optional { ?pcResPref act:assetId ?pcAssetId . }
    optional { ?pcResPref act:resourceId ?pcResourceId . }
  }
}
optional {
  ?sa act:siteTasking ?st .
  ?st a act:SiteTasking .
  optional { ?st act:activityId ?activityId }
  optional { ?st act:subTaskId ?subTaskId }
  optional { ?st act:workingSet ?workingSet }
}
```

```
  optional {
    ?sa act:targetParameters ?collParam .
    ?collParam a act:CollectionParameters .
    optional { ?collParam act:azAngle ?azAngle . }
    optional { ?collParam act:cloudFreeness ?cloudFreeness . }
    optional { ?collParam act:collectionMode ?collectionMode . }
    optional { ?collParam act:elAngle ?elAngle . }
  }
  optional {
    ?sa act:target ?tgt .
    ?tgt a act:Target .
    optional { ?tgt act:name ?tgtName }
    optional { ?tgt act:beNumber ?beNumber }
    optional { ?tgt act:altitudeInFeet ?altitude }

    optional {
      ?tgt geo:hasGeometry ?g .
      ?g a geo:Geometry ;
        geo:asWKT ?point .
    }
  }
}
```

1. Shows the chasing of paths through the data in a serious example.

Outline

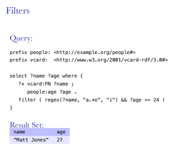
Filters

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name ;
      people:age ?age .
   filter ( regex(?name, "a.*o", "i") && ?age >= 24 )
}
```

Result Set:

name	age
"Matt Jones"	27



Filters

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name ;
      people:age ?age .
   filter ( regex(?name, "a.*o", "i") && ?age >= 24 )
}
```

Result Set:

name	age
"Matt Jones"	27

1. Filters are often not very efficient to evaluate.
2. They force the query processor to retrieve all instances of the pattern and then test them.
3. If you have an alternative, use it.
4. The regex syntax is from XQuery, which is a codified version of Perl's.

Available Filter Functionality

- Functional forms: bound, if, coalesce, not exists/exists, logical or/and, term equality, in/not in

Available Filter Functionality

- Functional forms: bound, if, coalesce, not exists/exists, logical or/and, term equality, in/not in
- RDF terms: isIRI, isBlank, isLiteral, isNumeric, str, lang, datatype, iri, bnode, strdt, strlang, UUID

Available Filter Functionality

- Functional forms: bound, if, coalesce, not exists/exists, logical or/and, term equality, in/not in
- RDF terms: isIRI, isBlank, isLiteral, isNumeric, str, lang, datatype, iri, bnode, strdt, strlang, UUID
- Strings: strlen, substr, ucase, lcase, strstarts, strends, contains, strbefore, strafter, encode_for_uri, concat, langMatches, regex, replace

Available Filter Functionality

- Functional forms: bound, if, coalesce, not exists/exists, logical or/and, term equality, in/not in
- RDF terms: isIRI, isBlank, isLiteral, isNumeric, str, lang, datatype, iri, bnode, strdt, strlang, UUID
- Strings: strlen, substr, ucase, lcase, strstarts, strends, contains, strbefore, strafter, encode_for_uri, concat, langMatches, regex, replace
- Numeric: abs, round, ceil, floor, rand

Available Filter Functionality

- Functional forms: bound, if, coalesce, not exists/exists, logical or/and, term equality, in/not in
- RDF terms: isIRI, isBlank, isLiteral, isNumeric, str, lang, datatype, iri, bnode, strdt, strlang, UUID
- Strings: strlen, substr, ucase, lcase, strstarts, strends, contains, strbefore, strafter, encode_for_uri, concat, langMatches, regex, replace
- Numeric: abs, round, ceil, floor, rand
- Dates and times: now, year, month, day, hours, minutes, seconds, timezone, tz

Available Filter Functionality

- Functional forms: bound, if, coalesce, not exists/exists, logical or/and, term equality, in/not in
- RDF terms: isIRI, isBlank, isLiteral, isNumeric, str, lang, datatype, iri, bnode, strdt, strlang, UUID
- Strings: strlen, substr, ucase, lcase, strstarts, strends, contains, strbefore, strafter, encode_for_uri, concat, langMatches, regex, replace
- Numeric: abs, round, ceil, floor, rand
- Dates and times: now, year, month, day, hours, minutes, seconds, timezone, tz
- Hashes: md5, sha1, sha256, sha384, sha512

Outline

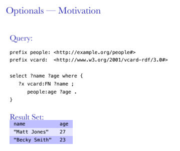
Optionals — Motivation

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name ;
      people:age ?age .
}
```

Result Set:

name	age
"Matt Jones"	27
"Becky Smith"	23



Optionals — Motivation

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name ;
      people:age ?age .
}
```

Result Set:

name	age
"Matt Jones"	27
"Becky Smith"	23

1. But what if we want to get the names of all the people, plus ages for those people that have them?

Optionals

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age }
}
```

Result Set:

name	age
"Matt Jones"	27
"Becky Smith"	23
"John Smith"	
"Sarah Jones"	

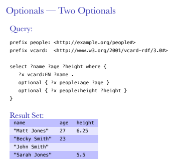
Optionals — Two Optionals

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age ?height where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age }
   optional { ?x people:height ?height }
}
```

Result Set:

name	age	height
"Matt Jones"	27	6.25
"Becky Smith"	23	
"John Smith"		
"Sarah Jones"		5.5



Optionals — Two Optionals

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age ?height where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age }
   optional { ?x people:height ?height }
}
```

Result Set:

name	age	height
"Matt Jones"	27	6.25
"Becky Smith"	23	
"John Smith"		
"Sarah Jones"		5.5

1. Suppose we do the same thing for another property (height).

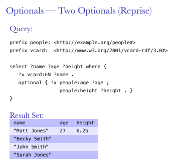
Optionals — Two Optionals (Reprise)

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age ?height where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age ;
                 people:height ?height . }
}
```

Result Set:

name	age	height
"Matt Jones"	27	6.25
"Becky Smith"		
"John Smith"		
"Sarah Jones"		



Optionals — Two Optionals (Reprise)

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age ?height where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age ;
                 people:height ?height . }
}
```

Result Set:

name	age	height
"Matt Jones"	27	6.25
"Becky Smith"		
"John Smith"		
"Sarah Jones"		

1. What happens when the two properties are together in one optional block?
2. Now the whole optional part must match before those variables are bound.

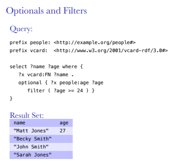
Optionals and Filters

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age
      filter ( ?age >= 24 ) }
}
```

Result Set:

name	age
"Matt Jones"	27
"Becky Smith"	
"John Smith"	
"Sarah Jones"	



Optionals and Filters

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age
      filter ( ?age >= 24 ) }
}
```

Result Set:

name	age
"Matt Jones"	27
"Becky Smith"	
"John Smith"	
"Sarah Jones"	

1. Similarly, a filter inside an optional block is evaluated before deciding whether to include the bindings within it.
2. Optional is a binary operator that combines two graph patterns. The second (optional) pattern is any group pattern and may involve any SPARQL pattern types. If the group matches, the solution is extended. If not, the original solution is given.
3. No age is included for Becky because she isn't 24.
4. No ages are included for John and Sarah because their ages are unknown.

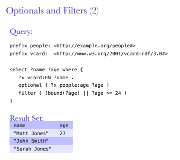
Optionals and Filters (2)

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age }
   filter ( !bound(?age) || ?age >= 24 )
}
```

Result Set:

name	age
"Matt Jones"	27
"John Smith"	
"Sarah Jones"	



Optionals and Filters (2)

Query:

```
prefix people: <http://example.org/people#>
prefix vcard:  <http://www.w3.org/2001/vcard-rdf/3.0#>

select ?name ?age where {
   ?x vcard:FN ?name .
   optional { ?x people:age ?age }
   filter ( !bound(?age) || ?age >= 24 )
}
```

Result Set:

name	age
"Matt Jones"	27
"John Smith"	
"Sarah Jones"	

1. What if the filter condition is moved out of the optional part?
2. In this example, if a solution has an age variable, then it must be greater than 24. It can also be unbound. Thus there are now three solutions.
3. We had to make the filter more complicated to allow for unbound variables.
4. Evaluating an expression which has an unbound variables where a bound one was expected causes an evaluation exception and the whole expression fails.

Optionals and Order-Dependence

Data:

```
:RebeccaSmith a            people:Person ;
              vcard:FN     "Becky Smith" ;
              people:name "Rebecca Smith" .
```

Query 1:

```
select ?x ?name where {
   ?x a people:Person .
   optional { ?x vcard:FN ?name }
   optional { ?x people:name ?name }
}
```

Query 2:

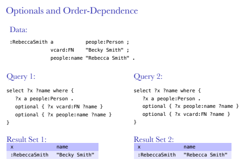
```
select ?x ?name where {
   ?x a people:Person .
   optional { ?x people:name ?name }
   optional { ?x vcard:FN ?name }
}
```

Result Set 1:

x	name
:RebeccaSmith	"Becky Smith"

Result Set 2:

x	name
:RebeccaSmith	"Rebecca Smith"



Optionals and Order-Dependence

Data:

```
:RebeccaSmith a           people:Person ;
              vcard:FN    "Becky Smith" ;
              people:name "Rebecca Smith" .
```

Query 1:

```
select ?x ?name where {
   ?x a people:Person .
   optional { ?x vcard:FN ?name }
   optional { ?x people:name ?name }
}
```

Query 2:

```
select ?x ?name where {
   ?x a people:Person .
   optional { ?x people:name ?name }
   optional { ?x vcard:FN ?name }
}
```

Result Set 1:

x	name
:RebeccaSmith	"Becky Smith"

Result Set 2:

x	name
:RebeccaSmith	"Rebecca Smith"

1. One thing to avoid is using the same variable in multiple optional clauses without using it in a non-optional pattern as well.
2. If the first optional does not match, then the second one is a new, independent attempt to bind `name`.
3. If the first optional does bind `x` and `name`, then the second optional is an attempt to match a ground triple (since `x` and `name` have values). Whether this attempt succeeds or fails doesn't matter, because it's optional.

Outline

Alternatives (Unions)

Data:

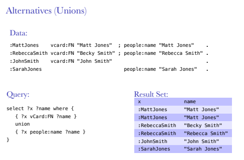
```
:MattJones    vcard:FN "Matt Jones"  ; people:name "Matt Jones"    .
:RebeccaSmith vcard:FN "Becky Smith" ; people:name "Rebecca Smith" .
:JohnSmith    vcard:FN "John Smith"                                .
:SarahJones                            people:name "Sarah Jones"   .
```

Query:

```
select ?x ?name where {
   { ?x vCard:FN ?name }
   union
   { ?x people:name ?name }
}
```

Result Set:

x	name
:MattJones	"Matt Jones"
:MattJones	"Matt Jones"
:RebeccaSmith	"Becky Smith"
:RebeccaSmith	"Rebecca Smith"
:JohnSmith	"John Smith"
:SarahJones	"Sarah Jones"



Alternatives (Unions)

Data:

```
:MattJones    vcard:FN "Matt Jones"  ; people:name "Matt Jones"    .
:RebeccaSmith vcard:FN "Becky Smith" ; people:name "Rebecca Smith" .
:JohnSmith    vcard:FN "John Smith"                                .
:SarahJones                            people:name "Sarah Jones"   .
```

Query:

```
select ?x ?name where {
   { ?x vCard:FN ?name }
   union
   { ?x people:name ?name }
}
```

Result Set:

x	name
:MattJones	"Matt Jones"
:MattJones	"Matt Jones"
:RebeccaSmith	"Becky Smith"
:RebeccaSmith	"Rebecca Smith"
:JohnSmith	"John Smith"
:SarahJones	"Sarah Jones"

1. Union concatenates the solutions from two possibilities.
2. More precisely, union computes the set-theoretic union of two sets of bindings.
3. Unions can behave like optionals, but there are subtle differences.
4. Use optional to augment solutions with additional properties.
5. Use union to concatenate solutions from two distinct queries.

Remembering the Source of Bindings within a Union

Data:

```
:MattJones     vcard:FN "Matt Jones"  ; people:name "Matt Jones"    .
:RebeccaSmith vcard:FN "Becky Smith" ; people:name "Rebecca Smith" .
:JohnSmith     vcard:FN "John Smith"                                .
:SarahJones                            people:name "Sarah Jones"    .
```

Query:

```
select ?x ?name1 ?name2 where {
   { ?x vCard:FN ?name1 }
   union
   { ?x people:name ?name2 }
}
```

Result Set:

x	name1	name2
:MattJones	"Matt Jones"	
:MattJones		"Matt Jones"
:RebeccaSmith	"Becky Smith"	
:RebeccaSmith		"Rebecca Smith"
:JohnSmith	"John Smith"	
:SarahJones		"Sarah Jones"

Outline

Distinct, Projection, Ordering, Limit/Offset

- The `distinct` modifier suppresses duplicate bindings:

```
select distinct ...
```

Distinct, Projection, Ordering, Limit/Offset

- The `distinct` modifier suppresses duplicate bindings:

```
select distinct ...
```

- Omitting variables from the `select` list that appear in the `where` clause is called "projection".

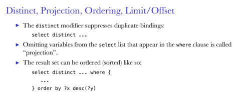
Distinct, Projection, Ordering, Limit/Offset

- The `distinct` modifier suppresses duplicate bindings:

```
select distinct ...
```

- Omitting variables from the `select` list that appear in the `where` clause is called "projection".
- The result set can be ordered (sorted) like so:

```
select distinct ... where {
   ...
} order by ?x desc(?y)
```



Distinct, Projection, Ordering, Limit/Offset

- The `distinct` modifier suppresses duplicate bindings:

```
select distinct ...
```

- Omitting variables from the `select` list that appear in the `where` clause is called "projection".
- The result set can be ordered (sorted) like so:

```
select distinct ... where {
   ...
} order by ?x desc(?y)
```

1. Ordering will suppress streaming of result sets, so consider sorting the results on the client side of the connection instead.

Distinct, Projection, Ordering, Limit/Offset

- The `distinct` modifier suppresses duplicate bindings:

```
select distinct ...
```

- Omitting variables from the `select` list that appear in the `where` clause is called “projection”.
- The result set can be ordered (sorted) like so:

```
select distinct ... where {
   ...
} order by ?x desc(?y)
```

- Use the `limit` modifier to prevent runaway queries:

```
select distinct ... where {
   ...
} limit 20
```

Distinct, Projection, Ordering, Limit/Offset

- The `distinct` modifier suppresses duplicate bindings:

```
select distinct ...
```

- Omitting variables from the `select` list that appear in the `where` clause is called "projection".
- The result set can be ordered (sorted) like so:

```
select distinct ... where {
   ...
} order by ?x desc(?y)
```

- Use the `limit` modifier to prevent runaway queries:

```
select distinct ... where {
   ...
} limit 20
```

- Paging (or windowing) of the result set is done like this:

```
select distinct ... where {
   ...
} order by ?name limit 5 offset 10
```

Construct Queries

- So far our queries have been select queries, which return a result set (table of bindings).

Construct Queries

- So far our queries have been select queries, which return a result set (table of bindings).
- Construct queries return RDF, not a result set. Many newcomers to SPARQL assume this is how SPARQL should work by default, but it isn't as useful as you might think. Select should be your default.

Construct Queries

- So far our queries have been select queries, which return a result set (table of bindings).
- Construct queries return RDF, not a result set. Many newcomers to SPARQL assume this is how SPARQL should work by default, but it isn't as useful as you might think. Select should be your default.
- Construct queries have a where clause that is evaluated just as for select queries.

Construct Queries

- So far our queries have been select queries, which return a result set (table of bindings).
- Construct queries return RDF, not a result set. Many newcomers to SPARQL assume this is how SPARQL should work by default, but it isn't as useful as you might think. Select should be your default.
- Construct queries have a where clause that is evaluated just as for select queries.
- The construct clause replaces the select clause. It instructs the query processor how to build a set of RDF statements for each query solution. (More on this in a minute.)

Construct Queries

- So far our queries have been select queries, which return a result set (table of bindings).
- Construct queries return RDF, not a result set. Many newcomers to SPARQL assume this is how SPARQL should work by default, but it isn't as useful as you might think. Select should be your default.
- Construct queries have a where clause that is evaluated just as for select queries.
- The construct clause replaces the select clause. It instructs the query processor how to build a set of RDF statements for each query solution. (More on this in a minute.)
- The RDF for all of the query solutions is combined via set-theoretic union into ***one graph,*** and this graph is the result of the query.

Evaluating Construct Queries

To Evaluate This:

```
construct {
   ?x people:name ?name
} where {
   ?x vCard:FN ?name
}
```

Evaluating Construct Queries

To Evaluate This:

```
construct {
   ?x people:name ?name
} where {
   ?x vCard:FN ?name
}
```

Do This First:

```
select distinct ?x ?name where {
   ?x vCard:FN ?name
}
```

Evaluating Construct Queries

To Evaluate This:

```
construct {
   ?x people:name ?name
} where {
   ?x vCard:FN ?name
}
```

Do This First:

```
select distinct ?x ?name where {
   ?x vCard:FN ?name
}
```

To Get This:

x	name
:MattJones	"Matt Jones"
:RebeccaSmith	"Becky Smith"
:JohnSmith	"John Smith"
:SarahJones	"Sarah Jones"

Evaluating Construct Queries

To Evaluate This:

```
construct {
   ?x people:name ?name
} where {
   ?x vCard:FN ?name
}
```

To Get This:

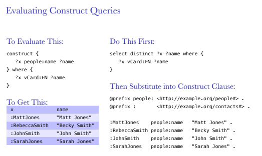
x	name
:MattJones	"Matt Jones"
:RebeccaSmith	"Becky Smith"
:JohnSmith	"John Smith"
:SarahJones	"Sarah Jones"

Do This First:

```
select distinct ?x ?name where {
   ?x vCard:FN ?name
}
```

Then Substitute into Construct Clause:

```
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .

:MattJones    people:name   "Matt Jones" .
:RebeccaSmith people:name   "Becky Smith" .
:JohnSmith    people:name   "John Smith" .
:SarahJones   people:name   "Sarah Jones" .
```



Evaluating Construct Queries

To Evaluate This:

```
construct {
   ?x people:name ?name
} where {
   ?x vCard:FN ?name
}
```

To Get This:

x	name
:MattJones	"Matt Jones"
:RebeccaSmith	"Becky Smith"
:JohnSmith	"John Smith"
:SarahJones	"Sarah Jones"

Do This First:

```
select distinct ?x ?name where {
   ?x vCard:FN ?name
}
```

Then Substitute into Construct Clause:

```
@prefix people: <http://example.org/people#> .
@prefix :       <http://example.org/contacts#> .

:MattJones    people:name   "Matt Jones" .
:RebeccaSmith people:name   "Becky Smith" .
:JohnSmith    people:name   "John Smith" .
:SarahJones   people:name   "Sarah Jones" .
```

1. Shows a transformation from one ontology to another.
2. Construct clause can also mirror the where clause. But, there is no shortcut for doing this.
3. Result is RDF: Means that if what you want is the names themselves (say to display in a GUI list), you must still walk the graph to extract that information.
4. One graph: Multiple solutions are combined, so again you must walk the graph dig each one out. This may require a lot of code for complex construct clauses.
5. Recommendation: Use construct only when you actually want an RDF graph as your end result.
6. For any solution with one or more unbound variables, any triple patterns in the construct clause involving those variables will not be evaluated, effectively suppressing them.

Construct Query Use Cases

- Use case 1: Transform RDF from one ontology to another (previous slide)

Construct Query Use Cases

- Use case 1: Transform RDF from one ontology to another (previous slide)
- Use case 2: Gather a subset of the original data store.

Construct Query Use Cases

- Use case 1: Transform RDF from one ontology to another (previous slide)
- Use case 2: Gather a subset of the original data store.

Get only the familiar names:

```
construct {
   ?x vCard:FN ?name
} where {
   ?x vCard:FN ?name
}
```

Construct Query Use Cases

- Use case 1: Transform RDF from one ontology to another (previous slide)
- Use case 2: Gather a subset of the original data store.

Get only the familiar names:

```
construct {
   ?x vCard:FN ?name
} where {
   ?x vCard:FN ?name
}
```

Or, more compactly:

```
construct where {
   ?x vCard:FN ?name
}
```

Construct Query Use Cases

- Use case 1: Transform RDF from one ontology to another (previous slide)
- Use case 2: Gather a subset of the original data store.

Get only the familiar names:

```
construct {
   ?x vCard:FN ?name
} where {
   ?x vCard:FN ?name
}
```

Or, more compactly:

```
construct where {
   ?x vCard:FN ?name
}
```

- To use the shorthand, the where clause must be a basic graph pattern — no filters, optionals, unions, negations, named graphs, property paths, etc.

Ask Queries

- Returns `true` or `false` according to whether the given pattern matches the data or not.

Ask Queries

- Returns `true` or `false` according to whether the given pattern matches the data or not.

Query:

```
prefix vcard: <...>
ask {
   ?x vcard:FN ?fn .
}
```

Result Set:

```
true
```

Ask Queries

- Returns `true` or `false` according to whether the given pattern matches the data or not.

Query:

```
prefix vcard: <...>
ask {
   ?x vcard:FN ?fn .
}
```

Result Set:

```
true
```

- Rarely used. (I have yet to find a use case.)

Describe Queries

- Returns RDF, not a result set, like construct queries.

Describe Queries

- Returns RDF, not a result set, like construct queries.
- Two forms: Explicit IRIs or resources from bindings.

Describe Queries

- Returns RDF, not a result set, like construct queries.
- Two forms: Explicit IRIs or resources from bindings.

Explicit IRI Query:

```
describe <http://example.org/contacts#MattJones>
```

Bound Resources Query:

```
prefix foaf: <...>
describe ?x ?y
where { ?x foaf:knows ?y }
```

Describe Queries

- Returns RDF, not a result set, like construct queries.
- Two forms: Explicit IRIs or resources from bindings.

Explicit IRI Query:

```
describe <http://example.org/contacts#MattJones>
```

Bound Resources Query:

```
prefix foaf: <...>
describe ?x ?y
where { ?x foaf:knows ?y }
```

- Returns a single result RDF graph containing data about the given resources.

Describe Queries

- Returns RDF, not a result set, like construct queries.
- Two forms: Explicit IRIs or resources from bindings.

Explicit IRI Query:

```
describe <http://example.org/contacts#MattJones>
```

Bound Resources Query:

```
prefix foaf: <...>
describe ?x ?y
where { ?x foaf:knows ?y }
```

- Returns a single result RDF graph containing data about the given resources.
- The data returned is determined by the SPARQL query processor.

Describe Queries

- Returns RDF, not a result set, like construct queries.
- Two forms: Explicit IRIs or resources from bindings.

Explicit IRI Query:

```
describe <http://example.org/contacts#MattJones>
```

Bound Resources Query:

```
prefix foaf: <...>
describe ?x ?y
where { ?x foaf:knows ?y }
```

- Returns a single result RDF graph containing data about the given resources.
- The data returned is determined by the SPARQL query processor.
- Rarely used.

Outline

Introduction to Bind

- Fetching the individual items from an online order:

```
select ?desc ?quantity ?unitPrice ?itemPrice where {
   <http://example.org/order-12345> :hasItem ?item .
   ?item a :OrderItem ;
      :description ?desc ;
      :quantity ?quantity ;
      :price ?unitPrice .
   bind (?quantity * ?unitPrice as ?itemPrice)
} order by ?itemPrice
```

- Gets order items for order 12345 (description, quantity, unit price)

Introduction to Bind

- Fetching the individual items from an online order:

```
select ?desc ?quantity ?unitPrice ?itemPrice where {
   <http://example.org/order-12345> :hasItem ?item .
   ?item a :OrderItem ;
      :description ?desc ;
      :quantity ?quantity ;
      :price ?unitPrice .
   bind (?quantity * ?unitPrice as ?itemPrice)
} order by ?itemPrice
```

- Gets order items for order 12345 (description, quantity, unit price)
- Last line computes the total price for the item and assigns it to `?itemPrice`

Introduction to Bind

- Fetching the individual items from an online order:

```
select ?desc ?quantity ?unitPrice ?itemPrice where {
   <http://example.org/order-12345> :hasItem ?item .
   ?item a :OrderItem ;
      :description ?desc ;
      :quantity ?quantity ;
      :price ?unitPrice .
   bind (?quantity * ?unitPrice as ?itemPrice)
} order by ?itemPrice
```

- Gets order items for order 12345 (description, quantity, unit price)
- Last line computes the total price for the item and assigns it to `?itemPrice`
- Can then use `?itemPrice` in other places in the query (select line, ordering clause)

Binding an IRI: Motivation

- Consider conference papers and the "reference" relationships between them:

```
select ?paper1 ?paper2 where {
   ?paper1 a :ConfPaper ;
      :references <http://example.org/paper-12345> .
   <http://example.org/paper-12345> a :ConfPaper ;
      :references ?paper2 .
   ?paper2 a :ConfPaper ;
      :references ?paper1 .
}
```

- This query finds cycles of length 3 that involve paper 12345.

Binding an IRI: Motivation

- Consider conference papers and the “reference” relationships between them:

```
select ?paper1 ?paper2 where {
   ?paper1 a :ConfPaper ;
      :references <http://example.org/paper-12345> .
   <http://example.org/paper-12345> a :ConfPaper ;
      :references ?paper2 .
   ?paper2 a :ConfPaper ;
      :references ?paper1 .
}
```

- This query finds cycles of length 3 that involve paper 12345.
- Unfortunately, the IRI for paper 12345 must be written out twice.

Binding an IRI: Motivation

- Consider conference papers and the "reference" relationships between them:

```
select ?paper1 ?paper2 where {
   ?paper1 a :ConfPaper ;
      :references <http://example.org/paper-12345> .
   <http://example.org/paper-12345> a :ConfPaper ;
      :references ?paper2 .
   ?paper2 a :ConfPaper ;
      :references ?paper1 .
}
```

- This query finds cycles of length 3 that involve paper 12345.
- Unfortunately, the IRI for paper 12345 must be written out twice.
- Also, it might be helpful to include this IRI in the result set.

Binding an IRI

- Here is the improved query:

```
select ?paper1 ?givenPaper ?paper2 where {
   bind (<http://example.org/paper-12345> as ?givenPaper)
   ?paper1 a :ConfPaper ;
      :references ?givenPaper .
   ?givenPaper a :ConfPaper ;
      :references ?paper2 .
   ?paper2 a :ConfPaper ;
      :references ?paper1 .
}
```

Binding an IRI

- Here is the improved query:

```
select ?paper1 ?givenPaper ?paper2 where {
   bind (<http://example.org/paper-12345> as ?givenPaper)
   ?paper1 a :ConfPaper ;
      :references ?givenPaper .
   ?givenPaper a :ConfPaper ;
      :references ?paper2 .
   ?paper2 a :ConfPaper ;
      :references ?paper1 .
}
```

- This technique is handy when writing code that dynamically assembles a query.

Multiple Bindings: Motivation

- Suppose we want the names of three specific types of entities that can have names:

```
select distinct ?name where {
   {
      ?iri a :Human .
   } union {
      ?iri a :Pet .
   } union {
      ?iri a :Ship .
   }
   ?iri :name ?name .
}
```

Multiple Bindings: Motivation

- Suppose we want the names of three specific types of entities that can have names:

```
select distinct ?name where {
   {
      ?iri a :Human .
   } union {
      ?iri a :Pet .
   } union {
      ?iri a :Ship .
   }
   ?iri :name ?name .
}
```

- This works, but it’s verbose and we can’t return the type in the results.

Multiple Bindings with “Bind”

- This enhanced query returns the type, but is still verbose:

```
select distinct ?name ?type where {
   {
      bind(:Human as ?type)
      ?iri a ?type .
   } union {
      bind(:Pet as ?type)
      ?iri a ?type .
   } union {
      bind(:Ship as ?type)
      ?iri a ?type .
   }
   ?iri :name ?name .
}
```

Multiple Bindings with “Values”

- This query is equivalent:

```
select distinct ?name ?type where {
   values ?type { :Human :Pet :Ship }
   ?iri a ?type ;
      :name ?name .
}
```

Multiple Bindings with “Values”

- This query is equivalent:

```
select distinct ?name ?type where {
   values ?type { :Human :Pet :Ship }
   ?iri a ?type ;
      :name ?name .
}
```

- The values statement successively binds `?type` to each of several values.

Bindings for Multiple Variables

Club Membership Data:

```
:Matt  :inCookingClub true ;
       :inChessClub   false .
:Becky :inCookingClub true ;
       :inChessClub   true .
:John  :inCookingClub false ;
       :inChessClub   true .
:Sarah :inCookingClub false ;
       :inChessClub   false .
```

Bindings for Multiple Variables

Club Membership Data:

```
:Matt  :inCookingClub true ;
       :inChessClub   false .
:Becky :inCookingClub true ;
       :inChessClub   true .
:John  :inCookingClub false ;
       :inChessClub   true .
:Sarah :inCookingClub false ;
       :inChessClub   false .
```

Query for specific combinations:

```
select ?person ?cooking ?chess where {
   ?person :inCookingClub ?cooking ;
      :inChessClub ?chess .
   values (?cooking ?chess) {
      (false true) (true undef)
   }
}
```

Bindings for Multiple Variables

Club Membership Data:

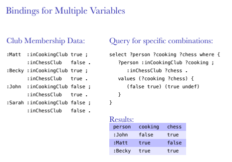
```
:Matt  :inCookingClub true ;
       :inChessClub   false .
:Becky :inCookingClub true ;
       :inChessClub   true .
:John  :inCookingClub false ;
       :inChessClub   true .
:Sarah :inCookingClub false ;
       :inChessClub   false .
```

Query for specific combinations:

```
select ?person ?cooking ?chess where {
   ?person :inCookingClub ?cooking ;
      :inChessClub ?chess .
   values (?cooking ?chess) {
      (false true) (true undef)
   }
}
```

Results:

person	cooking	chess
:John	false	true
:Matt	true	false
:Becky	true	true



Bindings for Multiple Variables

Club Membership Data:

```
:Matt  :inCookingClub true ;
       :inChessClub   false .
:Becky :inCookingClub true ;
       :inChessClub   true .
:John  :inCookingClub false ;
       :inChessClub   true .
:Sarah :inCookingClub false ;
       :inChessClub   false .
```

Query for specific combinations:

```
select ?person ?cooking ?chess where {
   ?person :inCookingClub ?cooking ;
      :inChessClub ?chess .
   values (?cooking ?chess) {
      (false true) (true undef)
   }
}
```

Results:

person	cooking	chess
:John	false	true
:Matt	true	false
:Becky	true	true

1. We have a database of who is a member of what club.
2. Our query fetches each person and their memberships.
3. The values statement restricts us to specific combinations of membership.
4. The term “undef” allows the corresponding variable have any (or no) value.
5. A single “values” statement with two variables differs from two “values” statements, one for each variable: The former restricts to exactly the combinations of values given, while the latter results in a Cartesian product of the values.

Outline

- Data
- Basic Select Queries
- Filters
- Optionals
- Alternatives (Unions)
- Producing Result Sets
- Assignment: Bind and Values
- Negation
- Property Paths
- Aggregates and Grouping
- Subqueries
- RDF Datasets and Named Graphs
- Conclusion

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.
- This is tricky to interpret:

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.
- This is tricky to interpret:
  - RDF adopts the Open World Assumption: The absence of a fact means only that we don't know whether it is true or not.

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.
- This is tricky to interpret:
  - RDF adopts the Open World Assumption: The absence of a fact means only that we don't know whether it is true or not.
  - Thus, negation is usually a query for what we *don't know*, not what *isn't true*.

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.
- This is tricky to interpret:
    - RDF adopts the Open World Assumption: The absence of a fact means only that we don't know whether it is true or not.
    - Thus, negation is usually a query for what we *don't know*, not what *isn't true*.
- There are two styles of negation:

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.
- This is tricky to interpret:
  - RDF adopts the Open World Assumption: The absence of a fact means only that we don't know whether it is true or not.
  - Thus, negation is usually a query for what we *don't know*, not what *isn't true*.
- There are two styles of negation:
  - Filtering based on facts not present ("not exists" filter), and

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.
- This is tricky to interpret:
  - RDF adopts the Open World Assumption: The absence of a fact means only that we don't know whether it is true or not.
  - Thus, negation is usually a query for what we *don't know*, not what *isn't true*.
- There are two styles of negation:
  - Filtering based on facts not present ("not exists" filter), and
  - Removing solutions that are not compatible with a second pattern ("minus" statement).

Negation Introduction

- SPARQL negation allows us to query for what is not present in our data set.
- This is tricky to interpret:
    - RDF adopts the Open World Assumption: The absence of a fact means only that we don't know whether it is true or not.
    - Thus, negation is usually a query for what we *don't know*, not what *isn't true*.
- There are two styles of negation:
    - Filtering based on facts not present ("not exists" filter), and
    - Removing solutions that are not compatible with a second pattern ("minus" statement).
- Many problems can be solved by either one, but they are not equivalent. They represent different ways of thinking about negation.

Not Exists Filter: An Example

Data Set:

```
:alice a :Person ;
   :name "Alice" .
:bob a :Person .
```

Not Exists Filter: An Example

Data Set:

```
:alice a :Person ;
   :name "Alice" .
:bob a :Person .
```

Query:

```
select ?person where {
   ?person a :Person .
   filter not exists { ?person :name ?name }
}
```

Not Exists Filter: An Example

Data Set:

```
:alice a :Person ;
   :name "Alice" .
:bob a :Person .
```

Query:

```
select ?person where {
   ?person a :Person .
   filter not exists { ?person :name ?name }
}
```

Results:

person
:bob

Not Exists Filter: An Example

Data Set:

```
:alice a :Person ;
   :name "Alice" .
:bob a :Person .
```

Query:

```
select ?person where {
   ?person a :Person .
   filter not exists { ?person :name ?name }
}
```

Results:

person
:bob

- This asks for all entities of type `:Person` for whom there is no `:name` statement.

Not Exists Filter: An Example

Data Set:

```
:alice a :Person ;
   :name "Alice" .
:bob a :Person .
```

Query:

```
select ?person where {
   ?person a :Person .
   filter not exists { ?person :name ?name }
}
```

Results:

person
:bob

- This asks for all entities of type `:Person` for whom there is no `:name` statement.
- More generally, the “filter not exists” construct yields all matches of the left-hand graph pattern for which there is no match of the right-hand graph pattern.

Minus Example

Data Set:

```
:alice :givenName "Alice" ;
       :familyName "Smith" .
:bob   :givenName "Bob" ;
       :familyName "Jones" .
```

Minus Example

Data Set:

```
:alice :givenName "Alice" ;
       :familyName "Smith" .
:bob   :givenName "Bob" ;
       :familyName "Jones" .
```

Query:

```
select distinct ?s where {
   ?s ?p ?o .
   minus { ?s :givenName "Bob" . }
}
```

Minus Example

Data Set:

```
:alice :givenName "Alice" ;
       :familyName "Smith" .
:bob   :givenName "Bob" ;
       :familyName "Jones" .
```

Query:

```
select distinct ?s where {
   ?s ?p ?o .
   minus { ?s :givenName "Bob" . }
}
```

Results:

s
:alice

Minus Example

Data Set:

```
:alice :givenName "Alice" ;
       :familyName "Smith" .
:bob   :givenName "Bob" ;
       :familyName "Jones" .
```

Query:

```
select distinct ?s where {
   ?s ?p ?o .
   minus { ?s :givenName "Bob" . }
}
```

Results:

s
:alice

- This computes all matches to the left-hand side, and then removes any binding that is consistent with the right-hand side.

Differences Between “Not Exists” and “Minus”

- What is the difference between this:

  ```
  select ∗ where {
     ?s ?p ?o
     filter not exists { ?x ?y ?z }
  }
  ```

  and this:

  ```
  select ∗ where {
     ?s ?p ?o
     minus { ?x ?y ?z }
  }
  ```

Differences Between “Not Exists” and “Minus”

- What is the difference between this:

  ```
  select * where {
     ?s ?p ?o
     filter not exists { ?x ?y ?z }
  }
  ```

  and this:

  ```
  select * where {
     ?s ?p ?o
     minus { ?x ?y ?z }
  }
  ```

- The first returns no results, while the second returns all triples in the data set.

Differences Between “Not Exists” and “Minus”

- What is the difference between this:

  ```
  select * where {
     ?s ?p ?o
     filter not exists { ?x ?y ?z }
  }
  ```

  and this:

  ```
  select * where {
     ?s ?p ?o
     minus { ?x ?y ?z }
  }
  ```

- The first returns no results, while the second returns all triples in the data set.
- In the first query, the triple pattern `?x ?y ?z` matches any solution to `?s ?p ?o`, so the “not exists” clause eliminates any solutions.

Differences Between “Not Exists” and “Minus”

- What is the difference between this:

  ```
  select * where {
     ?s ?p ?o
     filter not exists { ?x ?y ?z }
  }
  ```

  and this:

  ```
  select * where {
     ?s ?p ?o
     minus { ?x ?y ?z }
  }
  ```

- The first returns no results, while the second returns all triples in the data set.
- In the first query, the triple pattern `?x ?y ?z` matches any solution to `?s ?p ?o`, so the “not exists” clause eliminates any solutions.
- With minus, there is no shared variable between the first and second parts, so no bindings are eliminated.

Ian's Favorite Negation

- Suppose we want to retrieve any item belonging to a deep class hierarchy, and we also want to know the most specific class of each item.

```
select ?item ?mostDerivedType where {
   ?item a :RootType ;
      a ?mostDerivedType .
   ?mostDerivedType rdfs:subClassOf :RootType .
   filter not exists {
      ?item a ?someOtherType .
      ?someOtherType rdfs:subClassOf ?mostDerivedType .
      filter ( ?someOtherType != ?mostDerivedType )
   }
}
```

Ian's Favorite Negation

- Suppose we want to retrieve any item belonging to a deep class hierarchy, and we also want to know the most specific class of each item.

  ```
  select ?item ?mostDerivedType where {
     ?item a :RootType ;
        a ?mostDerivedType .
     ?mostDerivedType rdfs:subClassOf :RootType .
     filter not exists {
        ?item a ?someOtherType .
        ?someOtherType rdfs:subClassOf ?mostDerivedType .
        filter ( ?someOtherType != ?mostDerivedType )
     }
  }
  ```

- (This assumes subsumption inference.)

Ian’s Favorite Negation

- Suppose we want to retrieve any item belonging to a deep class hierarchy, and we also want to know the most specific class of each item.

  ```
  select ?item ?mostDerivedType where {
     ?item a :RootType ;
        a ?mostDerivedType .
     ?mostDerivedType rdfs:subClassOf :RootType .
     filter not exists {
        ?item a ?someOtherType .
        ?someOtherType rdfs:subClassOf ?mostDerivedType .
        filter ( ?someOtherType != ?mostDerivedType )
     }
  }
  ```

- (This assumes subsumption inference.)
- A similar query works with property hierarchies.

Outline

The Bacon Query

- The “six degrees of Kevon Bacon” query:

```
select ?x where {
   {
      :KevinBacon :isConnectedTo ?x
   } union {
      :KevinBacon :isConnectedTo ?tmp1
      ?tmp1 :isConnectedTo ?x
   } union {
      :KevinBacon :isConnectedTo ?tmp1
      ?tmp1 :isConnectedTo ?tmp2
      ?tmp2 :isConnectedTo ?x
   } union {
      ...
   }
}
```

The Bacon Query

- The “six degrees of Kevon Bacon” query:

```
select ?x where {
   {
      :KevinBacon :isConnectedTo ?x
   } union {
      :KevinBacon :isConnectedTo ?tmp1
      ?tmp1 :isConnectedTo ?x
   } union {
      :KevinBacon :isConnectedTo ?tmp1
      ?tmp1 :isConnectedTo ?tmp2
      ?tmp2 :isConnectedTo ?x
   } union {
      ...
   }
}
```

- This is verbose, and it doesn’t handle an arbitrary number of degrees.

The Bacon Query with a Path Expression

- The “N degrees of Kevon Bacon” query:

```
select ?x where {
   :KevinBacon :isConnectedTo+ ?x
}
```

The Bacon Query with a Path Expression

- The “N degrees of Kevon Bacon” query:

  ```
  select ?x where {
     :KevinBacon :isConnectedTo+ ?x
  }
  ```

- The way to think about the notation is as a regular expression on the predicate. In this case, the plus means “one or more” `:isConnectedTo` statements arranged in a continuous path.

The Bacon Query with a Path Expression

- The “N degrees of Kevon Bacon” query:

```
select ?x where {
   :KevinBacon :isConnectedTo+ ?x
}
```

- The way to think about the notation is as a regular expression on the predicate. In this case, the plus means “one or more” `:isConnectedTo` statements arranged in a continuous path.
- Warning: These queries can be computationally expensive.

Path Expression Syntax

Syntax	Description
iri	A single IRI, i.e., a path of length one.
ˆ*elt*	Inverse path, i.e., from object to subject.
elt1 / *elt2*	A sequence path of *elt1* followed by *elt2*.
elt1 | *elt2*	A alternative path of *elt1* or *elt2*. All possibilities are tried.
elt+	A path that connects the subject and object of the path by one or more matches of *elt*.
*elt**	A path that connects the subject and object of the path by zero or more matches of *elt*.
elt?	A path that connects the subject and object of the path by zero or one matches of *elt*.
(*elt*)	A group path. The parenthesis control precedence.
!*elt*	Negation

iri is an IRI (written in full, abbreviated by a prefixed name, or the keyword "a")
elt is a path element, which may itself be composed of path constructs

Path Expression Examples

- Fetch either title or label (or both):

```
:book1 dc:title|rdfs:label ?displayString
```

Path Expression Examples

- Fetch either title or label (or both):

  `:book1 dc:title|rdfs:label ?displayString`

- Find the names of people that Alice knows:

  `:alice foaf:knows/foaf:name ?name`

Path Expression Examples

- Fetch either title or label (or both):

```
:book1 dc:title|rdfs:label ?displayString
```

- Find the names of people that Alice knows:

```
:alice foaf:knows/foaf:name ?name
```

- Find pairs of distinct people that both know the same (unspecified) person:

```
?x foaf:knows/^foaf:knows ?y
filter ( ?x != ?y )
```

Path Expression Examples

- Fetch either title or label (or both):

  `:book1 dc:title|rdfs:label ?displayString`

- Find the names of people that Alice knows:

  `:alice foaf:knows/foaf:name ?name`

- Find pairs of distinct people that both know the same (unspecified) person:

  ```
  ?x foaf:knows/^foaf:knows ?y
  filter ( ?x != ?y )
  ```

- Find entities of type `:RootType`, without subsumption inference:

  `?x rdf:type/rdfs:subClassOf* :RootType`

Path Expression Examples

- Fetch either title or label (or both):

  ```
  :book1 dc:title|rdfs:label ?displayString
  ```

- Find the names of people that Alice knows:

  ```
  :alice foaf:knows/foaf:name ?name
  ```

- Find pairs of distinct people that both know the same (unspecified) person:

  ```
  ?x foaf:knows/^foaf:knows ?y
  filter ( ?x != ?y )
  ```

- Find entities of type `:RootType`, without subsumption inference:

  ```
  ?x rdf:type/rdfs:subClassOf* :RootType
  ```

- Find nodes connected by anything but `rdf:type` in either direction:

  ```
  ?x !(a|^a) ?y
  ```

Path Expression Examples

- Fetch either title or label (or both):

```
:book1 dc:title|rdfs:label ?displayString
```

- Find the names of people that Alice knows:

```
:alice foaf:knows/foaf:name ?name
```

- Find pairs of distinct people that both know the same (unspecified) person:

```
?x foaf:knows/^foaf:knows ?y
filter ( ?x != ?y )
```

- Find entities of type `:RootType`, without subsumption inference:

```
?x rdf:type/rdfs:subClassOf* :RootType
```

- Find nodes connected by anything but `rdf:type` in either direction:

```
?x !(a|^a) ?y
```

- Retrieve the elements in an RDF collection (in unspecified order):

```
:someList rdf:rest*/rdf:first ?element
```

Path Expression Notes

- No variables are allowed in the path expression. However, you can match any predicate with the sub-expression `(a|!a)`

Path Expression Notes

- No variables are allowed in the path expression. However, you can match any predicate with the sub-expression `(a|!a)`
- Path expressions are not intended to retrieve the paths between two endpoints — they only match the endpoints themselves. However, see below for work-arounds.

Finding the Paths (Intermediate Nodes)

- Recall the “N degrees of Kevon Bacon” query:

  ```
  select ?x where {
     :KevinBacon :isConnectedTo+ ?x
  }
  ```

  Unfortunately, this only gives you the endpoint of each path.

Finding the Paths (Intermediate Nodes)

- Recall the “N degrees of Kevon Bacon” query:

  ```
  select ?x where {
     :KevinBacon :isConnectedTo+ ?x
  }
  ```

  Unfortunately, this only gives you the endpoint of each path.

- To find the connections:

  ```
  construct {
     ?x1 :isConnectedTo ?x2 .
  } where {
     :KevinBacon :isConnectedTo* ?x1 .
     ?x1 :isConnectedTo ?x2 .
  }
  ```

  This creates a graph containing all paths originating at `:KevinBacon`. Traverse carefully — there may be cycles.

Finding the Paths (Intermediate Nodes)

- Recall the "N degrees of Kevon Bacon" query:

  ```
  select ?x where {
     :KevinBacon :isConnectedTo+ ?x
  }
  ```

  Unfortunately, this only gives you the endpoint of each path.

- To find the connections:

  ```
  construct {
     ?x1 :isConnectedTo ?x2 .
  } where {
     :KevinBacon :isConnectedTo* ?x1 .
     ?x1 :isConnectedTo ?x2 .
  }
  ```

  This creates a graph containing all paths originating at `:KevinBacon`. Traverse carefully — there may be cycles.

- Another approach:

  ```
  select ?x1 ?x2 where {
     :KevinBacon :isConnectedTo* ?x1 .
     ?x1 :isConnectedTo ?x2 .
  }
  ```

  In the code that processes the query result, create a multi-valued map with `?x1` as the key and `?x2` as the value. Then traverse the paths by map lookup (again taking care with cycles).

Outline

Introduction to Aggregates

- Aggregates apply expressions over groups of solutions.

Introduction to Aggregates

- Aggregates apply expressions over groups of solutions.
- By default a result set consists of a single group, containing all solutions.

Introduction to Aggregates

- Aggregates apply expressions over groups of solutions.
- By default a result set consists of a single group, containing all solutions.
- Grouping may be specified using the “group by” clause.

Introduction to Aggregates

- Aggregates apply expressions over groups of solutions.
- By default a result set consists of a single group, containing all solutions.
- Grouping may be specified using the “group by” clause.
- SPARQL 1.1 defines these aggregates: count, sum, min, max, avg, group_concat, and sample.

Introduction to Aggregates

- Aggregates apply expressions over groups of solutions.
- By default a result set consists of a single group, containing all solutions.
- Grouping may be specified using the “group by” clause.
- SPARQL 1.1 defines these aggregates: count, sum, min, max, avg, group_concat, and sample.
- Groups may be filtered with the “having” clause.

An Example Aggregate Query

- This query fetches online orders, sums the order items to compute a total price, and returns the orders whose total is over 1000:

```
select ?order (sum(?itemPrice) as ?totalPrice) where {
   ?order a :Order ;
      :hasItem ?item .
   ?item a :OrderItem ;
      :quantity ?quantity ;
      :price ?unitPrice .
   bind (?quantity * ?unitPrice as ?itemPrice)
} group by ?order having ( sum(?itemPrice) > 1000 )
```

An Example Aggregate Query

- This query fetches online orders, sums the order items to compute a total price, and returns the orders whose total is over 1000:

```
select ?order (sum(?itemPrice) as ?totalPrice) where {
   ?order a :Order ;
      :hasItem ?item .
   ?item a :OrderItem ;
      :quantity ?quantity ;
      :price ?unitPrice .
   bind (?quantity * ?unitPrice as ?itemPrice)
} group by ?order having ( sum(?itemPrice) > 1000 )
```

- This groups solutions according to the “group by” clause (all solutions where ?order takes a particular value appear in the same group).

An Example Aggregate Query

- This query fetches online orders, sums the order items to compute a total price, and returns the orders whose total is over 1000:

```
select ?order (sum(?itemPrice) as ?totalPrice) where {
   ?order a :Order ;
      :hasItem ?item .
   ?item a :OrderItem ;
      :quantity ?quantity ;
      :price ?unitPrice .
   bind (?quantity ∗ ?unitPrice as ?itemPrice)
} group by ?order having ( sum(?itemPrice) > 1000 )
```

- This groups solutions according to the "group by" clause (all solutions where ?order takes a particular value appear in the same group).
- It then evaluates the set function "sum" over each group.

An Example Aggregate Query

- This query fetches online orders, sums the order items to compute a total price, and returns the orders whose total is over 1000:

```
select ?order (sum(?itemPrice) as ?totalPrice) where {
   ?order a :Order ;
      :hasItem ?item .
   ?item a :OrderItem ;
      :quantity ?quantity ;
      :price ?unitPrice .
   bind (?quantity * ?unitPrice as ?itemPrice)
} group by ?order having ( sum(?itemPrice) > 1000 )
```

- This groups solutions according to the "group by" clause (all solutions where ?order takes a particular value appear in the same group).
- It then evaluates the set function "sum" over each group.
- Finally, groups are then filtered by the "having" clause, which removes groups whose sum is too small.

Aggregate Details

- If there is a “group by” clause, then variables that appear in the query pattern but not in the “group by” clause, can only be projected or used in aggregated select expressions.

Aggregate Details

- If there is a “group by” clause, then variables that appear in the query pattern but not in the “group by” clause, can only be projected or used in aggregated select expressions.
- If aggregates are used in “select”, “having”, or “order by” clauses, but the “group by” clause is not used, then the solution set is taken to be a single implicit group, to which all solutions belong.

Avoiding Cartesian Products (Motivation)

Data:

```
:bob :name  "Bob", "Robert", "Bobby" ;
     :phone "703-234-5678",
        "703-345-6789", "703-456-7890" .
```

Avoiding Cartesian Products (Motivation)

Data:

```
:bob :name  "Bob", "Robert", "Bobby" ;
     :phone "703-234-5678",
        "703-345-6789", "703-456-7890" .
```

Query:

```
select ?x ?name ?phone where {
   ?x :name ?name ;
      :phone ?phone .
}
```

Avoiding Cartesian Products (Motivation)

Data:

```
:bob :name  "Bob", "Robert", "Bobby" ;
     :phone "703-234-5678",
        "703-345-6789", "703-456-7890" .
```

Query:

```
select ?x ?name ?phone where {
   ?x :name ?name ;
      :phone ?phone .
}
```

Result Set:

x	name	phone
:bob	"Bob"	"703-234-5678"
:bob	"Robert"	"703-234-5678"
:bob	"Bobby"	"703-234-5678"
:bob	"Bob"	"703-345-6789"
:bob	"Robert"	"703-345-6789"
:bob	"Bobby"	"703-345-6789"
:bob	"Bob"	"703-456-7890"
:bob	"Robert"	"703-456-7890"
:bob	"Bobby"	"703-456-7890"

Avoiding Cartesian Products (Motivation)

Data:

```
:bob :name  "Bob", "Robert", "Bobby" ;
     :phone "703-234-5678",
        "703-345-6789", "703-456-7890" .
```

Result Set:

x	name	phone
:bob	"Bob"	"703-234-5678"
:bob	"Robert"	"703-234-5678"
:bob	"Bobby"	"703-234-5678"
:bob	"Bob"	"703-345-6789"
:bob	"Robert"	"703-345-6789"
:bob	"Bobby"	"703-345-6789"
:bob	"Bob"	"703-456-7890"
:bob	"Robert"	"703-456-7890"
:bob	"Bobby"	"703-456-7890"

Query:

```
select ?x ?name ?phone where {
   ?x :name ?name ;
      :phone ?phone .
}
```

- Notice the Cartesian product in the result set. This is a common source of performance problems.

Avoiding Cartesian Products

Data:

```
:bob :name  "Bob",
            "Robert",
            "Bobby" ;
     :phone "703-234-5678",
            "703-345-6789",
            "703-456-7890" .
```

Avoiding Cartesian Products

Data:

```
:bob :name  "Bob",
            "Robert",
            "Bobby" ;
     :phone "703-234-5678",
            "703-345-6789",
            "703-456-7890" .
```

Query:

```
select
   (group_concat(?name ; separator="|") as ?names)
   (group_concat(?phone ; separator="|") as ?phones)
where {
   ?x :name ?name ;
      :phone ?phone .
} group by ?x
```

Avoiding Cartesian Products

Data:

```
:bob :name  "Bob",
            "Robert",
            "Bobby" ;
     :phone "703-234-5678",
            "703-345-6789",
            "703-456-7890" .
```

Query:

```
select
   (group_concat(?name ; separator="|") as ?names)
   (group_concat(?phone ; separator="|") as ?phones)
where {
   ?x :name ?name ;
      :phone ?phone .
} group by ?x
```

Result Set:

names	phones
"Bob|Robert|Bobby"	"703-234-5678|703-345-6789|703-456-7890"

Avoiding Cartesian Products

Data:

```
:bob :name  "Bob",
            "Robert",
            "Bobby" ;
     :phone "703-234-5678",
            "703-345-6789",
            "703-456-7890" .
```

Query:

```
select
   (group_concat(?name ; separator="|") as ?names)
   (group_concat(?phone ; separator="|") as ?phones)
where {
   ?x :name ?name ;
      :phone ?phone .
} group by ?x
```

Result Set:

names	phones
"Bob|Robert|Bobby"	"703-234-5678|703-345-6789|703-456-7890"

- The Cartesian product is no more, but

Avoiding Cartesian Products

Data:

```
:bob :name  "Bob",
            "Robert",
            "Bobby" ;
     :phone "703-234-5678",
            "703-345-6789",
            "703-456-7890" .
```

Query:

```
select
   (group_concat(?name ; separator="|") as ?names)
   (group_concat(?phone ; separator="|") as ?phones)
where {
   ?x :name ?name ;
      :phone ?phone .
} group by ?x
```

Result Set:

names	phones
"Bob|Robert|Bobby"	"703-234-5678|703-345-6789|703-456-7890"

- The Cartesian product is no more, but
- Some parsing is required to process the results.

Outline

Data

Basic Select Queries

Filters

Optionals

Alternatives (Unions)

Producing Result Sets

Assignment: Bind and Values

Negation

Property Paths

Aggregates and Grouping

Subqueries

RDF Datasets and Named Graphs

Conclusion

Introduction to Subqueries

- Subqueries let us embed one SPARQL query within another query.

Introduction to Subqueries

- Subqueries let us embed one SPARQL query within another query.
- Common use case: Limiting the number of results from a sub-expression within the query.

Introduction to Subqueries

- Subqueries let us embed one SPARQL query within another query.
- Common use case: Limiting the number of results from a sub-expression within the query.
- Another use case: Query optimization.

Subquery Motivation

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Subquery Motivation

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
select ?x ?y ?n where {
   ?x :knows ?y .
   ?y :name ?n .
}
```

Subquery Motivation

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
select ?x ?y ?n where {
   ?x :knows ?y .
   ?y :name ?n .
}
```

- We get six results:

x	y	n
:alice	:bob	"Bob"
:alice	:bob	"Bob Bar"
:alice	:bob	"B. Bar"
:alice	:carol	"Carol"
:alice	:carol	"Carol Baz"
:alice	:carol	"C. Baz"

Subquery Motivation

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
select ?x ?y ?n where {
   ?x :knows ?y .
   ?y :name ?n .
}
```

- We get six results:

x	y	n
:alice	:bob	"Bob"
:alice	:bob	"Bob Bar"
:alice	:bob	"B. Bar"
:alice	:carol	"Carol"
:alice	:carol	"Carol Baz"
:alice	:carol	"C. Baz"

- But suppose that we don't care about all the names of `?y` — we just want one of them. What then?

Subquery Example

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
select ?x ?y ?anyN where {
   ?x :knows ?y .
   {
      select ?y (sample(?n) as ?anyN) where {
         ?y :name ?n .
      } group by ?y
   }
}
```

Subquery Example

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
select ?x ?y ?anyN where {
   ?x :knows ?y .
   {
      select ?y (sample(?n) as ?anyN) where {
         ?y :name ?n .
      } group by ?y
   }
}
```

- Logically, subqueries are evaluated first:

y	anyN
:alice	"A. Foo"
:bob	"Bob Bar"
:carol	"Carol"

Subquery Example

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
select ?x ?y ?anyN where {
   ?x :knows ?y .
   {
      select ?y (sample(?n) as ?anyN) where {
         ?y :name ?n .
      } group by ?y
   }
}
```

- Logically, subqueries are evaluated first:

y	anyN
:alice	"A. Foo"
:bob	"Bob Bar"
:carol	"Carol"

- Then subquery results are projected up to the outer query:

x	y	anyN
:alice	:bob	"Bob Bar"
:alice	:carol	"Carol"

Subquery Example

Data:

```
:alice :name "Alice", "Alice Foo", "A. Foo" ;
       :knows :bob, :carol .
:bob   :name "Bob", "Bob Bar", "B. Bar" .
:carol :name "Carol", "Carol Baz", "C. Baz" .
```

Query:

```
select ?x ?y ?anyN where {
   ?x :knows ?y .
   {
      select ?y (sample(?n) as ?anyN) where {
         ?y :name ?n .
      } group by ?y
   }
}
```

- Logically, subqueries are evaluated first:

y	anyN
:alice	"A. Foo"
:bob	"Bob Bar"
:carol	"Carol"

- Then subquery results are projected up to the outer query:

x	y	anyN
:alice	:bob	"Bob Bar"
:alice	:carol	"Carol"

- Only variables projected out of the subquery are visible to the outer query.

Outline

Datasets

- The fundamental RDF data model: A graph of (s, p, o) triples

Datasets

- The fundamental RDF data model: A graph of (s, p, o) triples
- Many RDF data stores hold multiple RDF graphs

Datasets

- The fundamental RDF data model: A graph of (s, p, o) triples
- Many RDF data stores hold multiple RDF graphs
- A SPARQL query is executed against an RDF Dataset which consists of:
  - One graph, the *default graph*, which does not have a name, and
  - Zero or more *named graphs*, where each named graph is identified by an IRI.

Datasets

- The fundamental RDF data model: A graph of (s, p, o) triples
- Many RDF data stores hold multiple RDF graphs
- A SPARQL query is executed against an RDF Dataset which consists of:
  - One graph, the *default graph*, which does not have a name, and
  - Zero or more *named graphs*, where each named graph is identified by an IRI.
- So far, all of our queries have been directed against only the default graph

Datasets

- The fundamental RDF data model: A graph of (s, p, o) triples
- Many RDF data stores hold multiple RDF graphs
- A SPARQL query is executed against an RDF Dataset which consists of:
  - One graph, the *default graph*, which does not have a name, and
  - Zero or more *named graphs*, where each named graph is identified by an IRI.
- So far, all of our queries have been directed against only the default graph
- A SPARQL query can match different parts of the query pattern against different graphs as described below

Datasets

- The fundamental RDF data model: A graph of (s, p, o) triples
- Many RDF data stores hold multiple RDF graphs
- A SPARQL query is executed against an RDF Dataset which consists of:
  - One graph, the *default graph*, which does not have a name, and
  - Zero or more *named graphs*, where each named graph is identified by an IRI.
- So far, all of our queries have been directed against only the default graph
- A SPARQL query can match different parts of the query pattern against different graphs as described below
- Two useful arrangements:
  - The default graph includes provenance information about the named graphs
  - The default graph contains the union of the named graphs

Query Over A Dataset

Given These Graphs:

```
# Default graph (located at http://example.org/dft)
@prefix :     <http://example.org/> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
:bob   dc:publisher "Bob Smith" .
:alice dc:publisher "Alice Jones" .

# Named graph: http://example.org/bob
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:b foaf:name "Bob" ;
   foaf:mbox <mailto:bob@oldcorp.example.org> .

# Named graph: http://example.org/alice
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:a foaf:name "Alice" ;
   foaf:mbox <mailto:alice@work.example.org> .
```

Query Over A Dataset

Given These Graphs:

```
# Default graph (located at http://example.org/dft)
@prefix :     <http://example.org/> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
:bob   dc:publisher "Bob Smith" .
:alice dc:publisher "Alice Jones" .

# Named graph: http://example.org/bob
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:b foaf:name "Bob" ;
   foaf:mbox <mailto:bob@oldcorp.example.org> .

# Named graph: http://example.org/alice
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:a foaf:name "Alice" ;
   foaf:mbox <mailto:alice@work.example.org> .
```

And This Query:

```
prefix :     <http://example.org/>
prefix dc:   <http://purl.org/dc/elements/1.1/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?who ?g ?mbox
from :dft
from named :alice
from named :bob
where {
   ?g dc:publisher ?who .
   graph ?g { ?x foaf:mbox ?mbox }
}
```

Query Over A Dataset

Given These Graphs:

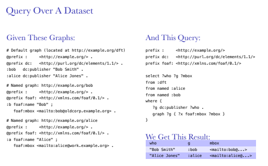
```
# Default graph (located at http://example.org/dft)
@prefix :     <http://example.org/> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
:bob   dc:publisher "Bob Smith" .
:alice dc:publisher "Alice Jones" .

# Named graph: http://example.org/bob
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:b foaf:name "Bob" ;
   foaf:mbox <mailto:bob@oldcorp.example.org> .

# Named graph: http://example.org/alice
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:a foaf:name "Alice" ;
   foaf:mbox <mailto:alice@work.example.org> .
```

And This Query:

```
prefix :     <http://example.org/>
prefix dc:   <http://purl.org/dc/elements/1.1/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?who ?g ?mbox
from :dft
from named :alice
from named :bob
where {
   ?g dc:publisher ?who .
   graph ?g { ?x foaf:mbox ?mbox }
}
```

We Get This Result:

who	g	mbox
"Bob Smith"	:bob	<mailto:bob@...>
"Alice Jones"	:alice	<mailto:alice@...>



Query Over A Dataset

Given These Graphs:

```
# Default graph (located at http://example.org/dft)
@prefix :     <http://example.org/> .
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
:bob   dc:publisher "Bob Smith" .
:alice dc:publisher "Alice Jones" .

# Named graph: http://example.org/bob
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:b foaf:name "Bob" ;
   foaf:mbox <mailto:bob@oldcorp.example.org> .

# Named graph: http://example.org/alice
@prefix :     <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
:a foaf:name "Alice" ;
   foaf:mbox <mailto:alice@work.example.org> .
```

And This Query:

```
prefix :     <http://example.org/>
prefix dc:   <http://purl.org/dc/elements/1.1/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?who ?g ?mbox
from :dft
from named :alice
from named :bob
where {
   ?g dc:publisher ?who .
   graph ?g { ?x foaf:mbox ?mbox }
}
```

We Get This Result:

who	g	mbox
"Bob Smith"	:bob	<mailto:bob@...>
"Alice Jones"	:alice	<mailto:alice@...>

1. Each graph is serialized in a document located at the graph's URI.
2. These three graphs show the arrangement where the default graph describes the named graphs.
3. In the query, the FROM establishes the default graph and each FROM NAMED declares a named graph.
4. Requires the query processor to fetch the three graphs over HTTP prior to performing the query.

Typical Triple Store Dataset

- Many triple stores have a default dataset, so that “from” and “from named” clauses are not necessary.

Typical Triple Store Dataset

- Many triple stores have a default dataset, so that “from” and “from named” clauses are not necessary.
- This lets us write a simpler query:

```
prefix dc:   <http://purl.org/dc/elements/1.1/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?who ?g ?mbox where {
   ?g dc:publisher ?who .
   graph ?g { ?x foaf:mbox ?mbox }
}
```

Typical Triple Store Dataset

- Many triple stores have a default dataset, so that “from” and “from named” clauses are not necessary.
- This lets us write a simpler query:

```
prefix dc:   <http://purl.org/dc/elements/1.1/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?who ?g ?mbox where {
   ?g dc:publisher ?who .
   graph ?g { ?x foaf:mbox ?mbox }
}
```

- Generally the default dataset starts with only the default graph.

Typical Triple Store Dataset

- Many triple stores have a default dataset, so that “from” and “from named” clauses are not necessary.
- This lets us write a simpler query:

```
prefix dc:   <http://purl.org/dc/elements/1.1/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

select ?who ?g ?mbox where {
   ?g dc:publisher ?who .
   graph ?g { ?x foaf:mbox ?mbox }
}
```

- Generally the default dataset starts with only the default graph.
- Using SPARQL Update statements, named graphs can be dynamically added and removed:

```
create graph <http://example.org/bob>
drop graph <http://example.org/bob>
```

Outline

Data

Basic Select Queries

Filters

Optionals

Alternatives (Unions)

Producing Result Sets

Assignment: Bind and Values

Negation

Property Paths

Aggregates and Grouping

Subqueries

RDF Datasets and Named Graphs

Conclusion

Other Topics Beyond the Scope of this Tutorial

- SPARQL Update
- Service Descriptions
- Federated Query
- Query Results, XML Format (Second Edition)
- Query Results, JSON Format
- Query Results, CSV and TSV Formats
- Entailment Regimes
- SPARQL Protocol
- Graph Store HTTP Protocol

Conclusion

- Don’t be afraid to consult the standard directly:

  `http://www.w3.org/TR/sparql11-query/`

- Most of it is quite accessible.