RDF Stores: SPARQL

Big Data Management

Anis Ur Rahman
Faculty of Computer Science & Information Technology
University of Malaya

October 1, 2019

Borrowed from http://www.ksi.mff.cuni.cz/~svoboda/courses/171-NDBI040/

Lecture Outline

RDF stores

- Introduction
- Linked Data

SPARQL query language

- Graph patterns
- Filter constraints
- Solution modifiers
- Aggregation
- Query forms

RDF Stores

Data model

- RDF **triples**
  - **Components.** subject, predicate, and object
  - Each triple **represents** a **statement** about a real-world entity
- Triples can be viewed as **graphs**
  - **Vertices** for **subjects** and **objects**
  - **Edges** directly correspond to individual **statements**

Query language

- **SPARQL.** SPARQL Protocol and RDF Query Language

Representatives

- Apache Jena, rdf4j (Sesame), Algebraix
- Multi-model: MarkLogic, OpenLink Virtuoso

Linked Data

Linked Data

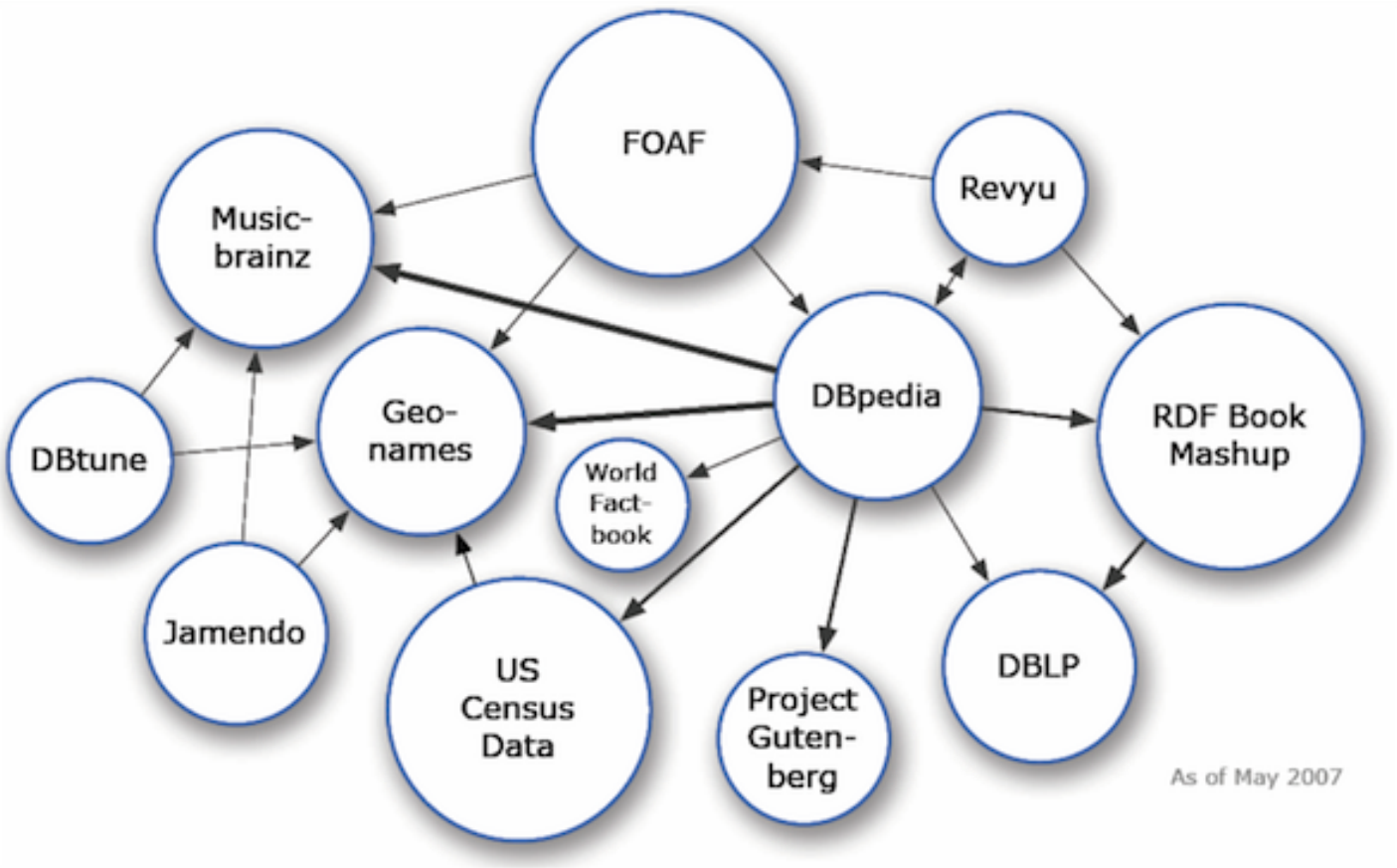
- To publish **structured** and **interlinked** data
- For **automated processing** by programs rather than browsing by human readers

Principles of Linked Open Data

- **Identify** resources using **URIs** or even better using **URLs**
- **Publish** data about resources in standard formats via **HTTP**
- **Mutually interlink** resources to form **Web of Data**
- **Release** the data under an **open licence**

Linked Open Data Cloud

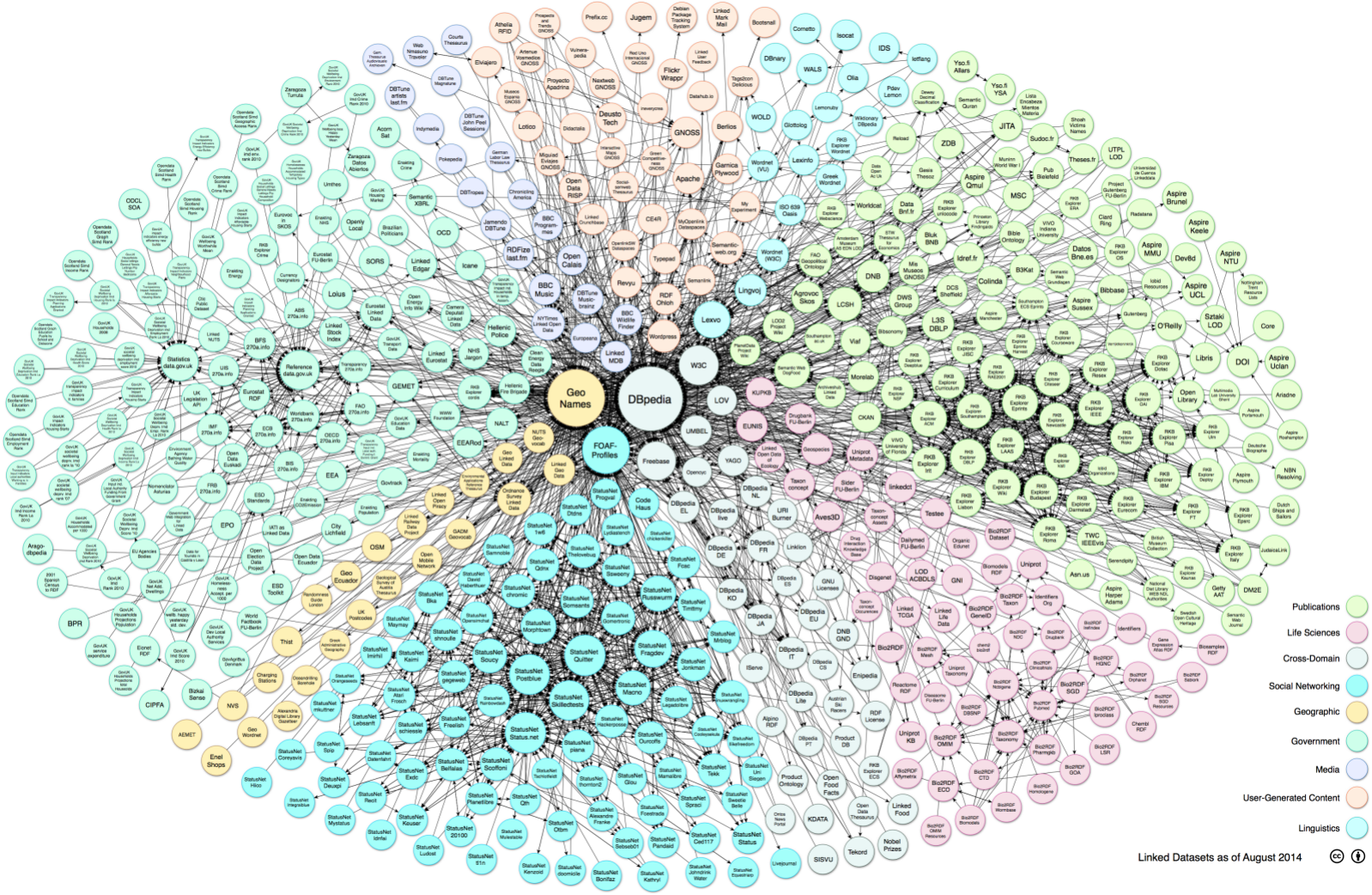
May 2007



Source: http://lod-cloud.net/

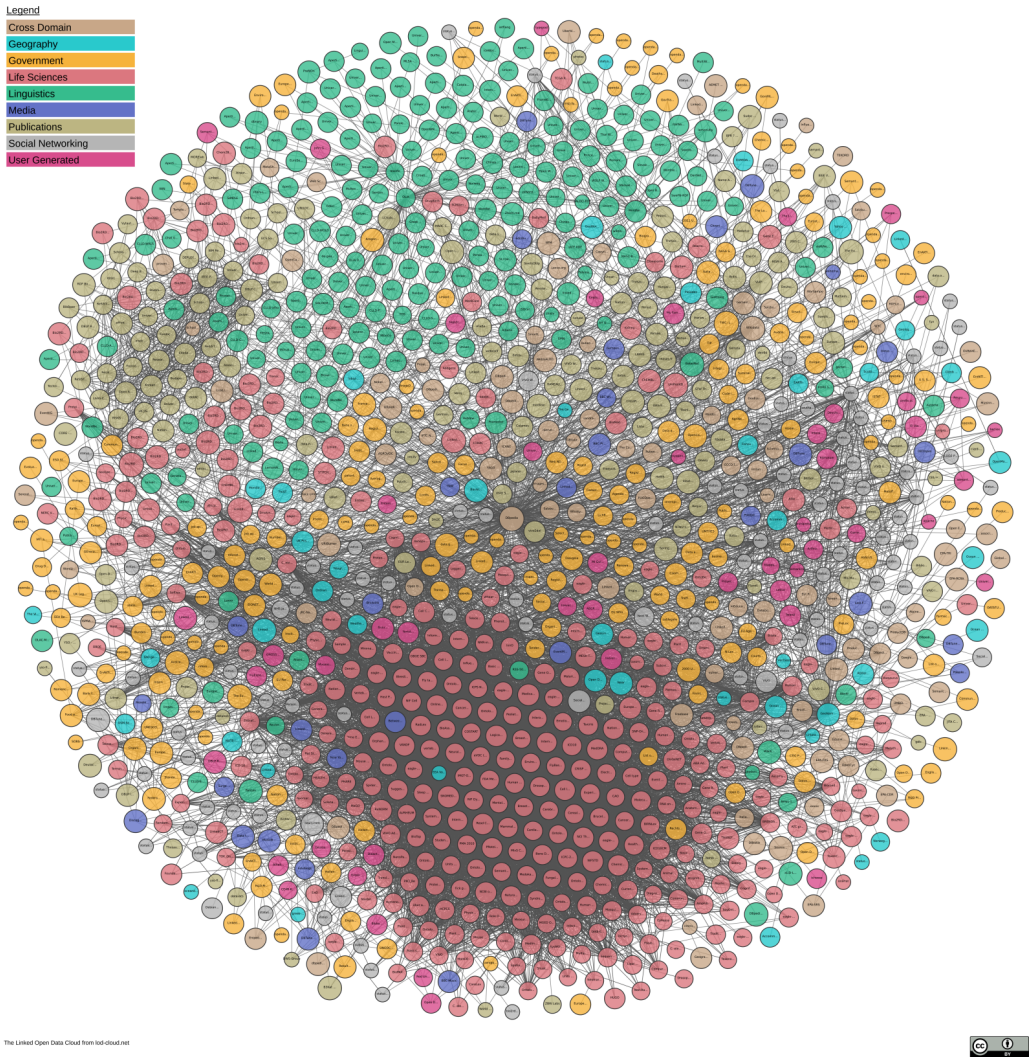
Linked Open Data Cloud

August 2014



Linked Open Data Cloud

August 2019



Outline

SPARQL

SPARQL Query Language

- Query language for **RDF data**
  - Graph patterns, optional graph patterns, subqueries, negation, aggregation, value constructors, ...
- **Versions.** 1.0 (2008), 1.1 (2013)
- W3C recommendations
  - `https://www.w3.org/TR/sparql11-query/`
  - Altogether 11 recommendations: query language, update facility, federated queries, protocol, result formats, ...

Sample Data

Graph of movies <http://db.my/movies>

```
@prefix i: <http://db.my/terms#> .
@prefix m: <http://db.my/movies/> .
@prefix a: <http://db.my/actors/> .
m:thedouble
  rdf:type i:Movie ; i:title "The Double" ;
  i:year "2007" ;
  i:actor a:wasikowska , a:eisenberg .
m:zombieland
  rdf:type i:Movie ; i:title "Zombieland" ;
  i:year "2009" ;
  i:actor a:stone , a:harrelson , a:eisenberg .
m:thesocialnetwork
  rdf:type i:Movie ; i:title "The Social Network" ;
  i:year "2010" ;
  i:actor a:eisenberg , a:harrelson ;
  i:director "David Fincher" .
m:Joker
  rdf:type i:Movie .
```

Sample Data

Graph of actors <http://db.my/actors>

```
@prefix i: <http://db.my/terms#> .
@prefix a: <http://db.my/actors/> .
a:harrelson
  rdf:type i:Actor ;
  i:firstname "Woody" ; i:lastname "Harrelson" ;
  i:year "1964" .
a:eisenberg
  rdf:type i:Actor ;
  i:firstname "Jesse" ; i:lastname "Eisenberg" ;
  i:year "1966" .
a:stone
  rdf:type i:Actor ;
  i:firstname "Emma" ; i:lastname "Stone" ;
  i:year "1973" .
a:wasikowska
  rdf:type i:Actor ;
  i:firstname "Mia" ; i:lastname "Wasikowska" ;
  i:year "1936" .
```

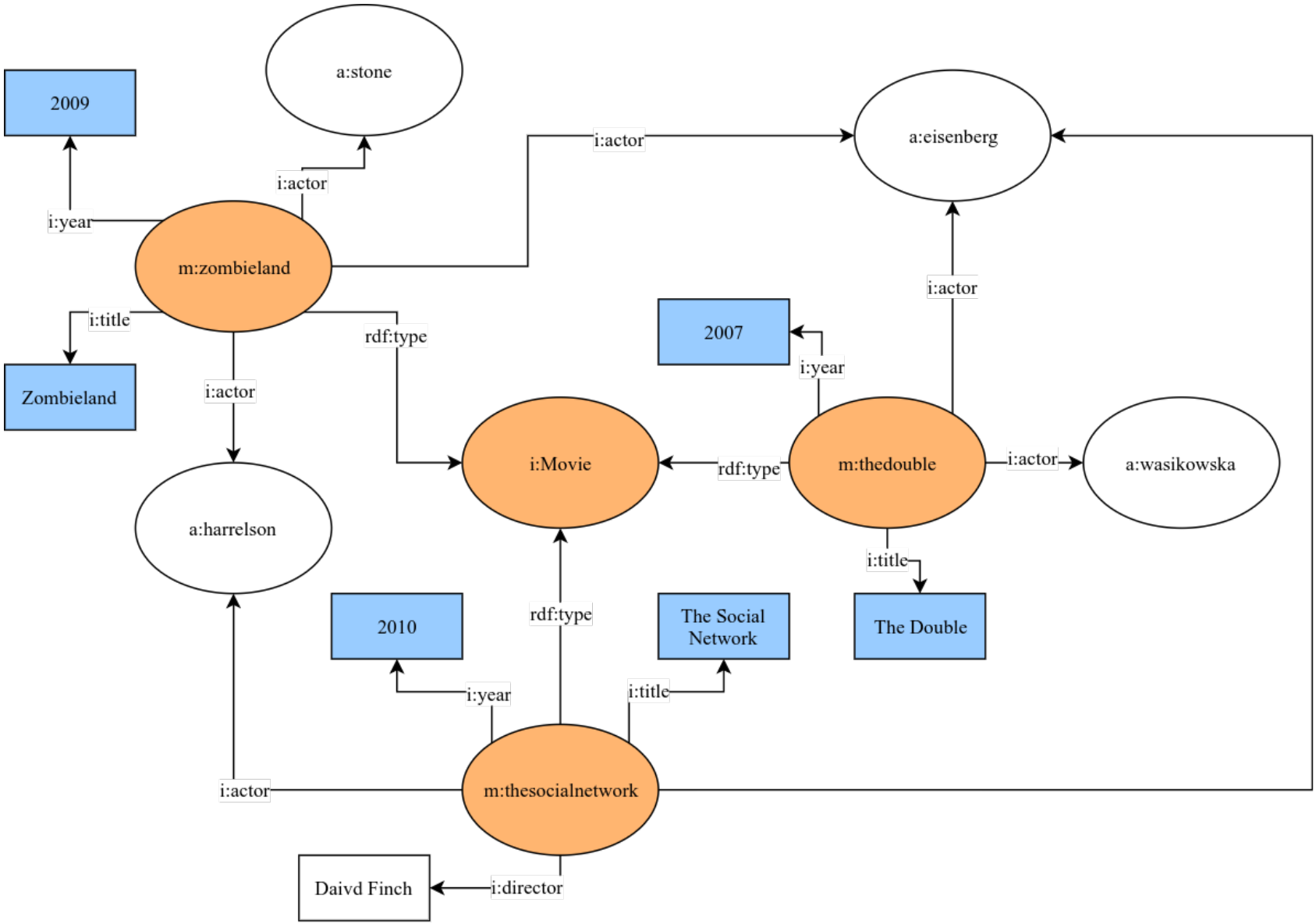
Sample Query

Find all movies, return their titles and years they were filmed

```
1  PREFIX i: <http://db.my/terms#>
2  SELECT ?t ?y
3  FROM <http://db.my/movies>
4  WHERE
5    {
6    ?m rdf:type i:Movie ;
7    i:title ?t ;
8    i:year ?y .
9    }
10 ORDER BY ?y
```

?t	?y
The Double	2007
Zombieland	2009
The Social Network	2010

Sample Query



Graph Pattern Matching

Graph patterns

1. **Basic graph pattern.** Based on ordinary **triples with variables**
   - ?variable or $variable
2. More **complicated** graph patterns
   - E.g. group, optional, minus, ...

Graph pattern matching

- Our goal is to **find all subgraphs** of the **data graph** that are **matched** by the **query graph pattern**
  - I.e. subgraphs of the data graph that are identical to the query graph pattern with variables substituted by particular terms
- One matching subgraph = one solution = one row of a table

Graph Pattern Matching

Query result = solution sequence = ordered multiset of solutions

?t	?y
Zombieland	2009
The Double	2007
The Social Network	2010

```
1 {
2   { (?t, "Zombieland"), (?y, "2009") },
3   { (?t, "The Double"), (?y, "2007") },
4   { (?t, "The Social Network"), (?y, "2010") }
5 }
```

Solution = set of variable bindings

?t	?y
Zombieland	2009

```
1 { (?t, "Zombieland"), (?y, "2009") }
```

Variable binding = pair of a variable name and a value it is assigned

?t
Zombieland

```
1 (?t, "Zombieland")
```

Graph Pattern Matching

Compatibility of solutions

- Two solutions are **mutually compatible** if and only if all the variables they share are **pairwise bound** to identical values

Examples

1. **Compatible** solutions

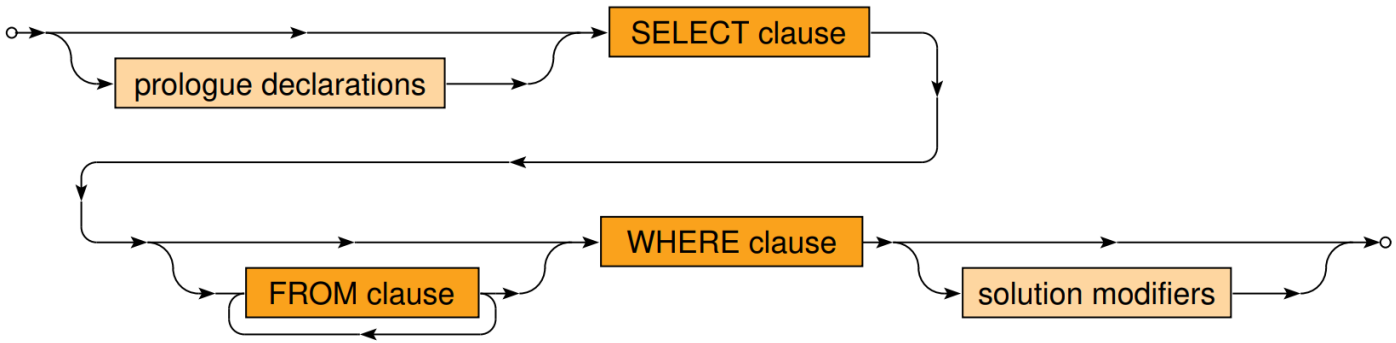
```
1 { (?m, m:zombieland), (?t, "Zombieland") }
2 { (?m, m:zombieland), (?y, "2009") }
```

2. **Incompatible** solutions

```
1 { (?m, m:zombieland), (?t, "Zombieland") }
2 { (?m, m:thesocialnetwork), (?y, "2010") }
```

Select Queries

SELECT queries

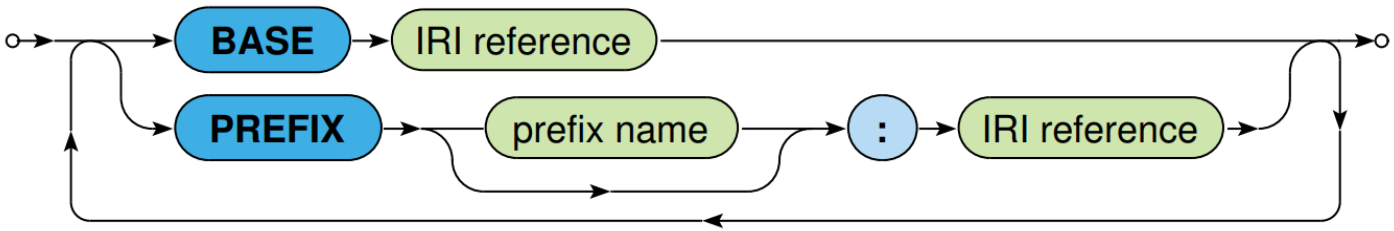


1. **Prologue declarations** – PREFIX, BASE
2. **Main clauses**
   - SELECT – variables to be projected
   - FROM – data graphs to be queried
   - WHERE – graph patterns to be matched
3. **Solution modifiers** – ORDER BY, ...

Prologue Declarations

Prologue declarations

- Allow to **simplify** IRI references **by declaring base IRIs**



BASE clause

- **One** single **base IRI** is defined **all relative IRI references** are then related to this base IRI

PREFIX clause

- **Several** **base IRIs** are defined, each is **associated** with a name all **prefixed names** are then related to the respective base IRI

Prologue Declarations

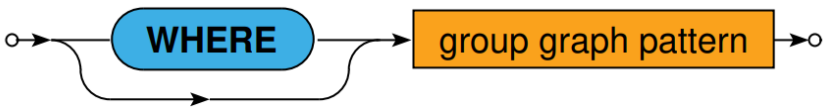
Examples

1. When **BASE <http://db.my/>** is defined, then
   - a **relative** IRI reference **terms#Movie** is interpreted as http://db.my/terms#Movie
2. When **PREFIX i: <http://db.my/>** is defined, then
   - a **prefixed** name **i:terms#Movie** is interpreted as http://db.my/terms#Movie

Where Clause

WHERE clause

- **Prescribes** one **group** graph pattern



Types of graph patterns

1. Basic – **triple patterns** to be matched
2. Group – **set of graph patterns** to be matched
3. Optional – graph pattern to be **matched only if possible**
4. Alternative – two or more **alternative graph patterns**
5. ...

Graph patterns can be **inductively combined** into complex ones

Graph Patterns

Basic graph pattern (triple block). One or more triple patterns to be all matched

- Ordinary triples **separated by .**
- ... or their **abbreviated forms** inspired by Turtle notation
  1. **Object lists** using ,
  2. **Predicate-object lists** using ;
  3. **Blank nodes** using []

Examples

```
1 s p1 o1 . s p1 o2 . s p2 o3 .
2 s p1 o1 , o2 ; p2 o3 .
```

Graph Patterns

Interpretation

- All the involved triple patterns must be **matched**
  - I.e. we combine them as if they were in **conjunction**
  - More precisely...
    - **Each** triple pattern is **evaluated** to its **solution sequence**
    - All combinations of **compatible solutions** are then **found**
- Note that all the **variables** need to be **bound**
  - I.e. if any of the involved variables **cannot be bound** at all, then
  - the entire basic graph pattern **cannot be matched!**

Graph Patterns: Example

Titles and years of all movies

```
1  PREFIX i: <http://db.my/terms#>
2  SELECT ?t ?y
3  FROM <http://db.my/movies>
4  WHERE
5    {
6      ?m rdf:type i:Movie . # triple 1
7      ?m i:title ?t . # triple 2
8      ?m i:year ?y . # triple 3
9    }
```

?t	?y
The Double	2007
Zombieland	2009
The Social Network	2010

Graph Patterns: Example

- t_1 =

?m
m:thedouble
m:zombieland
m:thesocialnetwork
m:joker

- t_2 =

?m	?t
m:thedouble	The Double
m:zombieland	Zombieland
m:thesocialnetwork	The Social Network

- t_3 =

?m	?y
m:thedouble	2006
m:zombieland	2009
m:thesocialnetwork	2007

$\Rightarrow$ The solution

?t	?y
The Double	2007
Zombieland	2009
The Social Network	2010

Graph Patterns

Equivalence of literals

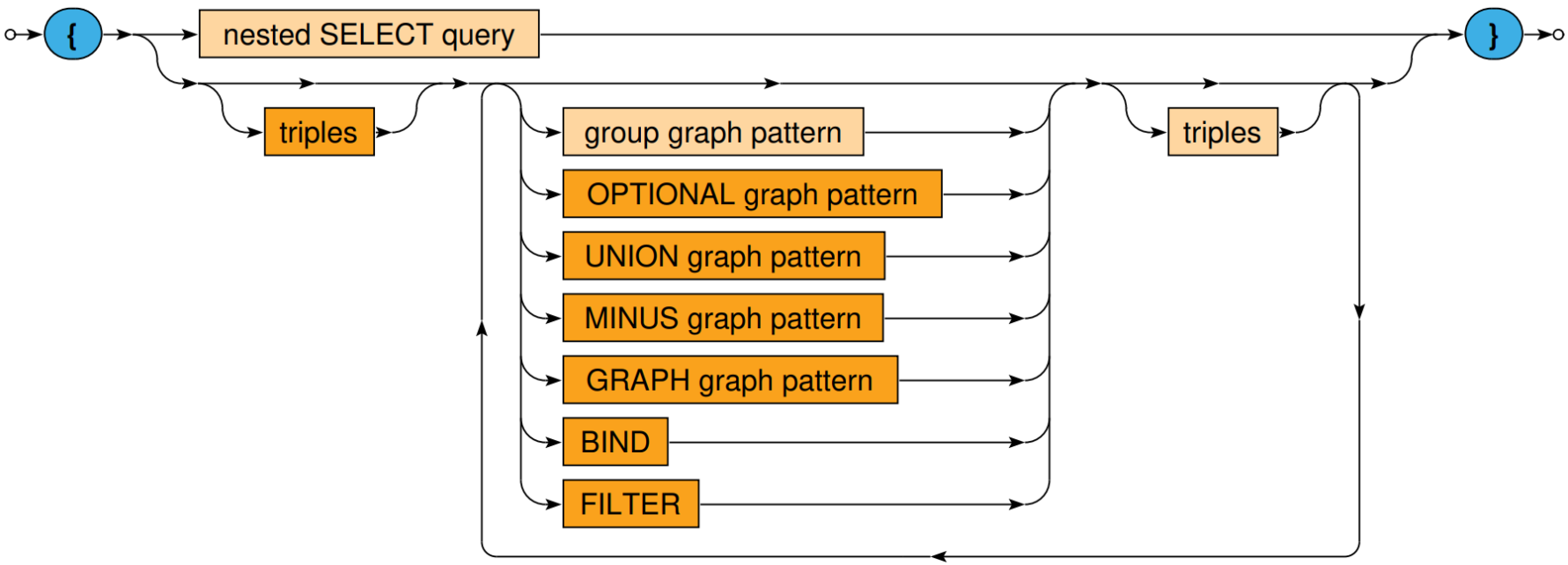
- Values must be **identical**
- Types/language tags when specified, must be **identical**
  - E.g.: "The Social Network"@cs != "The Social Network"

Equivalence of blank nodes

- Blank nodes in query patterns act as **non-selectable variables**
- Labels of blank nodes in data graphs/query graph patterns/query results **may not refer to the same nodes** despite being the same
  - I.e. the scope of **validity** is always **local only**

Graph Patterns

Set of graph patterns to be **all matched**



Group Graph Patterns

Two modes

1. **Nested SELECT** query
   - Only **with** **SELECT** and **WHERE** clauses and solution **modifiers**
   - i.e. **without** **FROM** clause
2. Set of graph patterns **interleaved** by triple blocks

Interpretation

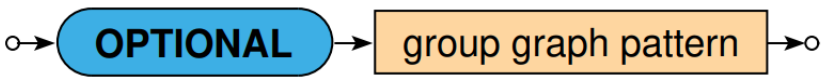
- All the involved graph patterns **must** be **matched**
- I.e. we combine them as if they were in conjunction

Note. **Empty group patterns** $\{\}$ are also allowed

Graph Patterns

OPTIONAL graph pattern

- **One** group graph pattern is tried to be matched



Interpretation

- When the optional part does **not match**, it creates **no bindings** but **does not eliminate the solution**

Graph Patterns: Example

Movies together with their directors when possible

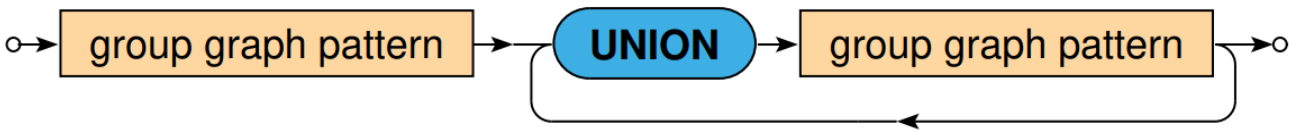
```
1  PREFIX i: <http://db.my/terms#>
2  SELECT ?t ?y ?d
3  FROM <http://db.my/movies>
4  WHERE
5    {
6      ?m rdf:type i:Movie ;
7      i:title ?t ;
8      i:year ?y .
9      OPTIONAL { ?m i:director ?d . }
10   }
```

?t	?y	?d
The Double	2007	
Zombieland	2009	
The Social Network	2010	David Fincher

Graph Patterns

UNION graph pattern

- Two or more group graph patterns are tried to be **matched**

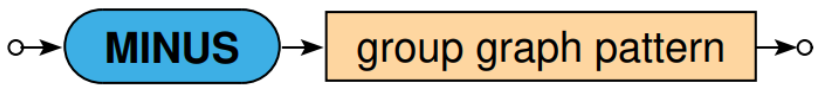


Interpretation. Standard set union of the **involved query results**

Graph Patterns

MINUS graph pattern

- **One group** graph pattern **removing** **compatible solutions**



Interpretation. Solutions of the first pattern are **preserved** if and only if they are **not compatible** with any solution of the second pattern

- I.e. minus graph pattern does not correspond to the standard set minus operation!

Graph Patterns: Example

Titles of movies that have no director

```
1 PREFIX i: <http://db.my/terms#>
2 SELECT ?t
3 FROM <http://db.my/movies>
4 WHERE
5   {
6     ?m rdf:type i:Movie ;
7     i:title ?t . # pattern 1
8     MINUS { ?m rdf:type i:Movie ; i:director ?d . } # pattern 2
9   }
```

?t
The Double
Zombieland

Graph Patterns: Example

Minus Graph Pattern

- p_1 =

?m	?t
m:thedouble	The Double
m:zombieland	Zombieland
m:thesocialnetwork	The Social Network

- p_2 =

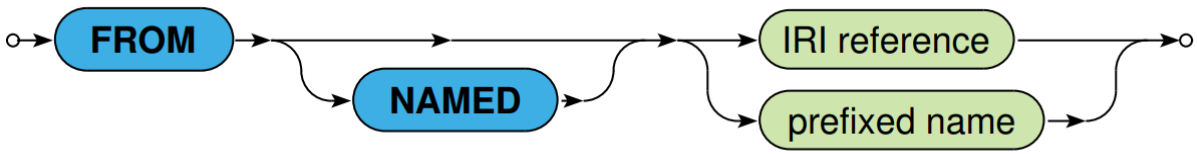
?m	?d
m:thesocialnetwork	David Fincher

⇒ The solution

?t
The Double
Zombieland

From Clause

FROM clause. Defines data graphs to be **queried**



Dataset = collection of graphs to be queried

1. One default graph
   - **Merge** of all the declared graphs from unnamed FROM clauses
   - **Empty** when no unnamed FROM clause is provided
2. Zero or more named graphs

Active graph = used for the **evaluation** of graph patterns

- The default graph unless changed using GRAPH graph pattern

From Clause: Example

Names of actors who played in The Social Network movie

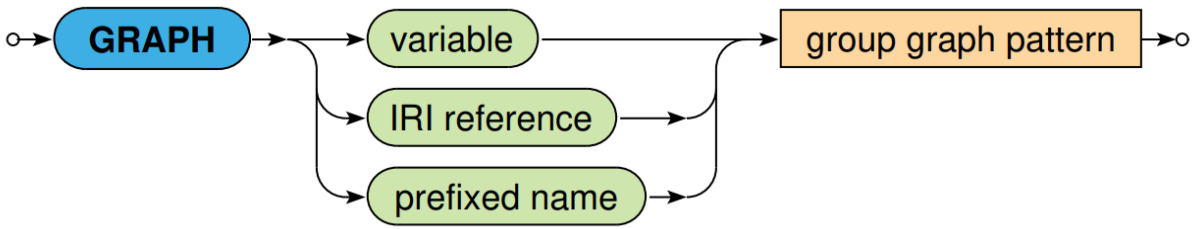
```
1  PREFIX i: <http://db.my/terms#>
2  PREFIX m: <http://db.my/movies/>
3  SELECT ?f ?l
4  FROM <http://db.my/movies>
5  FROM <http://db.my/actors>
6  WHERE
7    {
8      m:thesocialnetwork i:actor ?a .
9      ?a i:firstname ?f ; i:lastname ?l .
10   }
```

?f	?l
Jesse	Eisenberg
Woody	Harrelson

Graph Patterns

GRAPH graph pattern

- Pattern **evaluated** with respect to a **particular named graph**



- Changes the **active graph** for a given group graph pattern

```
1 GRAPH <http://db.my/actors> { ... }
```

- We can also consider **all the named graphs**

```
1 GRAPH ?g { ... }
```

Graph Patterns: Example

Names of actors who played in The Social Network movie

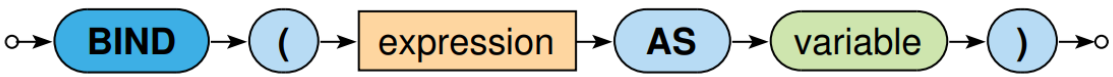
```
1  PREFIX i: <http://db.my/terms#>
2  PREFIX m: <http://db.my/movies/>
3  SELECT ?f ?l
4  FROM <http://db.my/movies>
5  FROM NAMED <http://db.my/actors>
6  WHERE
7    {
8      m:thesocialnetwork i:actor ?a .
9      GRAPH <http://db.my/actors> {
10         ?a i:firstname ?f ; i:lastname ?l .
11     }
12   }
```

?f	?l
Jesse	Eisenberg
Woody	Harrelson

Variable Assignments

BIND graph pattern

- Explicitly **assigns** a value to a given variable

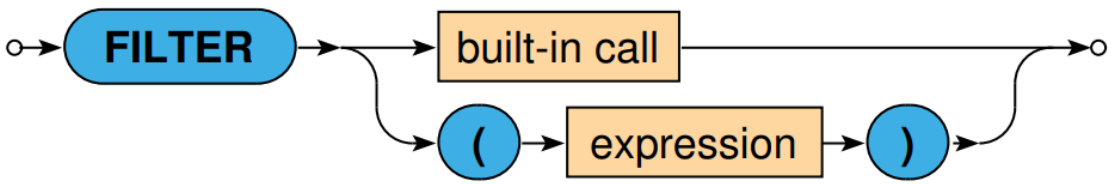


- This variable must not yet be bound!

Filter Constraints

FILTER constraints

- Impose **constraints** on variables and their values



- **Only solutions** satisfying the given condition are preserved
- Does not **create** any **new variable bindings!**
- Always **applied** on the **entire group** graph pattern i.e. evaluated at the very end

Filter Constraints: Example

Movies filmed in 2005 or later where Woody Harrelson played

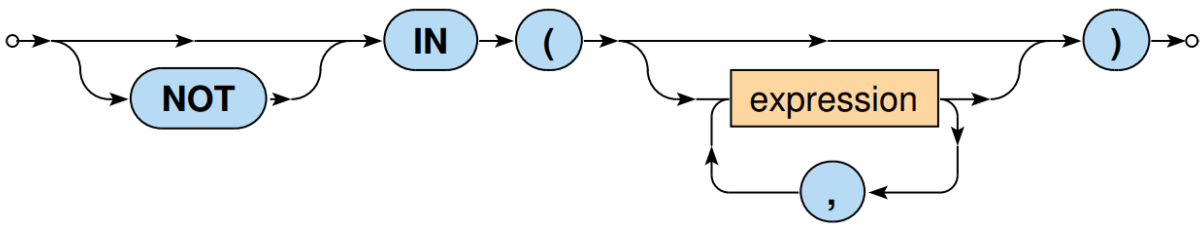
```
1  PREFIX i: <http://db.my/terms#>
2  PREFIX a: <http://db.my/actors/>
3  SELECT ?t ?y
4  FROM <http://db.my/movies>
5  WHERE
6    {
7      ?m rdf:type i:Movie ;
8        i:title ?t ;
9        i:year ?y .
10     FILTER (
11       (?y >= 2005) &&
12       EXISTS { ?m i:actor a:harrelson . }
13       )
14 }
```

?t	?y
Zombieland	2009
The Social Network	2010

Filter Constraints

Relational expressions

1. **Comparisons**
   - =, !=, <, <=, =>, >
   - Unbound variable < blank node < IRI < literal
2. **Set membership tests**
   - IN and NOT IN



Numeric expressions

- Unary/binary arithmetic operators +, -, *, /

Filter Constraints

Primary expressions

- Literals – numeric, boolean, RDF triples
- Variables
- Built-in calls
- Parentheses

Boolean expressions

1. **Logical connectives**
   - Conjunction &&, disjunction ||, negation !
2. **3-value logic** because of unbound variables (NULL values)
   - true, false, error

Filter Constraints

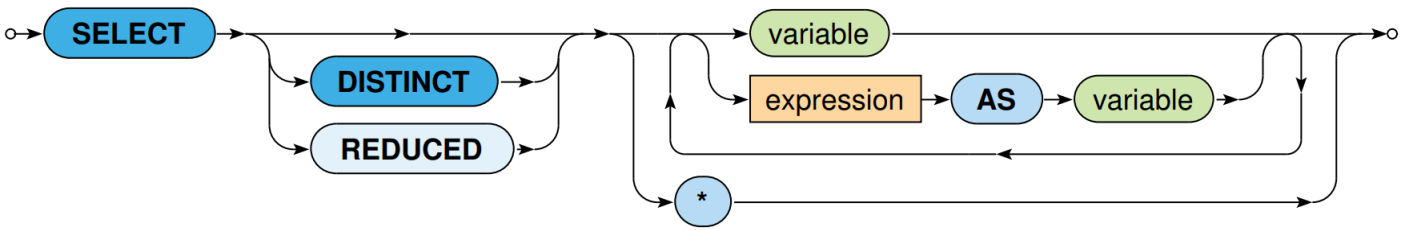
Built-in calls

1. **Term accessors**
   - **STR** – lexical form of an IRI or literal
   - **LANG** – language tag of a literal
   - **DATATYPE** – data type of a literal
2. **Variable tests**
   - **BOUND** – true when a variable is bound to a value
   - **isIRI**, **isBLANK**, **isLITERAL**
3. **Existence tests**
   - **EXISTS.** True when a provided group graph pattern is evaluated to at least one solution
   - **NOT EXISTS**

Select clause

SELECT clause

- **Enumerates variables** to be included in the **query result**



- Asterisk * selects all the variables

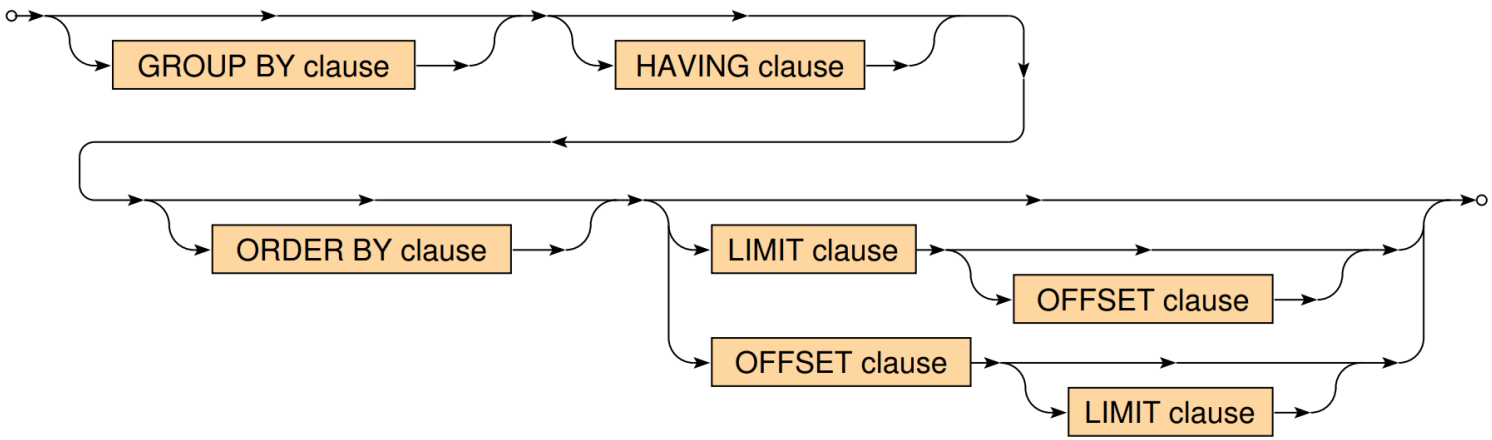
Solution modifiers

1. **DISTINCT** – duplicate solutions are removed
2. **REDUCED** – some duplicate solutions may be removed (implementation-dependent behavior)

Solution Modifiers

Solution modifiers – modify the entire solution sequence

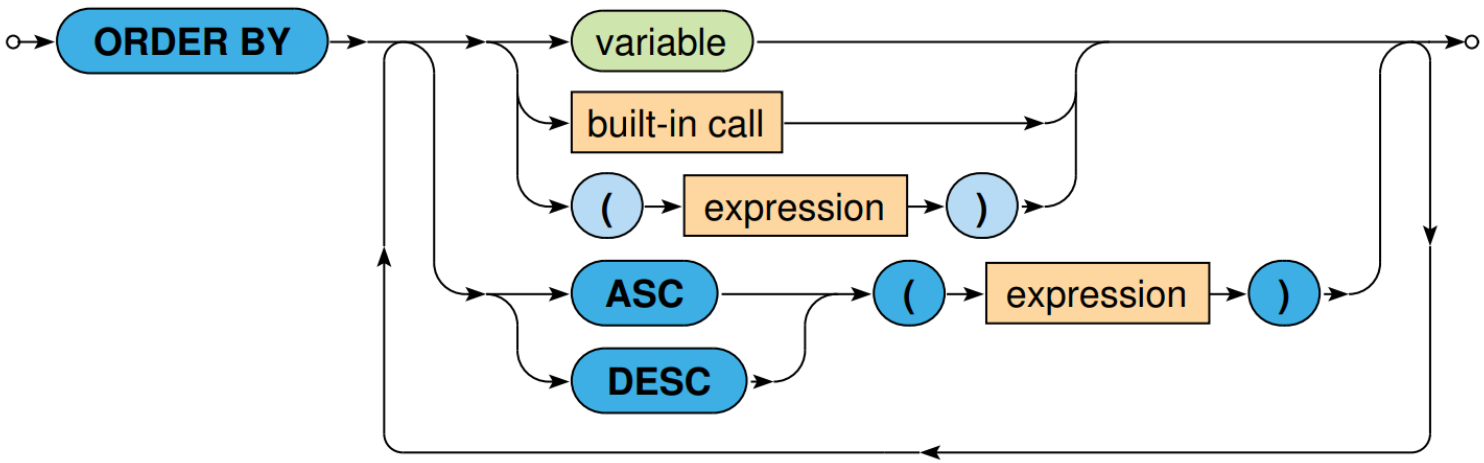
1. **Aggregation**
   - GROUP BY and HAVING
2. **Ordering**
   - ORDER BY
   - LIMIT and OFFSET



Solution Modifiers

ORDER BY clause

- Defines the order of solutions within the query result

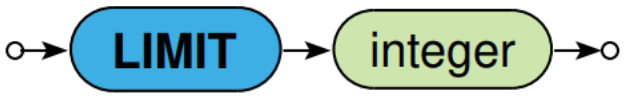


1. **ASC(...)** = ascending (default)
2. **DESC(...)** = descending

Solution Modifiers

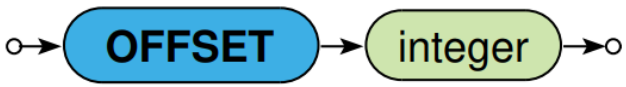
LIMIT clause

- Limits the **number of solutions** in the query result



OFFSET clause

- Skips a certain number of solutions in the query result



Solution Modifiers: Example

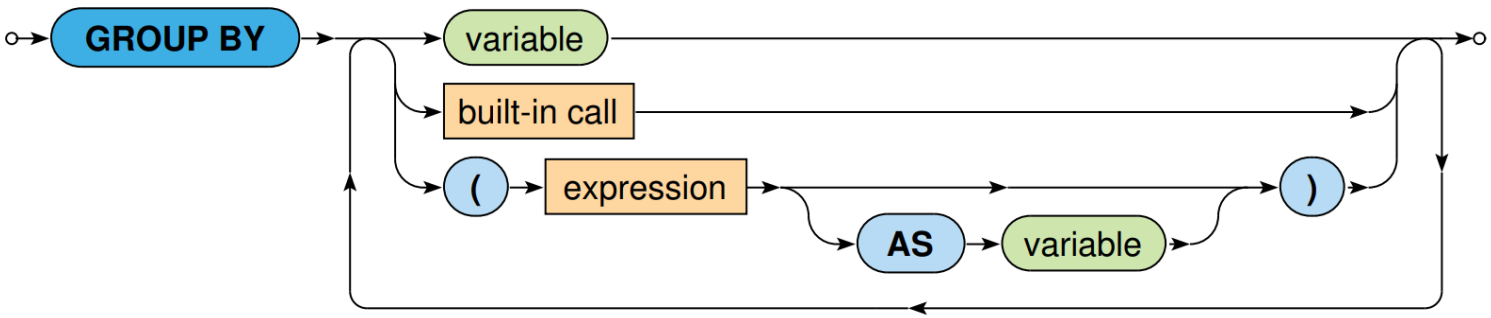
```
1  PREFIX i: <http://db.my/terms#>
2  SELECT ?t ?y
3  FROM <http://db.my/movies>
4    WHERE
5    {
6      ?m rdf:type i:Movie ;
7        i:title ?t ;
8        i:year ?y .
9    }
10 ORDER BY DESC(?y) ASC(?t)
11 OFFSET 1
12 LIMIT 5
```

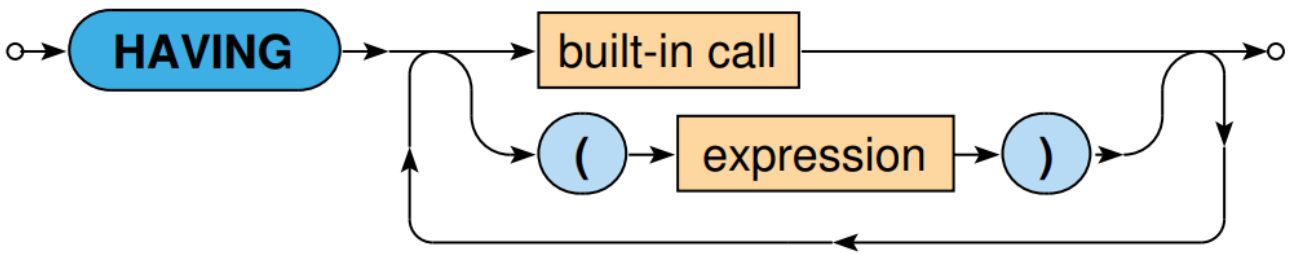
?t	?y
The Double	2007
Zombieland	2009

Aggregation

GROUP BY + HAVING clauses

- Standard aggregation over a solution sequence





Aggregation: Example

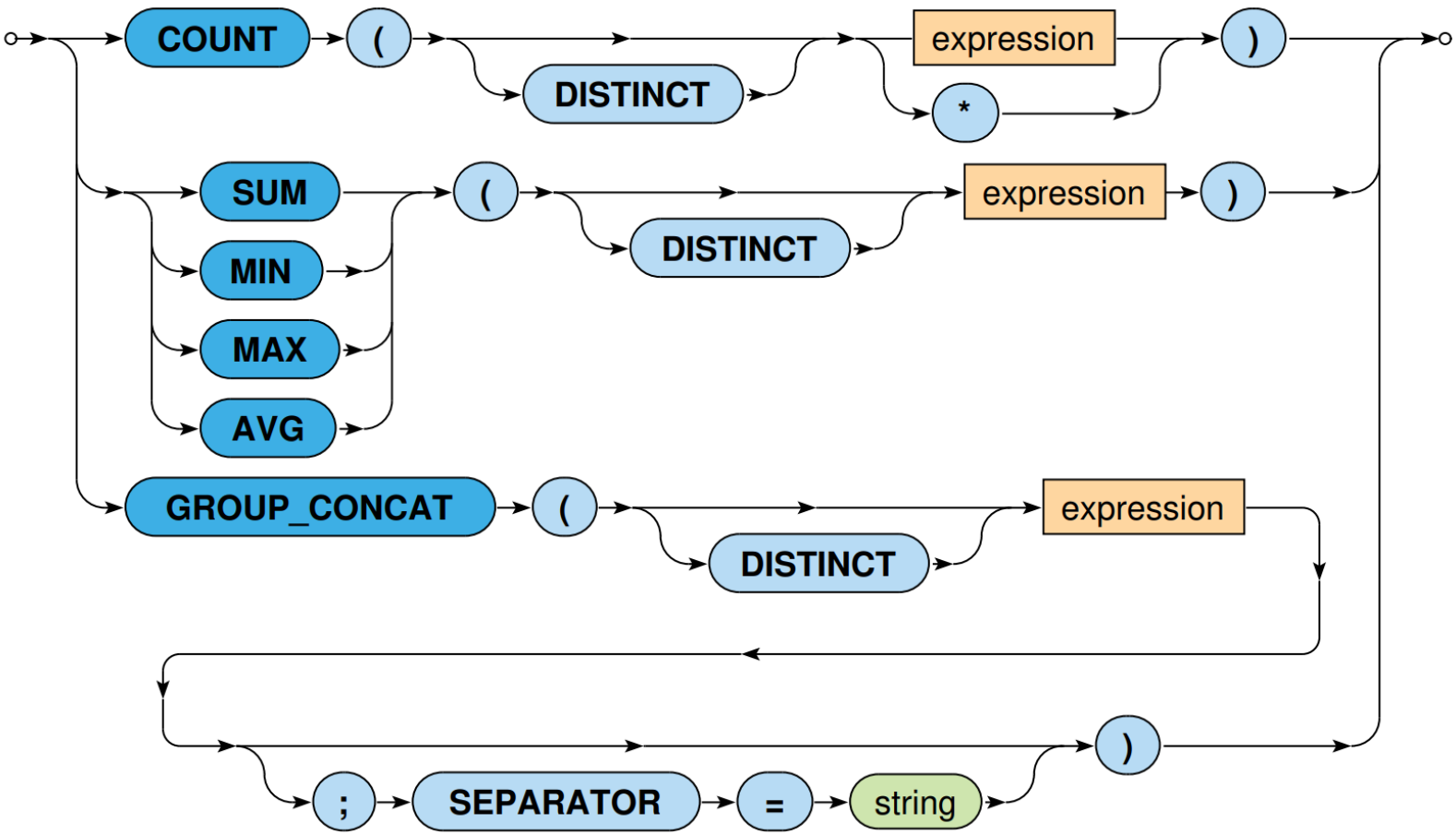
Numbers of actors in movies with at most 2 actors

```
1  PREFIX i: <http://db.my/terms#>
2  SELECT ?t (COUNT(?a) AS ?c)
3  FROM <http://db.my/movies>
4    WHERE
5    {
6      ?m rdf:type i:Movie ;
7        i:title ?t ;
8        i:actor ?a .
9    }
10 GROUP BY ?m ?t
11 HAVING (?c <= 2)
12 ORDER BY ?c ?t
```

?t	?c
The Double	2
The Social Network	2

Aggregation

Aggregate functions



Query Forms

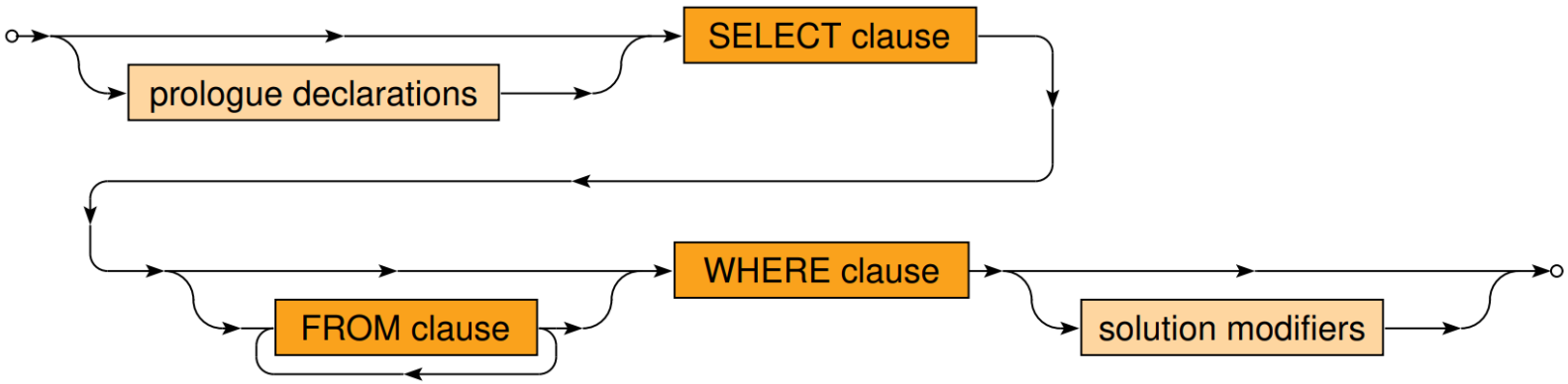
Query forms

- **SELECT.** Finds solutions matching a provided graph pattern
- **ASK.** Checks whether at least one solution exists
- **DESCRIBE.** Retrieves a graph with data about selected resources
- **CONSTRUCT.** Creates a new graph according to a provided pattern

Query Forms

SELECT query form

- Finds solutions **matching** a provided graph pattern prologue declarations

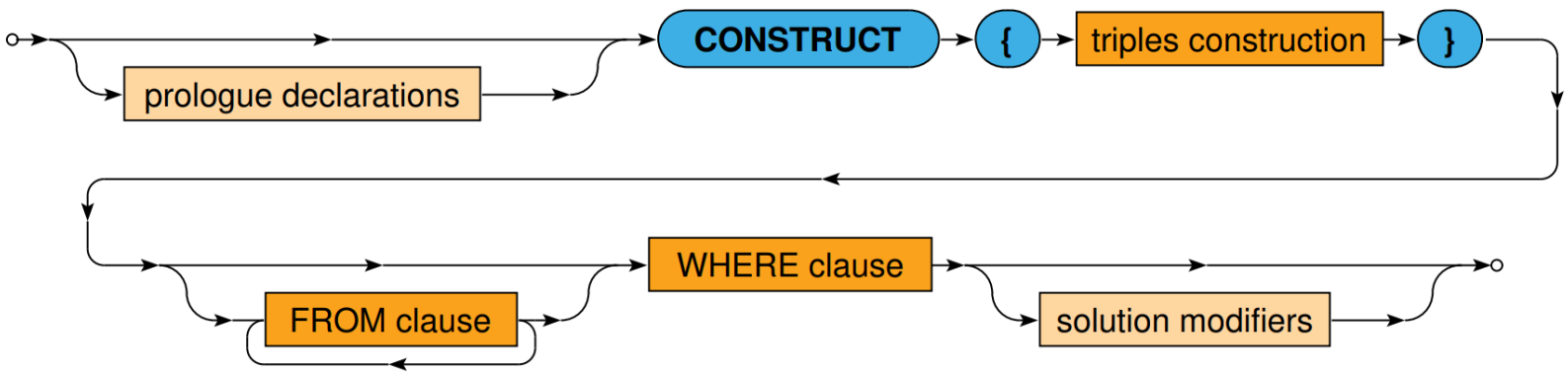


Result. Solution sequence = ordered multiset of solutions

Query Forms

CONSTRUCT query form

- Creates a **new graph** according to a provided pattern prologue declarations



Result. RDF graph constructed according to a group graph pattern

- Unbound or invalid triples are not involved

Query Forms: Example

CONSTRUCT

```
1  PREFIX i: <http://db.my/terms#>
2  CONSTRUCT
3    {
4      ?a i:name concat(?f, " ", ?l) .
5    }
6  FROM <http://db.my/actors>
7    WHERE
8    {
9      ?a rdf:type i:Actor ;
10       i:firstname ?f ;
11       i:lastname ?l .
12   }
```

```
1  <http://db.my/actors/harrelson> i:name "Woody Harrelson" .
2  <http://db.my/actors/eisenberg> i:name "Jesse Eisenberg" .
3  <http://db.my/actors/stone> i:name "Emma Stone" .
4  <http://db.my/actors/wasikowska> i:name "Mia Wasikowska" .
```

Lecture Conclusion

SPARQL

- **Query forms**
  - SELECT, ASK, DESCRIBE, CONSTRUCT
- **Graph patterns**
  - Basic, group, optional, alternative, minus
  - Variable assignments
  - Filters
- **Solution modifiers**
  - DISTINCT, REDUCED
  - GROUP BY, HAVING
  - ORDER BY, LIMIT, OFFSET