F21BD Big Data Management

RDF Triples Storage & SPARQL Query

Master Exam Preparation Set

Topics Covered:

- RDF Triples & Knowledge Representation
- Ontologies & Meta-languages
- Triple Stores & Storage Technologies
- JENA Framework & Fuseki Server
- SPARQL Query Language (Complete Coverage)
- Advanced SPARQL Features
- Reasoning & Inference

Prepared: February 13, 2026

Contents

1 RDF Fundamentals

1.1 What is a Triple? (ELI5 Explanation)

Simple Definition

A **Triple** is the basic building block of knowledge representation in RDF (Resource Description Framework).
Think of it like a simple sentence with 3 parts:

- **Subject:** Who or what we're talking about
- **Predicate:** The relationship or property
- **Object:** The value or another entity

Example: "Alice knows Bob"

- Subject: Alice
- Predicate: knows
- Object: Bob

1.2 RDF Triple Structure

Triple Format

Basic Structure:

<subject> <predicate> <object>

Also called: RDF Statements
Components:

1. **Subject:** Must be IRI (never literal)
2. **Predicate:** Must be IRI (describes relationship)
3. **Object:** Can be IRI, Literal, or Blank Node (bnode)

IRI (Internationalized Resource Identifier):

- A unique identifier for resources (like a web address)
- Example: `http://example.org/person/Alice`

Literal:

- A simple value (string, number, date)
- Example: "Alice", 28, "2026-02-13"

Blank Node (bnode):

- A node without an IRI (anonymous resource)
- Example: Used for temporary or intermediate entities

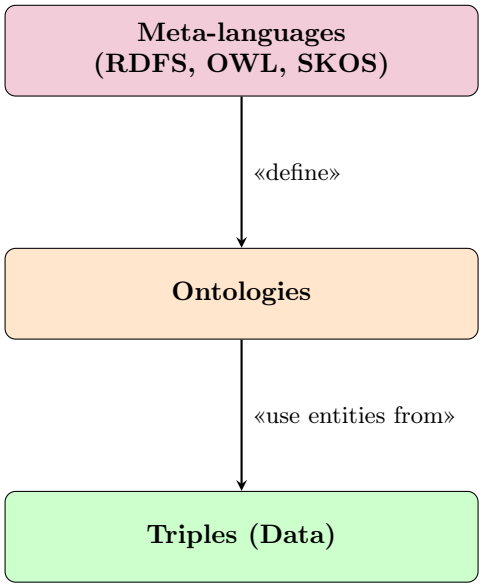
1.3 Ontologies & Meta-languages

Hierarchy of Knowledge Representation

From Top to Bottom:

1. **Meta-languages** (RDFS, OWL, SKOS)
   - Define vocabularies and constraints
   - Provide rules for creating ontologies
2. **Ontologies**
   - Define entities (classes, properties)
   - Expressed using meta-languages
   - Example: "Person", "knows", "hasAge"
3. **Triples** (Data)
   - Use entities from ontologies
   - Actual data instances
   - Example: Alice knows Bob

Terms:
- Vocabulary
- Constraints
- Meta-data



1.4 Creating RDF Triples - ETL Process

Extract

Creating triples manually is **slow and error-prone**. Automated tools convert data from various sources into RDF triples.
Process:

1. **Extract:** Get data from source (database, XML, CSV, text)
2. **Transform:** Map data to ontology entities
3. **Load:** Store as RDF triples in triplestore

Data Source	Conversion Tools
Relational Database	R2RML, triplify, D2RQ, ODEMapster, Datalift
XML	GRDDL, Xsparql
Excel, CSV	Google Refine, Any23, QuidiCRC, Lionel
Text	Dog4dag, Gate, Fred, OntoLing, LexOnt
Frameworks	Coeus, marimba, DataTank

Table 1: ETL Tools for RDF Triple Creation

2 Storing RDF Triples

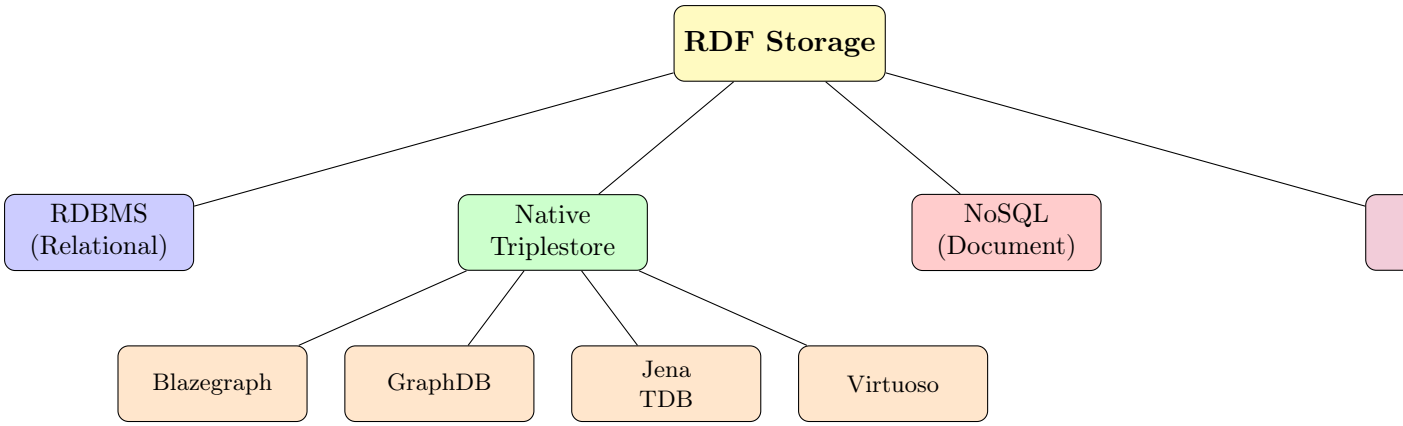
2.1 What is a Triplestore?

Triplestore Definition

A **Triplestore** (also called RDF Store) is a specialized database designed to store and retrieve RDF triples.
Key Features:

- Stores RDF data as graph structures
- Manages multiple datasets
- Each dataset can contain one or more graphs
- Supports SPARQL queries
- Allows inferencing through reasoners
- Can query across multiple graphs

2.2 RDF Storage Technologies



Type	Examples	Characteristics
RDBMS	PostgreSQL	Store triples in relational tables
Native Triplestore	Jena TDB, GraphDB	Optimized for RDF graph storage
NoSQL	MongoDB	Document-based storage
Cloud Solutions	Amazon Neptune, Stardog	Managed cloud services

Table 2: RDF Storage Options

3 JENA Framework

3.1 What is JENA?

JENA Framework Overview

JENA is a **Java framework** for building Semantic Web applications.
Provides:

- RDF API: Create and manipulate RDF graphs
- Ontology API: Work with OWL ontologies
- SPARQL API: Execute SPARQL queries
- Inference API: Reasoning over RDF data
- Storage API: Multiple storage backends

3.2 JENA Architecture

3.3 Fuseki Server

Fuseki - SPARQL Server

Fuseki is a **SPARQL server** built on JENA that exposes RDF datasets as web services.
Key Features:

- Based on Jetty (embedded web server + Java servlet container)
- Comes with TDB2 (native triplestore)
- Can run as: system service, standalone server, or web application

Installation:

- Can be installed as a system service
- Access via: `http://localhost:3030`

3.4 Running Fuseki (Commands)

```
# Check status
fuseki status

# Start server
fuseki start

# Stop server
fuseki stop

# Restart server
fuseki restart
```

Listing 1: Fuseki Terminal Commands (macOS)

Web Interface:

- Navigate to: `http://localhost:3030`
- Manage datasets
- Upload ontologies
- Execute SPARQL queries
- View results

3.5 Creating a Dataset in Fuseki

Exam Task: Dataset Creation Steps

Step 1: Click "add new dataset"
Step 2: Choose dataset type:

- In-memory: Lost on restart
- Persistent: Saved to disk
- Persistent (TDB2): Recommended

Step 3: Name your dataset (e.g., "periodTable")
Step 4: Click "create dataset"
Step 5: Upload ontology file:

- Click "upload data"
- Select file (e.g., PeriodicTable.owl)
- Click "upload now"

Step 6: Verify metrics:

- Check number of triples loaded
- Example: 1847 triples for Periodic Table

4 SPARQL Query Language

4.1 What is SPARQL? (ELI5)

Simple Definition

SPARQL = SQL for RDF graph data
Full Name: SPARQL Protocol and RDF Query Language
Think of it like:

- SQL queries tables → SPARQL queries graphs
- SQL uses FROM/WHERE → SPARQL uses graph patterns
- SQL returns rows → SPARQL returns bindings or graphs

Key Difference: SPARQL works with *triples and relationships*, not tables.

4.2 SPARQL Standard - Four Key Parts

SPARQL Components

1. **Query Language:** Retrieve data from RDF graphs
2. **Protocol:** Send queries to server, receive results over HTTP
3. **Query Results:** Format of returned data (XML, JSON, CSV, Turtle)
4. **Update Language:** Add, modify, or delete data in RDF graphs

4.3 SPARQL Endpoint

What is a SPARQL Endpoint?

A **SPARQL Endpoint** is a web service that:

- Implements the SPARQL protocol
- Accepts SPARQL queries over HTTP
- Returns results in standard formats
- Example: `http://localhost:3030/periodTable/query`

Think of it like: A REST API for querying RDF data

4.4 Principle of SPARQL Queries

Pattern Matching - Core Concept

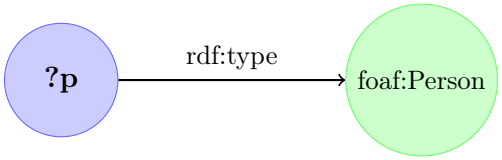
SPARQL is based on **pattern matching**.
How it works:

1. You describe a **pattern** (template) of triples you want
2. The SPARQL engine **searches the graph**
3. The results are the **parts of the graph that match** your pattern

Variables: Start with `?` or `$`

- Example: `?person`, `?name`, `$age`

Example Pattern:



Meaning: Find all `?p` (variable) that are of type `foaf:Person` (constant).

4.5 Four Types of SPARQL Queries

Query Type	Returns	Purpose
SELECT	Variables (table)	Retrieve specific values
CONSTRUCT	RDF graph	Build new triples from template
DESCRIBE	RDF graph	Get all information about resource
ASK	Boolean (true/false)	Check if pattern exists

Table 3: SPARQL Query Types

```
SELECT ?name ?age
WHERE {
    ?person foaf:name ?name .
    ?person foaf:age ?age .
}
```

Listing 2: SELECT Query Example

```
CONSTRUCT {
    ?person foaf:name ?name .
}
WHERE {
    ?person foaf:name ?name .
}
```

Listing 3: CONSTRUCT Query Example

```
DESCRIBE <http://example.org/person/123>
```

Listing 4: DESCRIBE Query Example

```
ASK
WHERE {
    ?person foaf:age 30 .
}
```

Listing 5: ASK Query Example

4.6 SPARQL Query Structure

Query Anatomy

```
PREFIX prefix1: <namespace1>
PREFIX prefix2: <namespace2>
...

QUERY_TYPE variables

FROM <graph_uri>

WHERE {
    # Graph pattern (triples)
    ?subject ?predicate ?object .
    FILTER (...)
    OPTIONAL {...}
}
GROUP BY ?variable
HAVING (...)
ORDER BY ?variable
LIMIT n
OFFSET m
```

Component Breakdown:

1. **PREFIX:** Define namespace shortcuts
   - Example: `PREFIX foaf: <http://xmlns.com/foaf/0.1/>`
   - Allows using `foaf:name` instead of full IRI
2. **QUERY_TYPE:** SELECT, CONSTRUCT, DESCRIBE, or ASK
3. **FROM:** (Optional) Specify which graph(s) to query
4. **WHERE:** Graph pattern to match (written in Turtle syntax)
5. **FILTER:** Apply conditions to restrict results
6. **OPTIONAL:** Match pattern if possible, but don't require it
7. **GROUP BY:** Group results by variable
8. **HAVING:** Filter groups (like SQL HAVING)
9. **ORDER BY:** Sort results
10. **LIMIT:** Maximum number of results
11. **OFFSET:** Skip first n results

5 Basic Graph Patterns in SPARQL

5.1 Triple Pattern Syntax

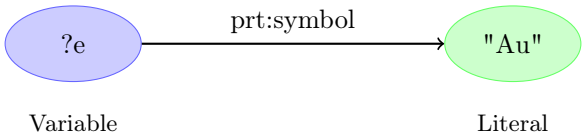
Most General Pattern

<subject> <predicate> <object> .

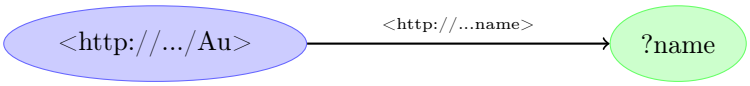
Important: Pattern ends with a **period** (.)
Each component (S, P, O) can be:

1. **Variable:** Starts with `?` or `$`
2. **Full IRI:** Complete URI in angle brackets
3. **QName:** Qualified name using prefix
4. **Literal:** String, number, or date value

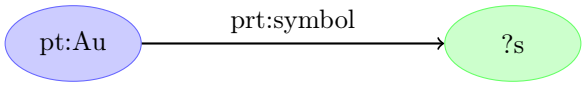
5.2 Triple Pattern Examples



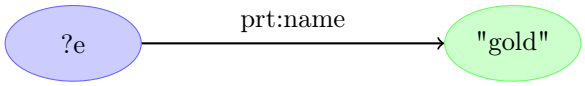
Variable Literal



Full IRI



QName



String Literal

5.3 Example 1: Simple SELECT Query

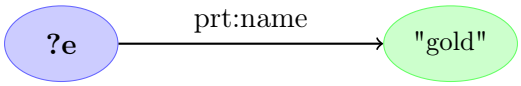
Goal: Select the IRI of the element named "gold"

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?e
WHERE {
    ?e prt:name "gold"^^xsd:string.
}
```

Listing 6: Query - Find Element by Name

Pattern Visualization:



Result:

e
prt:Au

5.4 Example 2: Multiple Triple Patterns

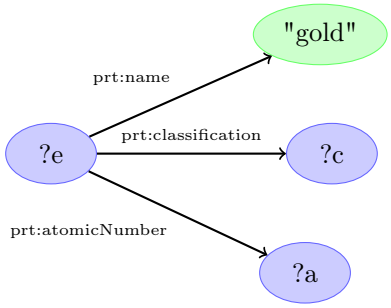
Goal: Get IRI, classification, and atomic number for element named "gold"

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?e ?c ?a
WHERE {
    ?e prt:name "gold".
    ?e prt:classification ?c.
    ?e prt:atomicNumber ?a.
}
```

Listing 7: Query with Multiple Patterns

Pattern Visualization:



Shorthand Syntax (using semicolon):

```
SELECT ?e ?c ?a
WHERE {
    ?e prt:name "gold";
       prt:classification ?c;
```

```
        prt:atomicNumber ?a.
}
```

Listing 8: Equivalent Query with Semicolon

Semicolon (;) means: Same subject, different predicate-object pairs
Result:

e	c	a
prt:Au	prt:Metallic	79^^xsd:integer

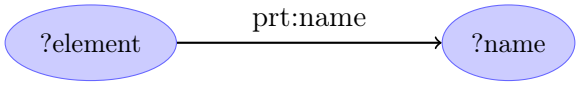
5.5 Example 3: SELECT All Element Names

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?name
WHERE {
    ?element prt:name ?name.
}
```

Listing 9: Simple Query - All Names

Pattern:



Result: List of all element names (Hydrogen, Helium, Lithium, ...)

5.6 Example 4: SELECT All Triples

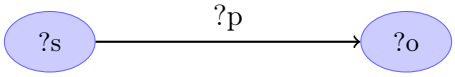
Most General Query: Retrieve everything

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT ?subject ?predicate ?object
WHERE {
    ?subject ?predicate ?object
}
```

Listing 10: Query All Triples

Pattern:



Exam Note

Use Case: This is the *typical first query* when exploring a new SPARQL endpoint to understand the data structure.
Warning: Can return HUGE result sets! Use LIMIT to restrict.

6 SPARQL OPTIONAL

6.1 What is OPTIONAL?

OPTIONAL Keyword

Purpose: Include optional patterns that might not match for all results.
Without OPTIONAL: If a pattern doesn't match, the entire result is excluded.
With OPTIONAL: If the optional pattern doesn't match, the result is still included (with empty/NULL value for optional variables).
Think of it like: SQL LEFT JOIN

6.2 Example: Elements With and Without Color

Without OPTIONAL (elements MUST have color):

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT *
WHERE {
    ?e prt:name ?n;
       prt:classification ?c;
       prt:atomicNumber ?a;
       prt:color ?color.
}
```

Listing 11: Query Requires Color

Result: Only elements WITH color (e.g., Chlorine, Bromine)
Elements WITHOUT color are excluded (e.g., elements 113, 115, 117)

With OPTIONAL (color is optional):

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT *
WHERE {
    ?e prt:name ?n;
       prt:symbol ?c;
       prt:atomicNumber ?a.
    OPTIONAL { ?e prt:color ?color. }
}
```

Listing 12: Query with Optional Color

Result: ALL 118 elements
Elements WITH color show the color value.
Elements WITHOUT color show empty/NULL for `?color`.

Exam Trap

Question: "Why do some elements not appear in results?"
Answer: Likely missing an OPTIONAL for a property that not all entities have. Add OPTIONAL to include all entities even if they lack that property.

7 SPARQL FILTER

7.1 What is FILTER?

FILTER Definition

FILTER restricts results by applying conditions (Boolean expressions).
Syntax:

```
FILTER ( condition )
```

Operators:

- Comparison: `>`, `<`, `>=`, `<=`, `=`, `!=`
- Logical: `&&` (AND), `||` (OR), `!` (NOT)
- Functions: `REGEX`, `STR`, `LANG`, `DATATYPE`

7.2 Filtering Numbers

Example 1: Elements with atomic number < 10

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?symbol ?number ?state
WHERE {
    ?element prt:symbol ?symbol;
             prt:atomicNumber ?number;
             prt:standardState ?state.
    FILTER(?number < 10)
}
```

Listing 13: Numeric Filter Example

Result: H, He, Li, Be, B, C, N, O, F (atomic numbers 1-9)

Example 2: Date range filter

```
PREFIX pep: <http://hw.ac.uk/#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT *
WHERE {
    ?sub pep:bornOn ?date.
    FILTER(?date < "2020" && ?date > "1962")
}
```

Listing 14: Date Range Filter

Condition: Born between 1962 and 2020

7.3 Filtering Strings (Regular Expressions)

REGEX Function

Syntax:

```
FILTER REGEX(?variable, 'pattern', 'flags')
```

Common Flags:

- `'i'`: Case insensitive

XPath Regular Expression Syntax:

- `^`: Start of string
- `$`: End of string
- `.`: Any character
- `*`: Zero or more
- `+`: One or more
- `[abc]`: Character class

Reference: `https://www.w3.org/TR/xpath-functions/#regex-syntax`

Example: Elements starting with 'N' (case insensitive)

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?symbol ?number ?state
WHERE {
    ?element prt:symbol ?symbol;
             prt:atomicNumber ?number;
             prt:standardState ?state.
    FILTER REGEX(?symbol, '^n', 'i')
}
```

Listing 15: String Filter with REGEX

Pattern Breakdown:

- `^n`: Starts with 'n' or 'N'
- `'i'`: Case insensitive flag

Result: N (Nitrogen), Na (Sodium), Nb (Niobium), Nd (Neodymium), Ne (Neon), Ni (Nickel), No (Nobelium), Np (Neptunium)

7.4 Filtering with Existence Checks

FILTER EXISTS and FILTER NOT EXISTS

Check if pattern exists:

```
FILTER EXISTS { pattern }
```

Check if pattern does NOT exist:

```
FILTER NOT EXISTS { pattern }
```

Example: Elements WITHOUT color

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT *
WHERE {
    ?element prt:name ?name;
             prt:symbol ?symbol;
             prt:atomicNumber ?number.
    FILTER NOT EXISTS { ?element prt:color ?color. }
}
```

Listing 16: Filter for Non-Existing Property

Result: Elements 113, 115, 117 (which have no color property)

Alternative: MINUS Keyword

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT *
WHERE {
    ?element prt:name ?name;
             prt:symbol ?symbol;
             prt:atomicNumber ?number.
    MINUS { ?element prt:color ?color. }
}
```

Listing 17: Equivalent using MINUS

MINUS: Removes matches from the result set (set difference operation)

Exam Comparison

FILTER NOT EXISTS vs MINUS:

- Both remove results that match a pattern
- FILTER NOT EXISTS: More flexible, can use in subqueries
- MINUS: Simpler syntax, set-based operation
- Generally interchangeable for simple cases

8 SPARQL UNION

8.1 What is UNION?

UNION Operator

Purpose: Combine results from multiple patterns (disjunction).
Think of it like: SQL UNION
Syntax:

```
WHERE {
    { pattern1 }
    UNION
    { pattern2 }
    UNION
    { pattern3 }
}
```

Result: Union of all triples that match pattern1 OR pattern2 OR pattern3

8.2 Example: Elements from Group 1 or Group 4

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?element
WHERE {
    {
        ?element prt:symbol ?symbol;
                 prt:atomicNumber ?number;
                 prt:group prt:group_1.
    }
    UNION
    {
        ?element prt:symbol ?symbol;
                 prt:atomicNumber ?number;
                 prt:group prt:group_4.
    }
}
```

Listing 18: UNION Query Example

Result: Elements from group 1 (H, Li, Na, K, Rb, Cs, Fr) AND elements from group 4 (Ti, Zr, Hf, Rf)

Exam Note

Question Type: "Select entities that satisfy condition A OR condition B"
Answer: Use UNION with separate patterns for A and B.

9 SPARQL Query Modifiers

9.1 Overview of Modifiers

Modifier	Purpose
ORDER BY	Sort results by variable value(s)
LIMIT	Restrict number of results returned
OFFSET	Skip first n results
GROUP BY	Group results by variable value
HAVING	Filter groups (after GROUP BY)

Table 4: SPARQL Query Modifiers

9.2 ORDER BY

ORDER BY Syntax

Ascending Order (default):

```
ORDER BY ?variable
```

Descending Order:

```
ORDER BY DESC(?variable)
```

Multiple Variables:

```
ORDER BY ?var1 DESC(?var2)
```

Example: Sort by atomic number (ascending)

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?name ?number
WHERE {
    ?element prt:name ?name;
             prt:atomicNumber ?number;
             prt:group prt:group_1.
}
ORDER BY ?number
```

Listing 19: ORDER BY Ascending

Result: Elements ordered 1, 2, 3, 4, ... (H, He, Li, Be, ...)

Example: Sort by atomic weight (descending)

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?name ?number
WHERE {
    ?element prt:name ?name;
             prt:atomicNumber ?number;
             prt:group prt:group_1.
}
ORDER BY DESC(?number)
```

Listing 20: ORDER BY Descending

Result: Elements ordered 118, 117, 116, ... (Og, Ts, Lv, ...)

9.3 LIMIT and OFFSET

LIMIT and OFFSET

LIMIT n: Return at most n results
OFFSET m: Skip first m results
Combined:

```
LIMIT 10
OFFSET 20
```

Means: Skip first 20 results, then return next 10 (results 21-30)

Example: Top 5 heaviest elements

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?name
WHERE {
    ?element prt:name ?name;
             prt:atomicWeight ?weight.
}
ORDER BY DESC(?weight)
LIMIT 5
```

Listing 21: LIMIT with ORDER BY

Result: 5 elements with highest atomic weight

Example: Pagination (skip 10, show next 5)

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?name
WHERE {
    ?element prt:name ?name;
             prt:atomicWeight ?weight.
}
ORDER BY DESC(?weight)
LIMIT 5
OFFSET 10
```

Listing 22: OFFSET and LIMIT for Pagination

Result: Elements ranked 11-15 by weight

Exam Use Case

Pagination Pattern:

- Page 1: LIMIT 10 OFFSET 0
- Page 2: LIMIT 10 OFFSET 10
- Page 3: LIMIT 10 OFFSET 20
- Page n: LIMIT 10 OFFSET (n-1)*10

9.4 GROUP BY

GROUP BY Syntax

Purpose: Create groups based on variable value, then apply aggregate functions.
Syntax:

```
GROUP BY ?variable
```

Aggregate Functions:

- `COUNT(?var)`: Count items in group
- `SUM(?var)`: Sum of numeric values
- `AVG(?var)`: Average of numeric values
- `MIN(?var)`: Minimum value
- `MAX(?var)`: Maximum value
- `SAMPLE(?var)`: Random item from group
- `GROUP_CONCAT(?var)`: Concatenate values with separator

Example: Count elements per state

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?state (COUNT(?symbol) AS ?num)
WHERE {
    ?element prt:name ?symbol;
             prt:standardState ?state;
             prt:atomicNumber ?number.
}
GROUP BY ?state
ORDER BY ?num
```

Listing 23: GROUP BY with COUNT

Result:

state	num
prt:liquid	3
prt:state_unknown	3
prt:gas	12
prt:solid	100

Explanation:

- 3 liquid elements (Br, Hg, ...)
- 3 unknown state
- 12 gases (H, He, N, O, ...)
- 100 solids (most metals)

Example: GROUP_CONCAT (collapse multiple rows)

```
SELECT ?name (GROUP_CONCAT(?pet; separator=", ") AS ?pets)
WHERE {
    ?person foaf:name ?name;
            ex:hasPet ?pet.
}
GROUP BY ?name
```

Listing 24: GROUP_CONCAT Example

Before GROUP BY:

name	pet
Henry	Piglet
Lisa	Snowball
Lisa	Snowball II
Madeline	Kirby
Madeline	Quigley

After GROUP BY with GROUP_CONCAT:

name	pets
Henry	Piglet
Lisa	Snowball, Snowball II
Madeline	Kirby, Quigley

9.5 HAVING

HAVING Clause

Purpose: Filter groups AFTER they've been created (like SQL HAVING)
Difference from FILTER:

- FILTER: Applied to individual rows BEFORE grouping
- HAVING: Applied to groups AFTER grouping

Syntax:

```
GROUP BY ?var
HAVING (condition on aggregate)
```

Example: States with more than 10 elements

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

SELECT ?state (COUNT(?symbol) AS ?num)
WHERE {
    ?element prt:name ?symbol;
             prt:standardState ?state;
             prt:atomicNumber ?number.
}
GROUP BY ?state
HAVING (?num > 10)
ORDER BY ?num
```

Listing 25: HAVING Example

Result:

state	num
prt:gas	12
prt:solid	100

Explanation: Only groups with count > 10 are shown (liquid and unknown are excluded)

Exam Question Type

"Find categories with more than X items"
Solution Pattern:

1. GROUP BY category
2. COUNT items
3. HAVING (count $>$ X)

10 Advanced SPARQL Features

10.1 SPARQL 1.1 New Features (2013)

Key Enhancements in SPARQL 1.1

1. **Negation:** FILTER NOT EXISTS, MINUS
2. **Transitive Queries:** Property paths (e.g., `foaf:knows+`)
3. **Aggregation:** COUNT, SUM, AVG, MIN, MAX, GROUP_CONCAT
4. **Subqueries:** Nested SELECT queries
5. **Property Paths:** Navigate graphs with path expressions

10.2 Property Paths (Transitive Queries)

Property Path Operators

Operators:

- `+`: One or more (transitive closure)
- `*`: Zero or more
- `?`: Zero or one (optional)
- `/`: Sequence (path concatenation)
- `|`: Alternative (or)
- `^`: Inverse property

Reference: `https://www.w3.org/TR/sparql11-query/#propertypaths`

Example: Friends of friends (transitive)

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?person2
WHERE {
    ?person1 foaf:knows+ ?person2.
}
```

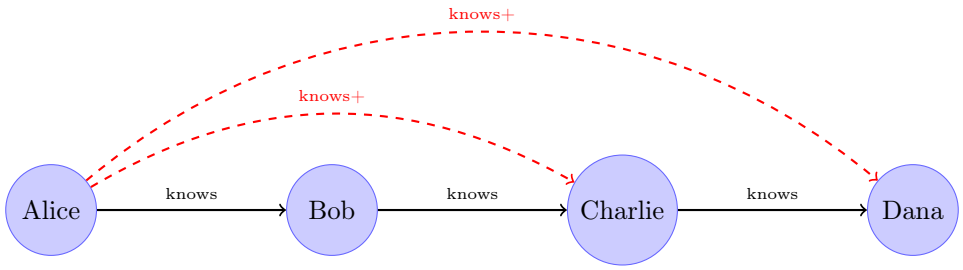
Listing 26: Transitive Property Path

Data:

- Alice knows Bob
- Bob knows Charlie
- Charlie knows Dana

Query: Find all connections from Alice (one or more hops)
Result: Bob (direct), Charlie (2 hops), Dana (3 hops)

Visualization:


knows+
knows+
Alice
knows
Bob
knows
Charlie
knows
Dana

10.3 Subqueries

Subquery Definition

A **subquery** is a nested SELECT query inside the WHERE clause.
Use Cases:

- Filter based on aggregate values
- Pre-compute intermediate results
- Compare individual values to aggregates

Example: People older than average age

```
SELECT ?person
WHERE {
    ?person foaf:name ?name.
    ?person foaf:age ?age.
    FILTER (?age > (
        SELECT (AVG(?age) AS ?avg_age)
        WHERE {
            ?person foaf:age ?age.
        }
    ))
}
```

Listing 27: Subquery Example

How it works:

1. Inner query calculates average age across all people
2. Outer query filters people whose age $>$ average

Result: All people with age above the average

11 CONSTRUCT, DESCRIBE, and ASK

11.1 CONSTRUCT Query

CONSTRUCT Definition

Purpose: Create NEW RDF triples from query results using a template.
Returns: RDF graph (not a table)
Syntax:

```
CONSTRUCT {
    # Template triples with variables
}
WHERE {
    # Pattern to match
}
```

Example: Generate rdfs:label from name and rdfs:comment from color

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

CONSTRUCT {
    ?element rdfs:label ?name;
             rdfs:comment ?color.
}
WHERE {
    ?element prt:name ?name;
             prt:symbol ?symbol;
             prt:atomicNumber ?number.
    OPTIONAL { ?element prt:color ?color. }
}
```

Listing 28: CONSTRUCT Example

Output (Turtle format):

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>.

# Hydrogen (no color)
prt:H rdfs:label "Hydrogen".

# Helium (no color)
prt:He rdfs:label "Helium".

# Chlorine (has color)
prt:Cl rdfs:label "Chlorine";
       rdfs:comment "greenish yellow".

# Bromine (has color)
prt:Br rdfs:label "Bromine";
       rdfs:comment "reddish-brown".
```

Listing 29: Generated Triples

Use Case: Transform data structure, create derived datasets, data integration

11.2 DESCRIBE Query

DESCRIBE Definition

Purpose: Retrieve ALL properties of a resource (or resources matching a pattern).
Returns: RDF graph with all triples about the resource
Syntax:

```
DESCRIBE <resource_iri>
```

Or with pattern:

```
DESCRIBE ?variable
WHERE {
    # Pattern to match resources
}
```

Example 1: Describe specific element

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

DESCRIBE prt:Au
```

Listing 30: DESCRIBE with IRI

Output: ALL triples where `prt:Au` is the subject:

```
prt:Au rdf:type prt:Element;
       prt:atomicNumber 79;
       prt:atomicWeight "196.96655"^^xsd:float;
       prt:block prt:d-block;
       prt:casRegistryID "7440-57-5";
       prt:classification prt:Metallic;
       prt:color "gold";
       prt:group prt:group_11;
       prt:name "gold";
       prt:period prt:period_6;
       prt:standardState prt:solid;
       prt:symbol "Au".
```

Listing 31: DESCRIBE Output

Example 2: Describe all elements

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

DESCRIBE ?e
WHERE {
    ?e a prt:Element
}
```

Listing 32: DESCRIBE with WHERE

Output: All triples for every element (complete dump of element data)

Exam Note

Use Case: DESCRIBE is useful for:

- Exploring unknown datasets

- Exporting complete entity data
- Data migration

Warning: Can produce HUGE results if many resources match!

11.3 ASK Query

ASK Definition

Purpose: Check if a pattern exists in the graph.
Returns: Boolean (true or false)

- **true:** At least one match found
- **false:** No matches found

Syntax:

```
ASK {
    # Pattern to test
}
```

Example: Check if any element symbol contains 'A'

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

ASK {
    ?e a prt:Element;
       prt:symbol ?s.
    FILTER REGEX(?s, 'A', 'i')
}
```

Listing 33: ASK Query Example

Result: `true` (because Ag, Al, Ar, As, At, Au all contain 'A')

Example: Check if element 119 exists

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/PeriodicTable#>

ASK {
    ?e prt:atomicNumber 119.
}
```

Listing 34: ASK for Existence Check

Result: `false` (element 119 not yet discovered/added)

Exam Use Case

Question: "Does the dataset contain any X?"
Answer Pattern:

```
ASK {
    ?entity property value.
    # Add filters if needed
}
```

Common Use: Validation, data quality checks, conditional logic in applications

12 SERVICE - Federated Queries

12.1 What is SERVICE?

SERVICE Keyword

Purpose: Query a **remote SPARQL endpoint** from within your query.
Use Case: Combine data from multiple sources (federated queries)
Syntax:

```
SERVICE <endpoint_url> {
    # Pattern to send to remote endpoint
}
```

Think of it like: Distributed JOIN across multiple databases

12.2 Example: Query DBpedia from Local Fuseki

Goal: Find people in DBpedia whose name starts with "Hadj"

```
PREFIX dbpedia: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name ?birth_date
WHERE {
    SERVICE <http://dbpedia.org/sparql> {
        ?person a foaf:Person;
                foaf:name ?name;
                dbpedia:birthDate ?birth_date.
        FILTER REGEX(?name, '^Hadj', 'i')
    }
}
```

Listing 35: SERVICE Query to DBpedia

How it works:

1. Query sent to `http://dbpedia.org/sparql`
2. DBpedia executes the pattern
3. Results returned to your local query
4. Can combine with local data

Result: Names and birthdates from DBpedia

Example: Combine Local and Remote Data

```
SELECT *
WHERE {
    # Local data
    {
        ?person prt:name ?localName;
                prt:symbol ?symbol.
    }

    # Remote data from DBpedia
    SERVICE <http://dbpedia.org/sparql> {
        ?person foaf:name ?remoteName;
```

```
                    dbpedia:birthDate ?birthDate.
    }
}
```

Listing 36: Federated Query

Use Case: Enrich local data with external knowledge bases

Exam Note

Limitations:

- Dependent on remote endpoint availability
- Network latency can slow queries
- Remote endpoint might have query limits
- Some endpoints require authentication

Multiple SERVICE: Can use multiple SERVICE clauses to query different endpoints

13 Reasoning within Triplestores

13.1 What is Reasoning?

Reasoning Definition

Reasoning (or **Inference**) is the process of deriving NEW triples from EXISTING triples using logical rules.
Why?

- Make implicit knowledge explicit
- Enable smarter queries
- Reduce data redundancy

Example:

- Rule: "If X is a Dog, then X is a Mammal"
- Fact: "Buddy is a Dog"
- Inference: "Buddy is a Mammal" (automatically derived)

13.2 Types of Reasoning Rules

Rule Type	Description
RDFS subClassOf	If A subClassOf B, then all instances of A are also instances of B
RDFS subPropertyOf	If P1 subPropertyOf P2, then (X P1 Y) implies (X P2 Y)
OWL sameAs	If X sameAs Y, then all statements about X also apply to Y
OWL TransitiveProperty	If P is transitive, (X P Y) and (Y P Z) implies (X P Z)
Custom Rules	User-defined inference rules

Table 5: Common Reasoning Rule Types

Example: RDFS subClassOf
Ontology:

```
:Fruit rdfs:subClassOf :HealthyFood.
:Vegetable rdfs:subClassOf :HealthyFood.
```

Data:

```
:apple a :Fruit.
:carrot a :Vegetable.
```

Inferred (automatically):

```
:apple a :HealthyFood.    # Because Fruit subClassOf HealthyFood
:carrot a :HealthyFood.   # Because Vegetable subClassOf HealthyFood
```

13.3 Forward vs Backward Chaining

Two Reasoning Strategies

Forward Chaining: Data-driven (facts → conclusions)

- Start with known facts
- Apply all applicable rules
- Generate ALL inferred triples upfront
- Store inferred triples with original data
- **Pro:** Query time is fast (inference done once)
- **Con:** Slow initial loading (many inferences)

Backward Chaining: Goal-driven (conclusions → facts)

- Start with query goal
- Work backwards to find supporting facts
- Infer triples only when needed (during query)
- Don't store inferred triples
- **Pro:** Fast loading (no upfront inference)
- **Con:** Slower queries (inference per query)

13.4 Example: Rain, Ground, Grass

Rules:

1. If it is raining, then the ground is wet.
2. If the ground is wet, then the grass is wet.

Fact: It is raining.

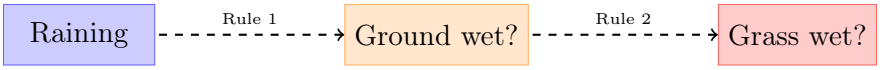
Forward Chaining (facts → conclusions):

1. **Start:** It is raining (known fact)
2. **Apply Rule 1:** Raining → Ground is wet
3. **Apply Rule 2:** Ground wet → Grass is wet
4. **Result:** All inferences stored

Backward Chaining (conclusions → facts):
Goal: Prove "The grass is wet"

1. **Start:** Want to prove "Grass is wet"

2. **Apply Rule 2 backwards:** Need to prove "Ground is wet"
3. **Apply Rule 1 backwards:** Need to prove "It is raining"
4. **Fact found:** "It is raining"
5. **Result:** Goal proven (grass IS wet)



Exam Comparison

Forward vs Backward:

14 Exam Strategy & Quick Reference

14.1 High-Probability Exam Questions

Question Type 1: Write SPARQL Query

Example: "Write a SPARQL query to find all elements with atomic number less than 20"
Answer Template:

```
PREFIX prt: <...>

SELECT ?element ?name
WHERE {
    ?element prt:name ?name;
             prt:atomicNumber ?num.
    FILTER(?num < 20)
}
```

Steps:

1. Define PREFIX
2. Identify variables to SELECT
3. Write triple patterns (subject-predicate-object)
4. Add FILTER conditions
5. Add ORDER BY/LIMIT if needed

Question Type 2: Explain Query Result

Example: "What does this query return?"
Answer Template:

1. Identify query type (SELECT/CONSTRUCT/ASK/DESCRIBE)
2. Explain the pattern being matched
3. Describe any filters or conditions
4. State what variables/triples are returned

Question Type 3: Compare Query Approaches

Example: "What's the difference between OPTIONAL and FILTER?"
Answer:

- **OPTIONAL:** Includes results even if pattern doesn't match (like LEFT JOIN)
- **FILTER:** Excludes results that don't meet condition (WHERE clause)

14.2 One-Page SPARQL Cheat Sheet

SPARQL Quick Reference

Query Structure:

```
PREFIX ns: <uri>
SELECT/CONSTRUCT/DESCRIBE/ASK
WHERE {
    ?s ?p ?o .
    FILTER(...)
    OPTIONAL {...}
}
GROUP BY ?var
HAVING(...)
ORDER BY ?var
LIMIT n OFFSET m
```

Query Types:

- SELECT: Returns variables (table)
- CONSTRUCT: Returns RDF graph
- DESCRIBE: Returns all triples about resource
- ASK: Returns true/false

Operators:

- `OPTIONAL`: Include pattern if it matches, don't require
- `FILTER`: Apply condition
- `UNION`: Combine patterns (OR)
- `MINUS`: Exclude pattern

FILTER Functions:

- `REGEX(?var, 'pattern', 'i')`: String matching
- `?num > 10`: Numeric comparison
- `FILTER EXISTS`: Check pattern exists
- `FILTER NOT EXISTS`: Check pattern doesn't exist

Aggregate Functions:

- COUNT, SUM, AVG, MIN, MAX, SAMPLE, GROUP_CONCAT

Property Paths:

- `+`: One or more
- `*`: Zero or more
- `?`: Zero or one
- `/`: Sequence
- `|`: Alternative

Triple Pattern Syntax:

- `?var`: Variable
- `<http://...>`: Full IRI
- `prefix:name`: QName
- `"value"`: Literal
- `;`: Same subject
- `,`: Same subject and predicate

14.3 Common Exam Mistakes

1. **Forgetting the period (.) at end of triple**
   - Wrong: `?s ?p ?o`
   - Right: `?s ?p ?o .`
2. **Not defining PREFIX**
   - Must declare all prefixes used in query
3. **Confusing OPTIONAL and FILTER**
   - OPTIONAL: Include if exists, don't require
   - FILTER: Exclude if condition false
4. **Using GROUP BY without aggregate**
   - GROUP BY requires COUNT/SUM/AVG/etc.
5. **Wrong variable scope in FILTER**
   - FILTER can only reference variables from WHERE patterns above it
6. **Forgetting OPTIONAL for missing properties**
   - If property might not exist, use OPTIONAL

14.4 Exam Answer Checklist

Before Submitting SPARQL Query:

1. All PREFIX declarations present
2. Query type specified (SELECT/CONSTRUCT/etc.)
3. All triple patterns end with period (.)
4. All variables start with ? or $
5. FILTER syntax correct
6. Closing braces match opening braces
7. Optional patterns use OPTIONAL
8. GROUP BY used with aggregates

14.5 Practice Questions with Model Answers

Practice Question 1

Question: Write a SPARQL query to find the names and atomic numbers of all elements in group 1, ordered by atomic number.
Model Answer:

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/
   PeriodicTable#>

SELECT ?name ?number
WHERE {
    ?element prt:name ?name;
             prt:atomicNumber ?number;
             prt:group prt:group_1.
}
ORDER BY ?number
```

Practice Question 2

Question: Explain the difference between these two queries:
Query A:

```
SELECT ?e WHERE { ?e prt:color ?c. }
```

Query B:

```
SELECT ?e WHERE { ?e prt:name ?n. OPTIONAL { ?e prt:color ?c. } }
```

Model Answer:

- **Query A:** Returns only elements that HAVE a color property. Elements without color are excluded.
- **Query B:** Returns ALL elements. For elements with color, the `?c` variable is bound. For elements without color, `?c` is unbound/NULL.
- **Key Difference:** OPTIONAL makes the color pattern optional, so elements without color are still included.

Practice Question 3

Question: Write a query to find the number of elements in each classification, but only show classifications with more than 5 elements.
Model Answer:

```
PREFIX prt: <http://www.daml.org/2003/01/periodictable/
   PeriodicTable#>

SELECT ?classification (COUNT(?element) AS ?count)
WHERE {
    ?element prt:classification ?classification.
}
GROUP BY ?classification
HAVING (?count > 5)
ORDER BY DESC(?count)
```

15 Final Exam Checklist

15.1 Topics to Master

1. Understand RDF triple structure (subject-predicate-object)
2. Know difference between ontologies and data
3. Understand triplestore concept
4. SPARQL query structure (PREFIX, SELECT, WHERE)
5. Four query types: SELECT, CONSTRUCT, DESCRIBE, ASK
6. Triple pattern syntax (variables, IRIs, literals)
7. OPTIONAL keyword and use cases
8. FILTER with numeric, string, existence conditions
9. REGEX for string filtering
10. UNION for combining patterns
11. Query modifiers: ORDER BY, LIMIT, OFFSET
12. GROUP BY with aggregate functions
13. HAVING for filtering groups
14. Property paths (transitive queries)
15. Subqueries
16. SERVICE for federated queries
17. Reasoning concepts (forward vs backward chaining)

15.2 Quick Mental Checks

Can you answer instantly?

- What are the 3 parts of a triple? (Subject, Predicate, Object)
- What's the difference between IRI and Literal? (IRI identifies resource, Literal is value)
- What does OPTIONAL do? (Includes results even if pattern doesn't match)
- What's FILTER for? (Restrict results with conditions)
- GROUP BY requires what? (Aggregate functions like COUNT, SUM)
- What does ASK return? (Boolean: true or false)
- Forward chaining timing? (At load time)
- Backward chaining timing? (At query time)

Good Luck on Your Exam!

Master SPARQL query writing, understand graph patterns, and practice with real datasets. You're ready!