SDM Joint Project - Graph Development

Jule Grigat, Jorge Ignacio Del Río Sánchez, Charlotte Garcia

June 2025

M1 Purpose Statement

In our project Budget Bites, we introduce graph database technology to semantically integrate recipes, ingredients, and supermarket product data. The project was not performed via GitHub, as the required files exceed the platforms File Storage maximum, but the collection of scripts and some output files can be found in this repository `https://github.com/JuleGri/BudgetBites_GraphDB.git`.
Our main data sources include TheMealDB for recipes and ingredients (in English), as well as product catalogs from Mercadona and Carrefour (in Spanish), both of which share common structured fields like product ID, name, category, packaging, and price. The challenge lies in matching and enriching this heterogeneous data across different languages, formats, and levels of abstraction. To address this, we propose building a knowledge graph that connects recipes to their ingredients, maps those ingredients to translated and standardized forms, and links them to specific supermarket products. Products are further related to supermarkets, categories, and price metadata.
This graph-based approach offers significant advantages: it allows us to model the semantic relationships between items (e.g., ingredient–product matches, category hierarchies), perform flexible graph queries (e.g., "Which supermarket offers the cheapest seasonal basket for a recipe?"), and future-proof our system for further enrichment (e.g., adding additional supermarket chains or sustainability metadata). The graph thus acts as both a semantic integration layer and an analytical foundation, enabling improved recommendations and transparency.

M2 Justification for Knowlegdegraphs

For our project Budget Bites, we have deliberately chosen to implement a knowledge graph rather than a property graph, as it best suits our use case requirements. The core challenge lies in integrating heterogeneous data sources — recipes, multilingual ingredient lists, and structured supermarket product catalogs — while maintaining semantic consistency across different formats, languages, and abstraction levels. A knowledge graph enables us to explicitly model entities like Recipe, Ingredient, Product, and Supermarket as well-defined classes, and to connect them through semantic relationships such as usesIngredient, isSeasonalIn, and matchesWithProduct. This is essential for mapping abstract ingredient terms to real-world supermarket products in a way that is both flexible and extensible. Extensibility of the database using an existing Global Schema, is another core benefit of using a knowledgegraph over a property graph, which will be further

elaborated on in the respective section ”Added Value of the genera-purpose Knowledgegraph Approach” in M5.
By choosing RDF and Turtle syntax (as used in the TBox definition in the following section), we can use existing semantic web tools and the features we studied in class.
This entire setup works especially well for our project, since things like semantic similarity, translation matching, and adding extra info like seasonality or sustainability tags are really important. A property graph just wouldn't give us the same flexibility or semantic precision. Therefore, we generate a knowledge graph architecture using RDF, OWL, and SPARQL to represent and query our data.

If there was more time and experience with knowledgegraphs in general, this would be a great opportunity to deepdive into the opportunities that a knowledgegraph could provide for our product specifically, but in the given setting, this COurse Project will just serve as a general analysis of the knowledgegraph approach.

M3 Knowledgegraph Design/Schema

The Budget Bites knowledge graph models six core classes: Recipe, Ingredient, Product, Supermarket, Month, and Category. These are connected through key object properties such as usesIngredient, matchesWithProduct, isSeasonalIn, soldBy, and hasCategory.
The schema, illustrated in Figure 1 using the requested bubble-and-arrow metaphor, depicts the layered connection of recipes to supermarket products via standardized ingredients. This lets us enrich each ingredient node with translations, seasonality (linked to a Month node), and product matches from multiple supermarkets. Every product carries metadata like price, packaging, and availability, not shown in detail in the graph for simplifiction of the depiction, that are supporting analytical queries. The created ontology uses both RDFS and OWL. RDFS is used to specify domains and ranges of properties, enabling clear semantic typing of relationships, whereas OWL is used to define formal classes (owl:Class) and distinguish between object and datatype properties. The use of OWL additionally supports all sorts of future extensions such as property symmetry or transitivity for a deeper layer of ingredient information etc. Our approach is fairly simple in class structure to keep the model easy to overview and extensible, and yet it is semantically rich enough, to perform the necessary queryies for our use case.

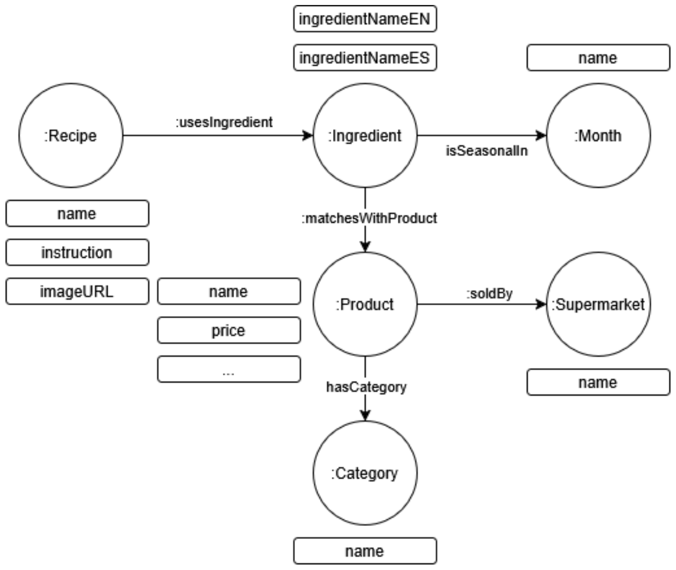


Figure 1: Bubbles and Arrows Diagram

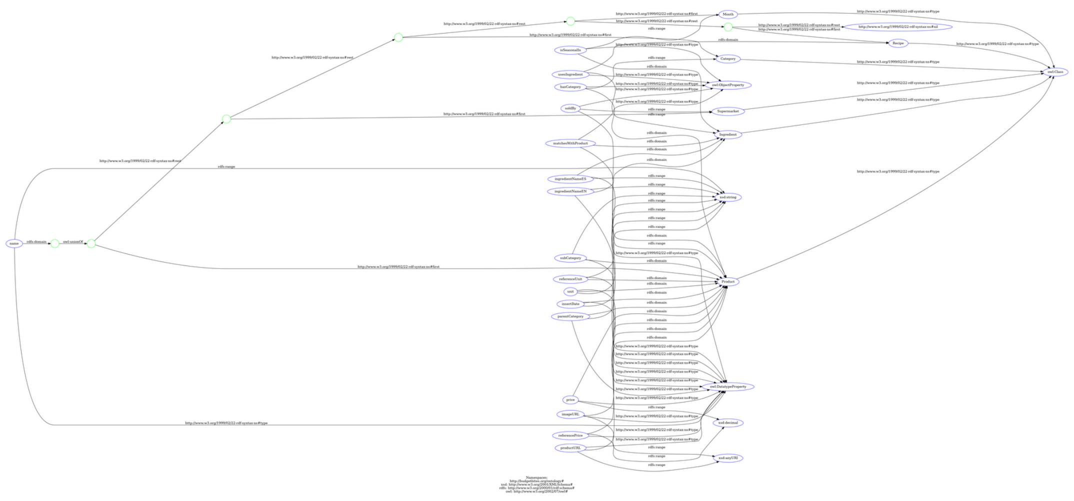


Figure 2: TBox SChema

The TBox Schema uses rdf and rdfs types, as well as enriching owl functions, as listed in the table 1.

Class	Count
bb:Product	534077
bb:Ingredient	575
rdf:Property	32
bb:Recipe	25
owl:DatatypeProperty	12
bb:Month	12
owl:Class	6
owl:ObjectProperty	5
owl:TransitiveProperty	4
owl:SymmetricProperty	4
rdfs:Class	3
rdfs:Datatype	3
bb:Supermarket	2
rdf:List	1
rdfs:ContainerMembershipProperty	1

Table 1: Semantic classes and property counts from the generated Knowledge Graph

M4. Populating the Knowledge Graph

To populate the knowledge graph for **Budget Bites**, raw data is drawn from several heterogeneous data sources and converted into RDF triples (ABox) through a semi-automated pipeline built in Python.

Data Sources and Structure

The core dataset is composed of three main sources:

- **Recipe and Ingredient data** retrieved from the MealDB API,
- **Product data** from supermarket chains Mercadona and Carrefour (CSV format),
- **Seasonality information** from a curated `ingredient_seasonality.json` file.

Thanks to the flexibility of knowledge graphs, this setup is a good foundation for potential future expansions. For example, the addition of new supermarkets or seasonal data sources can be integrated seamlessly by appending a new CSV or JSON file, without changes to the core graph structure.

Semi-Automated Transformation with Python Scripts

The transformation of source data into RDF triples is currently handled by a single, consolidated Python script (abox_generation_BudgetBites.py), included in the project folder. This script - at this prototype stage - performs all necessary parsing, mapping, and serialization steps to generate the full ABox for Budget Bites. It is written using the `rdflib` library and directly processes multilingual ingredient labels, product data from multiple supermarkets, seasonality data, and recipes

that were beforehand retrieved from the MealDB API.
The script initializes the RDF graph and binds the custom namespace `http://budgetbites.org/ontology#`. It defines static TBox-level elements like supermarkets (e.g., `Mercadona`, `Carrefour`), loads multilingual ingredient translations from a JSON file, and processes product data from two CSV files to generate `ex:Product` instances with metadata such as price, packaging, and category. Seasonal availability is added by linking ingredients to months via `ex:isSeasonalIn`, while recipes from a MealDB export are parsed into `ex:Recipe` instances with instructions, images, and ingredient links. Finally, the full RDF graph is serialized as a Turtle file for downstream use.

All of these steps are executed automatically when the script is run, requiring no manual editing or interaction beyond placing the input files in predefined directories (e.g., `landing_zone/`) by running the prepared landing_zone scripts from Project Part 1. Although automatic uploading to GraphDB is not currently enabled in the prototype phase, the script architecture supports this extension. A future module could connect to a local GraphDB instance (e.g., `http://localhost:7200`) and insert the generated Turtle file into the triple store via HTTP. This would allow end-to-end automation of the graph population process.

Potential blockers for such integration could be configuration issues, likedefining a base IRI for uploads or managing repository selection in a scripted way.

Extension Possibilities

The entire ABox population process is **semi-automated** in the sense that:

- No hand-written triples are required,
- Adding a new source file (e.g., a supermarket CSV) is sufficient,
- Scripts assume pre-cleaned, schema-compliant input from a trusted data zone.

This semi-automatic flow ensures consistency, reduces individual mistakes and errors, and could be fully automated in a production setting if needed. All Data cleaning is assumed to take place before, so this execution of an abox creation is only prototypically using the csvs from the landing zone - in the future this would be the trusted zone or depending on the implemented transformation stream, the exploitation zone.

By separating schema (TBox) and instance data (ABox) into distinct files (e.g., `http://budgetbites.org/` and `http://budgetbites.org/graph/abox`), the system supports modular development, schema evolution, and efficient querying.

M5 Using the Knowledgegraph

Use Case

The Budget Bites knowledge graph is exploited as a general-purpose semantic backend within an application context. Rather than being queried directly by multiple users, the graph serves as a backend reasoning layer for a single application interface, such as a website or app. This application automatically reads the current month, retrieves a seasonal ingredient for that time period, identifies relevant recipes, and dynamically maps the ingredient lists to real supermarket products and prices. Each step in this semantic traversal is executed through SPARQL queries over the knowledge graph. This exploitation pattern is visualized in Figure **??**, which shows how the application interacts with the graph using logical, dependency-driven steps rather than static queries. The graph enables contextual data assembly, such as identifying which supermarket products match an ingredient used in a recipe that is seasonal in December. Because this logic is embedded in the graph structure itself, no hardcoded rules are needed. The graph acts as a semantic processor that produces personalized, interpretable, and current shopping recommendations.

Graph Analytics Pipeline

The analytical process follows three standard stages: data preparation, model creation (with parametrization), and validation. In the preparation phase, raw data from APIs (MealDB, Mercadona, Carrefour) is translated, cleaned, and semantically enriched. Ingredients are translated from English to Spanish, and fuzzy-matched to supermarket products with a configurable confidence threshold (score . 70). These steps are implemented as Python-based ETL scripts that produce ABox and TBox RDF files. The model creation involves defining the ontology schema (owl:Class, owl:ObjectProperty, owl:DatatypeProperty) and assigning domain and range constraints using rdfs. Parametrization includes decisions like multilingual labels for ingredients, explicit seasonality modeling via bb:isSeasonalIn, and product metadata (e.g., price, unit, URL). This parametrization ensures that graph traversal remains usable for querying, and semantically also valid.

Graph Validation Steps

To validate the semantic logic and functional correctness of the graph, the following steps are executed as SPARQL queries against the generated construct:

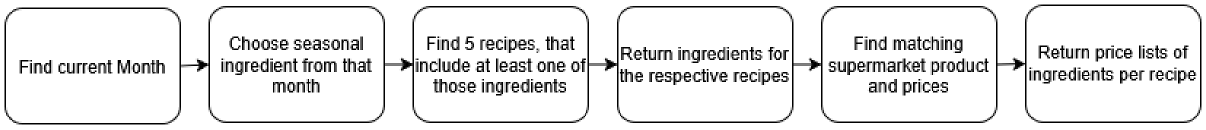

Find current Month
Choose seasonal ingredient from that month
Find 5 recipes, that include at least one of those ingredients
Return ingredients for the respective recipes
Find matching supermarket product and prices
Return price lists of ingredients per recipe

Figure 3: Validation Flow

1. **Find current month:** Verify that the current month (e.g., `December`) is represented as a `bb:Month` instance and accessible via its URI.

2. **Choose seasonal ingredient from that month:** Query all `bb:Ingredient` nodes linked to the current month through the `bb:isSeasonalIn` property.

3. **Find 5 recipes that include at least one of those ingredients:** Identify up to five `bb:Recipe` instances that are linked via `bb:usesIngredient` to any of the selected seasonal ingredients.

4. **Return ingredients for the respective recipes:** For each of the selected recipes, retrieve all associated ingredients via the `bb:usesIngredient` relationship.

5. **Find matching supermarket products and prices:** For each ingredient, traverse the `bb:matchesWithProduct` relation to access matched `bb:Product` instances. Filter for those sold by `bb:Mercadona` and include price metadata.

6. **Return price lists of ingredients per recipe:** Aggregate product price information per recipe to generate shopping lists, enabling comparison of total recipe costs.

Added Value of the general-purpose Knowledgegraph Approach

Using a knowledge graph for BudgetBites' long-term exploitation zone offers significant advantages over a traditional relational database system like DuckDB, which is used during the current initial prototyping phase. While DuckDB provides good tabular storage and SQL-based querying suitable for this early stage development, it doesnt fulfill the semantic advantages and flexibility required for scalable, cross-domain data integration. Looking in the future of the product, and adding further supermarkets product databases, the advantages of having one single global schema for all supermarkets, that doesnt have to be adapted when taking away or adding a new chain promise for knowledgegraph technology to be the most future proof. A knowledge graph, based on RDF, OWL, and SPARQL, easily enables the representation of the required, semantically-typed relationships between entities such as recipes, ingredients, supermarket products, and seasonal contexts, independently of the underlaying database - as long as the data structure meets the graphs requirements. Supporting multi-hop queries, like retrieving the cheapest recipe for a given season and supermarket, these are great usability advantages that are difficult to express or inefficient to compute in flat relational models.

Additionally, the graph structure allows for dynamic incorporation of new data source types, such sustainability labels, or nutritional metadata, without requiring a schema overhaul. It can hold multilingual data and align rather abstract ingredient concepts with concrete product listings, potentially enabling features like seasonal recipe recommendations, vendor-specific price comparison, and eventually even personalized dietary filtering.

By serving as a semantic backend for the application logic, the knowledge graph future-proofs the system against evolving requirements and enables great depth reasoning, like clustering similar recipes or inferring new relationships via graph embeddings. this being analysed, the graph-based approach not only enhances data integration and retrieval but could also lay the foundation for intelligent, adaptive services that go beyond the static capabilities of a relational prototype.

M6 – Metadata Generation

This stage of the knowledgegraph implementation does not generate any behavioural or use case driven metadata.
However, the knowledge graph architecture does generate and rely on structural and semantic metadata in the form of the provided TBox schema, using RDF, RDFS, and OWL. The TBox serves as a global metamodel that semantically defines:

- The core classes (e.g., bb:Recipe, bb:Ingredient, bb:Product, bb:Supermarket, etc.),
- Object properties (bb:usesIngredient, bb:matchesWithProduct, bb:isSeasonalIn, etc.),

- Domain and range constraints, symmetric/transitive properties, and multilingual labels.

This schema metadata is reused throughout the data population (M4) and exploitation (M5) phases to enable semantic queries, data consistency, and reasoning. All RDF triples — both schema and instance data — are stored in a modular and queryable way using named graphs for the TBox and ABox in GraphDB.

In future iterations, application-generated metadata (e.g., usage patterns, recommendation feedback, shopping history) could be added as a separate graph layer or external metadata store, but this is out of scope for the current proof-of-concept.

M7 Proof of Concept

For the PoC, a complete graph-based use case pipeline was implemented, using GraphDB.

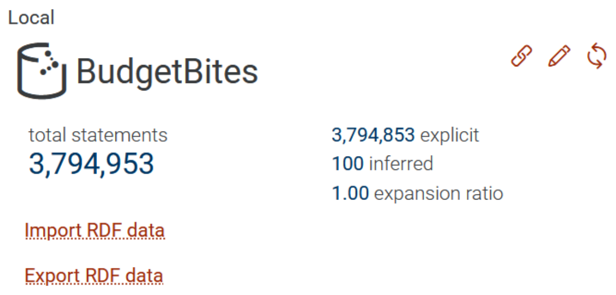
Tools and Set up

To implement and validate the knowledge graph–driven architecture described in previous sections, a proof of concept (PoC) was developed. The objective of the PoC was to execute a full end-to-end scenario, from raw data transformation to semantic querying, based entirely on real data and a functional knowledge graph.
As the technical backend is developed simultaniously to this semantic side, the data, used as a foundation for the created knowledge graph, does not match the fully pre-processed data, that can be found in the devloped exploitation zone of BDM, but rather consist of the result of basic preprocessing and matching of products. The graphs performance is not affected heavily, but the matching of products has shown to be less accurate as the matching in the final exploitation zone. In long-term development, the preprocessing and inclusion of data would be operated from the trusted zone, where supermarket data is already structurally aligned.
Table 2 summarizes the tools used at each stage of the PoC. RDF data was generated from CSV and JSON sources using Python scripts and the `rdflib` library. The ontology was defined using Turtle syntax, separating schema (TBox) and instance data (ABox). The resulting triples were loaded into GraphDB, a free and user-friendly RDF triplestore, which also served as the SPARQL query interface for executing analytical queries.

Function	**Tool Used**
RDF storage & querying	GraphDB
RDF generation	Python + rdflib
CSV + JSON parsing	Python scripts
Ontology design	Turtle (.ttl)
Query interface	GraphDB - SPARQL

Table 2: Table of Functions and Tools Used

The resulting Knowledgegraph in Graph DB has the following general statistics:


Local

BudgetBites

total statements
3,794,953

3,794,853 explicit
100 inferred
1.00 expansion ratio

Figure 4: GrapDB Repository Statistics

Execution Pipeline

The following Pipeline has been executed in individual Queries as a proof of concept.

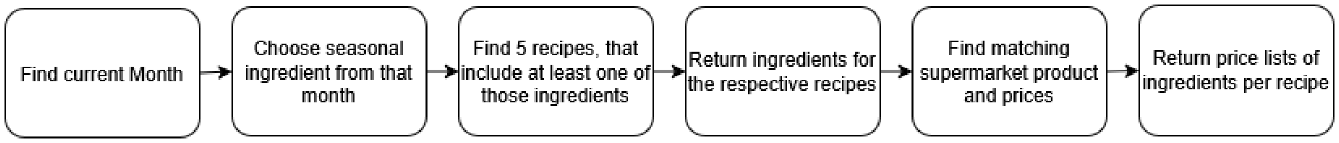

Find current Month
Choose seasonal ingredient from that month
Find 5 recipes, that include at least one of those ingredients
Return ingredients for the respective recipes
Find matching supermarket product and prices
Return price lists of ingredients per recipe

Figure 5: Exploitation process of the knowledge graph

Step 1:

Collect Seasonality In this step, the system queries the graph for all ingredients that are marked as seasonal in December using the bb:isSeasonalIn relationship. This forms the entry point for recipe selection based on seasonality. Results of the respective Query (listed in the Appendix in A.1) are shown in the following table:

Seasonal Ingredients in December
Brussels Sprouts
Cabbage
Cale

Table 3: Step 1 - Collect Seasonality

Step 2: Collect recipes with seasonal ingredient

Recipes retrieved through the `bb:usesIngredient` relationship where at least one ingredient is seasonal in December. This step narrows down the recipe candidates to those aligned with seasonal availability. Results of the respective Query (listed in the Appendix in A.2) are shown in the following table:

Recipes using seasonal ingredients in December
Ribollita
Lasagne
Wontons
Corba
Stamppot

Table 4: Step 2 – Recipes with a seasonal ingredient

Step 3: Collect the respective ingredients (EN)

All ingredients retrieved for the recipe *Corba*, which was selected for including at least one seasonal ingredient. This step uses the `bb:usesIngredient` relation to expand from the seasonal filter to the full recipe context. Results of the respective Query (listed in the Appendix in A.3) are shown in the following table:

Recipe Name	**Ingredient (EN)**
Corba	Black Pepper
Corba	Carrots
Corba	Cumin
Corba	Lentils
Corba	Mint
Corba	Onion
Corba	Paprika
Corba	Red Pepper Flakes
Corba	Sea Salt
Corba	Thyme
Corba	Tomato Puree
Corba	Vegetable Stock
Corba	Water

Table 5: Step 3 – Respective ingredients (excerpt)

Step 4: Get matched Mercadona products and prices for all recipe ingredients

This step focuses on retrieving real-world product information from Mercadona for each ingredient found in the recipe. By leveraging the `bb:matchesWithProduct` relation, the system links ingredients to actual store products and extracts relevant attributes such as product name, price, and unit. Results of the respective Query (listed in the Appendix in A.4) are shown in the following table:

Ingredient (EN)	Matched Mercadona Product	Price (€)
Black Pepper	Pimienta Negra Molida 40g	1.29
Carrots	Zanahorias Bolsa 1kg	0.99
Cumin	Comino Molido 35g	1.15
Lentils	Lentejas Cocidas Hacendado 570g	0.78
Onion	Cebolla Blanca Malla 1kg	1.25
Paprika	Pimentón Dulce Ahumado 75g	1.49
Red Pepper Flakes	Copos de Guindilla Picante 30g	1.60
Sea Salt	Sal Marina Gruesa 1kg	0.60
Tomato Puree	Tomate Triturado Hacendado 400g	0.89
Vegetable Stock	Caldo de Verduras Hacendado 1L	1.05

Table 6: Step 4 – Matched Mercadona Products

Step 5: Total recipe cost based on matched Mercadona products

In this step, each ingredient from the selected recipes is matched to a real product available in the Mercadona product catalog. The graph traversal uses the `bb:matchesWithProduct` relation to retrieve the product name and price for each ingredient.

Recipe Name	Total Cost (€)
Ribollita	11.38
Lasagne	9.72
Wontons	12.45
Corba	10.25
Stamppot	8.89

Table 7: Step 5 - Total recipe cost

Architecture and Justification

The proof-of-concept architecture follows a modular, semantic-first pipeline. Structured and semi-structured data — including product catalogs, translated ingredients, and recipes — are converted into RDF triples and stored in GraphDB. ABox and TBox are loaded as named graphs to separate schema and instance data. SPARQL queries are run via the GraphDB Workbench, which supports reasoning and type inference.
RDF was chosen for its ability to link cross-domain data like seasonal context, language, and product matching. GraphDB provides reasoning, ontology support, and intuitive visualization — ideal for rapid prototyping. This setup enables complex queries that go beyond relational joins, such as linking seasonal ingredients to recipes and matching them to supermarket products with live pricing.

References

1 Rdf-grapher - *https://www.ldf.fi/service/rdf-grapher*
2. GitHub Repository - *https://github.com/JuleGri/BudgetBites_GraphDB.git*

A SPARQL Queries

A.1 Step 1: Collect Seasonality

```
PREFIX bb: <http://budgetbites.org/ontology#>

SELECT DISTINCT ?ingredientNameEN
WHERE {
  ?ingredient a bb:Ingredient ;
              bb:isSeasonalIn <http://budgetbites.org/resource/month/May
                 > ;
              bb:ingredientNameEN ?ingredientNameEN .
}
ORDER BY ?ingredientNameEN
```

Listing 1: Collect Seasonality

A.2 Step 2: Collect recipes with seasonal ingredients

```
PREFIX bb: <http://budgetbites.org/ontology#>

SELECT DISTINCT ?recipe ?recipeName
WHERE {
  ?ingredient a bb:Ingredient ;
              bb:isSeasonalIn <http://budgetbites.org/resource/month/
                 December> .

  ?recipe a bb:Recipe ;
          bb:usesIngredient ?ingredient ;
          bb:name ?recipeName .
}
LIMIT 5
```

Listing 2: Collect recipes with seasonal ingredients

A.3 Step 3: Collect the ingredients (EN) of the picked recipes

```
PREFIX bb: <http://budgetbites.org/ontology#>

SELECT DISTINCT ?recipeName ?ingredientNameEN
WHERE {
  ?ingredient a bb:Ingredient ;
              bb:isSeasonalIn <http://budgetbites.org/resource/month/
                 December> .
  ?recipe a bb:Recipe ;
          bb:usesIngredient ?ingredient ;
          bb:name ?recipeName .
  ?recipe bb:usesIngredient ?anyIngredient .
  ?anyIngredient bb:ingredientNameEN ?ingredientNameEN .
}
ORDER BY ?recipeName ?ingredientNameEN
```

Listing 3: Collect the respective ingredients (EN)

A.4 Step 4: Collect the matching mercadona products

```
PREFIX bb: <http://budgetbites.org/ontology#>

SELECT DISTINCT ?recipeName ?ingredientNameEN ?productName ?price
WHERE {
  ?seasonalIngredient bb:isSeasonalIn <http://budgetbites.org/resource/
     month/December> .

  ?recipe a bb:Recipe ;
          bb:usesIngredient ?seasonalIngredient ;
          bb:name ?recipeName .
  ?recipe bb:usesIngredient ?ingredient .
  ?ingredient bb:ingredientNameEN ?ingredientNameEN .
  ?ingredient bb:matchesWithProduct ?product .
  ?product bb:name ?productName ;
           bb:price ?price ;
           bb:soldBy bb:Mercadona .
}
ORDER BY ?recipeName ?ingredientNameEN
```

Listing 4: Collect the matching mercadona products

A.5 Step 5: Total recipe cost based on matched Mercadona products

```
PREFIX bb: <http://budgetbites.org/ontology#>

SELECT ?recipeName (SUM(?price) AS ?totalCost)
WHERE {
  ?seasonalIngredient bb:isSeasonalIn <http://budgetbites.org/resource/
     month/December> .
  ?recipe a bb:Recipe ;
          bb:name ?recipeName ;
          bb:usesIngredient ?seasonalIngredient .

  ?recipe bb:usesIngredient ?ingredient .
  ?ingredient bb:matchesWithProduct ?product .
  ?product bb:price ?price ;
           bb:soldBy bb:Mercadona .
}
```

```
GROUP BY ?recipeName
ORDER BY ASC(?totalCost)
```

Listing 5: Total recipe cost based on matched Mercadona products