

GameMatcher

*Suggesting compatible video games based on your laptop's
hardware specifications*

Actionable Knowledge Representation

D3 - Knowledge Graph, Knowledge Extraction & Queries

Group Members:

Andreea Scrob

Edoardo Tommasi

1 Updated Ontology Definitions

To make the ontology work in a proper manner we had to update one data property by splitting it in two and modify an already existing one.

Initially we had the `requiresBenchmarkScore` data property but it wasn't enough to properly express the minimum requirements for both CPU and GPU so it was split into `requiresCPUBenchmarkScore` and `requiresGPUBenchmarkScore` allowing the ontology to hold multiple integer values.

```
class requiresCPUBenchmarkScore(DataProperty, FunctionalProperty):
    domain = [MinimumRequirement]
    range = [int]
    label = ["requires CPU benchmark score"]

class requiresGPUBenchmarkScore(DataProperty, FunctionalProperty):
    domain = [MinimumRequirement]
    range = [int]
    label = ["requires GPU benchmark score"]
```

Then we updated the `hasBenchmarkScore` data property by simply replacing the domain from `[CPU,GPU]` to `[CPU|GPU]` because it gave us disjointness problems.

```
class hasBenchmarkScore(DataProperty, FunctionalProperty):
    domain = [CPU | GPU]
    range = [int]
    label = ["has benchmark score"]
```

2 Data Collection

2.1 Scraped Data

This subsection details the parameters extracted during the data scraping process. The dataset integrates commercial metadata, gameplay classifications, and technical hardware requirements to facilitate a multi-dimensional analysis of the current gaming landscape.

2.1.1 Core Game Metadata

The following attributes define the identity and market positioning of each title:

- **Name:** the official title of the software.
- **Price:** the retail value at the time of scraping.
- **PEGI:** the age rating as defined by the Pan European Game Information board.
- **Mode:** player configuration support, categorized by Single-player, Multiplayer, or Co-operative play etc...

2.1.2 Technical System Requirements

To assess the hardware compatibility and accessibility of each title, the following specifications were recorded:

- **OS:** the minimum supported Operating System version. We used Windows as it was present in the vast majority of games and is the most used by gaming computers.
- **CPU & GPU:** the specific processor and graphics card models recommended by the developers.
- **RAM:** the system memory requirement, standardized in Gigabytes (GB).
- **STORAGE:** the persistent storage footprint required for a full installation.

2.1.3 Standardized Performance Benchmarks

Because hardware names (e.g., "GTX 1060" vs "RX 580") do not provide a linear scale for analysis, the dataset includes synthetic benchmark scores. This allows for quantitative modeling of the relationship between game complexity and hardware demand. An "RX 580" corresponds to a G3DMark score of 8,795 while a "GTX 1060" corresponds to a score of 10,060.

$10,060 > 8,795$ meaning the "GTX 1060" is the better card. The same discourse is used for CPUs.

These scores provide a normalized baseline, enabling the calculation of hardware intensity regardless of specific manufacturer branding.

2.2 Sites

The dataset construction relies on data aggregated from two external platforms. [Steam](#) serves as the source for game cataloging, providing metadata and textual system requirements for the most popular titles. Complementarily, [PassMark](#) is utilized to convert these hardware specifications into numerical "Mark" scores. The following sections detail the custom scrapers developed to extract, clean, and harmonize data from these domains.

2.2.1 Steam

To compile a dataset of the "Top 100 Most Played" games, a custom Python scraper was developed using `requests` and `BeautifulSoup`.

Access and Retrieval

The script queries the Steam leaderboard filtered by `?filter= mostplayed` to identify target URLs.

- **Access Control:** to automate access to mature content (Age-Gate), HTTP headers were injected with session cookies (`birthtime`, `lastagecheckage`) to preemptively validate the session.
- **Rate Limiting:** a 0.7-second delay was enforced between requests to prevent server-side blocking.

Extraction and Storage

Data extraction employed specific parsing strategies based on the attribute type:

- **Metadata:** basic fields (Price, PEGI) were extracted by targeting specific DOM elements (e.g., `game_purchase_price`).
- **Game Modes:** modes were classified by matching tags within the `game_area_features_list_ctn` container against a predefined keyword list (e.g., “Co-op”, “Multi-player”).
- **Hardware Specs:** system requirements were parsed using Regular Expressions to locate labels (e.g., “Processor:”) and capture the immediate sibling text.

The extracted data was cleaned, serialized, and exported firstly to CSV (to analyze it in excel type of software, as it is more compatible), then to JSON format.

2.2.2 PassMark

To translate hardware requirements into quantitative metrics, an object-oriented scraper was developed to retrieve “CPU Mark” and “G3D Mark” scores from PassMark databases.

Access and Architecture Target domains utilize TLS fingerprinting to block standard requests. The `curl_cffi` library was implemented to overcome this by impersonating a legitimate browser handshake (Chrome 120).

- **Session Logic:** the scraper executes a specific request sequence to establish session validity before accessing data endpoints.
- **Caching Strategy:** extracted data is serialized to local JSON files (`cpu_data.json`). The system prioritizes local storage, querying the server only if the cache is missing.

Data Processing The module initializes in either CPU or GPU mode and applies sanitization during extraction:

- **Normalization:** raw inputs containing artifacts (e.g., “NA”, commas) are stripped and converted to integers.
- **Fault Tolerance:** invalid or missing entries default to 0 to preserve dataset integrity.

2.3 Data Processing and Hardware Standardization

The raw data collected from Steam regarding system requirements is often inconsistent because it is written by humans for humans, not for machines. To transform this unstructured text into a usable format for our Knowledge Graph, we developed a two-step processing pipeline. You can think of this process as a translation service that converts rough notes into a formal dictionary.

1. Text Cleaning and Parsing

The first step acts like a filter. The raw requirement strings often contain noise, such as brand trademarks, marketing terms, or unnecessary words like *processor* or *graphics card*.

Our system cleans this text by stripping away these elements, leaving only the essential model names. Additionally, when a game lists multiple options (for example, suggesting either an Intel or an AMD processor), the system splits these into distinct items so they can be analyzed individually.

2. Fuzzy Matching with Benchmark Data

Once we have the clean model names, we need to assign them a performance score. However, the names used on Steam rarely match the official names in our benchmark database perfectly. To solve this, we implemented a similarity matching algorithm. The system compares the name from Steam against thousands of entries in our database and selects the one with the highest text similarity. This ensures that a requirement written simply as *GTX 1060* is correctly identified and linked to the official *NVIDIA GeForce GTX 1060* entry in our records, allowing us to assign precise performance metrics to every game.

3. Pipeline Efficiency Results

The quantitative results demonstrate the robustness of this two-step approach. Out of 100 top-selling games scraped from Steam, our parsing module successfully structured the requirements for 93 titles and saved them in the `steam_parsed_CLEAN.json` file, achieving a 93% success rate in handling unstructured natural language. The remaining 7 titles were identified as edge cases and saved in the `steam_parsed_COMPLEX.json` file. These entries exhibited highly unstructured formatting that fell outside the scope of our automated parsing patterns. Since resolving these ambiguities would have required manual intervention and considering they represented a statistically negligible fraction of the dataset (7%), we collectively decided to exclude them from the final Knowledge Graph to prioritize process efficiency.

Crucially, the fuzzy matching strategy proved essential for optimizing the Knowledge Graph’s size. Instead of blindly importing the entire benchmark database, the system filtered and instantiated only the components actually required by the valid games, yielding the following reduction rates:

- **CPU Optimization:** the dataset was reduced from 6,226 total entries to just 69 relevant individuals (a 98.89% reduction).
- **GPU Optimization:** the dataset was reduced from 2,987 entries to 80 relevant individuals (a 97.32% reduction).

This selective instantiation ensures that the resulting ontology remains lightweight and highly efficient for reasoning queries, avoiding thousands of unused nodes.

Scalability Note

It is important to note that the drastic reduction in the number of hardware individuals was a deliberate design choice to maintain the Knowledge Graph lightweight and focused during the development phase. We strictly imported only those components that were explicitly referenced by the games in our dataset. However, the system is designed for full scalability: the fuzzy matching algorithm can inherently query the entire benchmark database (over 9,000 components combined), allowing the ontology to expand dynamically as new games with different requirements are added in the future.

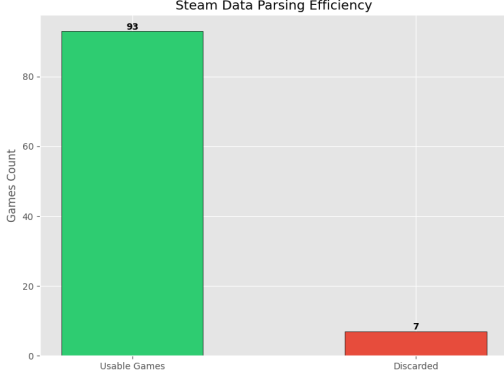


Figure 1: Steam Parsing Success Rate

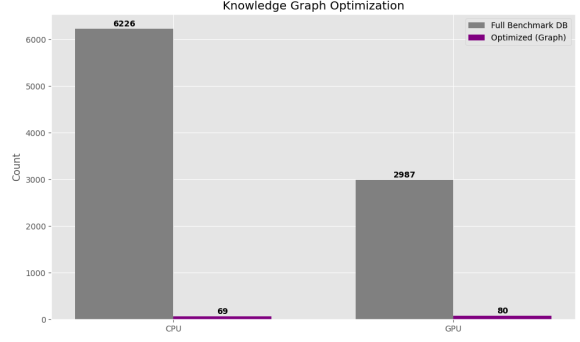


Figure 2: Knowledge Graph Optimization

2.4 ABox Population and Linking Strategy

To ensure data consistency and optimize the Knowledge Graph structure, we implemented a sophisticated ABox population logic using Python and the `owlready2` library. Instead of simply instantiating new hardware objects for every user, we designed a **Semantic Linking Strategy** that prioritizes data reuse.

The implementation follows a three-step logic:

- **Pre-population of the Hardware Catalog (Standard Individuals):**

Before generating any user data, the system populates the ABox with a comprehensive “catalog” of hardware components derived from our benchmark datasets. Unique individuals are created for every specific CPU and GPU model (e.g., `CPU_Intel_Core_i7-4790K`), as well as for standard RAM sizes (e.g., `RAM_16GB`) and Operating System versions. These serve as shared reference nodes within the graph.

- **Dynamic Discovery and Linking:**

When populating a User profile or defining a Game, the system does not immediately create new hardware components and preferences. Instead, it parses the input specifications (whether they come from a user’s PC config or a game’s minimum/recommended specs), normalizes the strings to generate valid IRIs, and queries the ontology to check for the existence of a corresponding “Standard Individual.”

- *If a match is found:* The entity (User’s Computer or Game Requirement) is linked directly to the existing shared component. This creates a **“Single Source of Truth,”** ensuring that a User who owns a specific CPU and a Game that requires that exact same CPU are linked to the identical node, greatly simplifying the reasoning process.

- **Robust Fallback Mechanism:**

To ensure the system’s robustness against incomplete datasets, we implemented a fallback logic. If a user possesses a component (or if a game lists a specific requirement) that is not present in the pre-populated catalog (e.g., a rare CPU or a non-standard OS version), the system automatically detects the absence and instantiates a **Local Individual** (e.g., `CPU_Local_Instance`). This guarantees that the ontology remains valid and complete even when dealing with edge-case hardware configurations.

This hybrid approach minimizes data redundancy (avoiding thousands of duplicate nodes for popular hardware) while maintaining the flexibility to handle custom or edge-case configurations.

2.4.1 Missing Data Fixes

Some data from steam could not be scraped as it either wasn't present or written in an inconsistent manner. To fix this, placeholder instances were created in the ontology for data classes that produced missing content but still required an instance (like PEGI), other were just given a value of 0 to avoid coherency problems.

2.4.2 Ontology Numbers

Ontology Metrics:

- **Metrics:** 93 Classes, 128 Object Properties, 18 Data Properties.
- **Individuals (441 total):** 93 Videogames, 93 Min. Requirements, 75 CPUs, 84 GPUs, 11 RAM, 14 Storage, 22 Genres, 11 Modes, 5 PEGI ratings, 5 OS, 14 users, 14 computers.

2.5 Synthetic User Generation via LLM

To populate the ABox with a diverse and realistic set of test users, we leveraged the **Gemini 3 Pro** Large Language Model (LLM) to generate the Python instantiation code automatically. This approach allowed us to rapidly create varied user personas (e.g., "Budget Gamer", "Workstation User", "Retro Gamer") without manually writing boilerplate code.

We designed a strict prompt, reported in full in Appendix A, providing the model with the exact function signature (`create_user_with_existing_hardware`) and specific constraints regarding data types and hardware realism.

Initial Prompt Strategy and Validation

The initial prompt instructed the LLM to generate 10 users using "real and popular hardware names from the last 10 years."

*"Generate 10 new user instances... Use real and popular hardware names...
Create a mix of user personas (Low-End, Mid-Range, High-End)..."*

Upon executing the generated code, we observed that approximately **75% of the hardware components** (CPUs and GPUs) suggested by the LLM were not present in our filtered benchmark dataset. This high "miss rate" occurred because the LLM drew from its general knowledge of all existing hardware, whereas our Knowledge Graph contains a specific subset of components derived from Steam requirements.

However, this discrepancy served as a valuable **stress test for our Semantic Linking Strategy**. It verified that the system's fallback logic functions correctly: when the specific hardware was not found in the standard catalog, the system successfully instantiated a local hardware individual for that user, preventing the application from crashing.

Refined Generation

To ensure a balanced ABox containing both “Standard” (linked) and “Local” (fallback) individuals, we refined the prompting strategy. We injected the list of valid `cpu_filtered.json` and `gpu_filtered.json` names directly into the prompt context, explicitly instructing the model:

“For 5 out of the 10 users, please select the CPU and GPU from the files I have attached.”

Consequently, the LLM followed a split generation logic:

- **5 Users** used hardware strictly selected from the provided JSON lists (guaranteeing a match and semantic linking).
- **5 Users** were instructed to use diverse hardware outside of these lists. However, due to the high coverage of our filtered dataset, only **1.5 users on average** resulted in completely new local instances, as the model often selected popular hardware that, although not explicitly forced, was already present in our knowledge base.

This hybrid approach resulted in a robust test dataset that covers all branches of our population logic. The resulting synthetic user profiles have been successfully instantiated and are currently populated within the final version of the ontology.

3 Rules

Lastly, we implemented different rules to facilitate research of specific data in queries by introducing inferred object properties.

R1 Suggested Games

Idea: To suggest games to a user that, even though his computer might not run, he still might like to play in the future.

Description: This rule simply checks whether a specific game has both the genre and mode favourite to the user. If it does, the "PerfectMatchFor" object property gets created linking the videogame -> user.

```
if not "isPerfectMatchFor" in locals():
    class isPerfectMatchFor(ObjectProperty):
        domain = [VideoGame]
        range = [User]
        label = ["is a perfect match for"]
```

```
rule_perfect = Imp()
rule_perfect.set_as_rule(
    """VideoGame(?v), User(?u),
       hasGenre(?v, ?g), preferredGenre(?u, ?g),
       hasPlayerMode(?v, ?m), preferredPlayerMode(?u, ?m)
       -> isPerfectMatchFor(?v, ?u)"""
)
```

R2 Compatibility Award

Idea: A fun easter egg that appears when a user has all the specific minimum requirement for a game, making some sounds and animations on the website to "award" him.

Description: This rule checks whether the CPU, GPU, RAM, OS of the user are the same of those in the minimum requirements for games.

```
if not "isPerfectHardwareMatch" in locals():
    class isPerfectHardwareMatch(ObjectProperty):
        domain = [MinimumRequirement]
        range = [Computer]
        label = ["Compatibility Award"]
```

```
rule_perfect_hw = Imp()
rule_perfect_hw.set_as_rule(
    """MinimumRequirement(?req), Computer(?c),
    hasCPU(?c, ?u_cpu), hasBenchmarkScore(?u_cpu, ?score_cpu),
    hasGPU(?c, ?u_gpu), hasBenchmarkScore(?u_gpu, ?score_gpu),
    hasOS(?c, ?u_os), hasOSVersionValue(?u_os, ?os_ver),
    requiresCPUBenchmarkScore(?req, ?score_cpu),
    requiresGPUBenchmarkScore(?req, ?score_gpu),
    requiresMinOSVersionValue(?req, ?os_ver)
    -> isPerfectHardwareMatch(?req, ?c)""",
    namespaces=[onto]
)
```

4 Queries

Once the Knowledge Graph was finalized, it was uploaded to the [TriplyDB](#) platform. This environment was selected to streamline the development, testing, and execution of the following SPARQL queries, providing a robust interface for interacting with the graph's linked data.

CQ1 Hardware Compatibility Check

Question: Which video games can run on the computer owned by User X?

Description: This query retrieves the hardware specifications of a specific user's computer (CPU, GPU, RAM, Storage, and OS) and compares them against the minimum requirements of all video games in the database. It filters the results to show only games where every user specification meets or exceeds the game's requirements.

```
PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?nameGame
WHERE {
    # Define the input
    VALUES (?targetUser) {
        ("Giulio")
    }
```

```

?user rdfs:label ?name .
FILTER(CONTAINS(LCASE(?name), LCASE(?targetUser)))
?videogame rdfs:label ?nameGame .
?user onto:hasComputer ?pc .
?pc onto:hasCPU ?cpuU ;
    onto:hasGPU ?gpuU ;
    onto:hasRAM ?ramU ;
    onto:hasOS ?osU ;
    onto:hasStorage ?romU.

?cpuU onto:hasBenchmarkScore ?cpu_scoreU .
?gpuU onto:hasBenchmarkScore ?gpu_scoreU .
?ramU onto:hasMemorySizeGB ?ram_sizeU .
?romU onto:hasStorageSizeGB ?rom_sizeU .
?osU onto:hasOSVersionValue ?os_versionU .

?videogame onto:hasMinRequirement ?min_req .
?videogame onto:hasPrice ?price .

?min_req onto:requiresCPUBenchmarkScore ?cpu_scoreG ;
    onto:requiresGPUBenchmarkScore ?gpu_scoreG ;
    onto:requiresMemoryGB ?ram_sizeG ;
    onto:requiresStorageSpaceGB ?rom_sizeG ;
    onto:requiresMinOSVersionValue ?os_versionG .

BIND(IF(?cpu_scoreU >= ?cpu_scoreG, "OK", "WEAK") AS ?cpu_status)
BIND(IF(?gpu_scoreU >= ?gpu_scoreG, "OK", "WEAK") AS ?gpu_status)
BIND(IF(?ram_sizeU >= ?ram_sizeG, "OK", "WEAK") AS ?ram_status)
BIND(IF(?rom_sizeU >= ?rom_sizeG, "OK", "WEAK") AS ?storage_status)
BIND(IF(?os_versionU >= ?os_versionG, "OK", "WEAK") AS ?os_status)

FILTER(?cpu_scoreU >= ?cpu_scoreG && ?gpu_scoreU >= ?gpu_scoreG && ?
    ↪ ram_sizeU >= ?ram_sizeG && ?rom_sizeU >= ?rom_sizeG && ?
    ↪ os_versionU >= ?os_versionG)
}

```

CQ2 Budget Feasibility Check

Question: Can User X afford to buy VideoGame Y?

Description: This query compares the user's defined budget with the price of the video games available in the Knowledge Graph. It returns a list of games that fall within the user's spending limit, ordered by price in ascending order.

```

PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?nameGame ?price
WHERE {
    # Define the input
    VALUES (?targetUser) {
        ("Giulio")
    }
}

```

```

}

?user rdfs:label ?name .
FILTER(CONTAINS(LCASE(?name), LCASE(?targetUser)))
?videogame rdfs:label ?nameGame .
?user onto:hasBudget ?budget .
?videogame onto:hasPrice ?price .

BIND(IF(?price <= ?budget, "CanBuy", "TooExpensive") AS ?
      ↪ budget_status)
FILTER(?price <= ?budget)
}
ORDER BY ASC(?price)

```

CQ3 PEGI Age Compliance

Question: Is User X old enough to play VideoGame Y based on the PEGI rating?

Description: This query checks for age compliance by matching the user's age against the PEGI age threshold associated with each video game. It filters out games that are rated for an age higher than the user's current age.

```

PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?nameGame ?pegi_value
WHERE {
  # Define the input
  VALUES (?targetUser) {
    ("Giulio")
  }

  ?user rdfs:label ?name .
  FILTER(CONTAINS(LCASE(?name), LCASE(?targetUser)))

  ?videogame rdfs:label ?nameGame .

  ?user onto:hasAge ?age .
  ?videogame onto:hasPEGI ?pegi .
  ?pegi onto:hasPEGIAgeThreshold ?pegi_value.

  BIND(IF(?pegi_value <= ?age, "CanPlay", "TooYoung") AS ?pegi_status)

  FILTER(?pegi_value <= ?age)
}

```

CQ4 Genre Preference Matching

Question: Which video games belong to the genres preferred by User X?

Description: This query identifies games that match the genres explicitly preferred by the user in their profile. It uses a DISTINCT clause to ensure that each matching game is listed only once, even if it falls under multiple preferred categories.

```

PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?nameGame ?genreLabel
WHERE {
  VALUES (?targetUser) { ("Dev_User") }

  ?user rdfs:label ?name .
  FILTER(CONTAINS(LCASE(?name), LCASE(?targetUser)))

  ?user onto:preferredGenre ?genre .
  ?genre rdfs:label ?genreLabel .

  ?videogame a onto:VideoGame ;
             rdfs:label ?nameGame ;
             onto:hasGenre ?genre .
}
ORDER BY ?genreLabel

```

CQ5 Player Mode Compatibility

Question: Does VideoGame Y support the player mode preferred by User X?

Description: This query filters video games based on the player modes (e.g., Single-player, Multiplayer) that the user has marked as a preference. It returns all games that support at least one of these preferred modes.

```

PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?nameGame ?modeLabel
WHERE {
  # Define the input user
  VALUES (?targetUser) {
    ("Mark")
  }

  ?user rdfs:label ?name .
  FILTER(CONTAINS(LCASE(?name), LCASE(?targetUser)))

  ?user onto:preferredPlayerMode ?mode .
  ?mode rdfs:label ?modeLabel .
  ?videogame a onto:VideoGame ;
             rdfs:label ?nameGame ;
             onto:hasPlayerMode ?mode .
}
ORDER BY ?nameGame

```

CQ6 RAM Requirement

Question: Which video games require more than Z GB of RAM?

Description: This query analyzes technical requirements across the database to identify games that require a high amount of memory. It uses a threshold value to

filter the results and orders them by the amount of RAM required in descending order.

```
PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?nameGame ?requiredRAM
WHERE {
  # Define the input threshold Z (e.g., games requiring MORE than 4 GB)
  BIND(10.0 AS ?thresholdRAM)

  ?videogame a onto:VideoGame ;
             rdfs:label ?nameGame .

  ?videogame onto:hasMinRequirement ?min_req .

  ?min_req onto:requiresMemoryGB ?requiredRAM .

  FILTER(?requiredRAM > ?thresholdRAM)
}
ORDER BY DESC(?requiredRAM)
```

4.1 Additional Queries

N1 *Name:* Suggested Games

Description: This query simply checks whether there are games having the **isPerfectMatchFor** (meaning same genre and mode as the user's favourite) object property connecting it to the current user.

```
PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?nameGame
WHERE {
  VALUES ?targetUser { "Giulio" }
  ?user rdfs:label ?userName .
  FILTER(CONTAINS(LCASE(?userName), LCASE(?targetUser)))

  ?videogame onto:isPerfectMatchFor ?user .

  ?videogame rdfs:label ?nameGame .
}
```

N2 *Name:* Compatibility Award

Description: This query checks whether there are minimum requirements having the **isPerfectHardwareMatch** (meaning exactly same gpu,cpu,os and ram) object property connecting them to the user's computer.

```
PREFIX onto: <http://www.semanticweb.org/gamematcher.owl#>
```

```

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

SELECT ?nameGame
WHERE {
  VALUES ?targetUser { "Marco" }
  ?user rdfs:label ?userName .
  FILTER(CONTAINS(LCASE(?userName), LCASE(?targetUser)))

  ?user onto:hasComputer ?pc .

  ?videogame rdfs:label ?nameGame .
  ?videogame onto:hasMinRequirement ?min_req .

  ?min_req onto:isPerfectHardwareMatch ?pc .
}

```

A LLM Prompt for Synthetic User Generation

The following text represents the complete prompt used to instruct the Large Language Model in generating the initial batch of synthetic users.

```

I am developing an ontology for a Game Matching system and I need to populate
    ↳ the ABox with diverse test users.
I have a Python function named create_user_with_existing_hardware that handles
    ↳ the instantiation.

The Goal:
Please generate 10 new user instances using exactly the same function signature
    ↳ and structure as the examples below.

Constraints & Requirements:
1. Format: Follow the Python syntax strictly. Do not change the dictionary keys
    ↳ (user_name, age, budget, pc_specs, preferences).
2. Hardware Realism: Use real and popular hardware names (CPUs and GPUs) from
    ↳ the last 10 years (e.g., Intel Core i3/i5/i7/i9, AMD Ryzen, NVIDIA GTX/
    ↳ RTX series, AMD Radeon). This is crucial because my system matches these
    ↳ strings against a benchmark database.
3. OS Version: Keep os_ver as a float (e.g., 10.0, 11.0, 7.0).
4. Variety: Create a mix of user personas, for example:
    - Low-End/Retro Gamer: Older hardware, low budget, older OS.
    - Mid-Range/Student: Decent CPU/GPU, moderate RAM (8-16GB).
    - High-End/Streamer: Top-tier specs, 32GB+ RAM.
    - Laptop User: Integrated graphics or mobile GPU versions.
5. Preferences: Vary the genres (e.g., Action, RPG, Strategy, Sports, Adventure
    ↳ , Racing) and modes (Single-player, Multiplayer, Co-op).

Here is the code pattern to follow (Examples):

# 1. Mark: High-End PC
user_mark = create_user_with_existing_hardware(

```

```

    user_name="Mark",
    age=25,
    budget=80.0,
    pc_specs={
        "cpu_name": "Intel Core i7-4790K",
        "gpu_name": "NVIDIA GeForce GTX 1050",
        "ram_gb": 64.0,
        "storage_gb": 1000.0,
        "os_ver": 11.0
    },
    preferences={
        "genres": ["Action", "RPG"],
        "modes": ["Multiplayer"]
    }
)

# 2. Giulio: Budget Gamer
user_giulio = create_user_with_existing_hardware(
    user_name="Giulio",
    age=16,
    budget=20.0,
    pc_specs={
        "cpu_name": "Intel Core i5-6600K",
        "gpu_name": "NVIDIA GeForce GTX 1060",
        "ram_gb": 8.0,
        "storage_gb": 500.0,
        "os_ver": 10.0
    },
    preferences={
        "genres": ["Simulation", "Indie"],
        "modes": ["Single-player"]
    }
)

# 3. Dev_User: Workstation (Ultra High Specs)
user_workstation = create_user_with_existing_hardware(
    user_name="Dev_User",
    age=30,
    budget=500.0,
    pc_specs={
        "cpu_name": "AMD Ryzen 9 7950X",
        "gpu_name": "GeForce RTX 4090",
        "ram_gb": 100.0,
        "storage_gb": 4000.0,
        "os_ver": 19.0
    },
    preferences={
        "genres": ["Strategy"],
        "modes": ["Single-player"]
    }
)

```

Please output only the Python code for the new users.

B Links

1. Github repository: <https://github.com/andreeascrob/GameMatcher.git>
2. Steam top 100: <https://store.steampowered.com/charts/mostplayed>
3. Benchmarks: https://www.cpubenchmark.net/CPU_mega_page.html and https://www.videocardbenchmark.net/GPU_mega_page.html
4. TriplyDB: <https://trilydb.com/EdoardoTommasi/GameMatcher/>
 - Query 1: <https://api.trilydb.com/s/Mkc-kiM5Y>
 - Query 2: https://api.trilydb.com/s/_FEDN4ChW
 - Query 3: <https://api.trilydb.com/s/BMkNn793G>
 - Query 4: <https://api.trilydb.com/s/lPTNsH5oD>
 - Query 5: <https://api.trilydb.com/s/tTiXDq66M>
 - Query 6: <https://api.trilydb.com/s/kCeB-N1ST>
 - Query N1: <https://api.trilydb.com/s/nd7p0igrh>
 - Query N2: https://api.trilydb.com/s/HX2I_caFZ
5. Gemini prompt:
 - (a) First attempt: <https://gemini.google.com/share/e9ce998c6ade>
 - (b) Second attempt: <https://gemini.google.com/share/251dbd59e6eb>