CHAFFIN Antoine, YASSINE Taha, DELAUNAY Julien

M2 SIF

Projet DKM - DBPanda

Sommaire

Chapitre 1

Introduction

Dans le cadre du cours de DKM (**D**ata and **K**nowledge **M**anagement), nous avons étudié le web sémantique et la représentation RDF.
Ainsi, il nous a été demandé de réaliser une petite base de données mettant en œuvre les différents aspects traités en cours.

Le présent rapport détaille les différentes phases et la réalisation de ce projet :

- Le choix de la base à modéliser,
- La récupération des données,
- La création de l'endpoint SPARQL,
- La modélisation RDF,
- Le requêtage SPARQL,
- L'exploration via SPARKLIS.

Le modèle sous la forme d'un fichier .nt (au format **n-triples**), un fichier .xml (au format **RDF/XML**) ainsi qu'un fichier .ttl (au format **turtle**) est disponible sur le dépôt *github* du projet [1]. Le code écrit pour récupérer la base de données au format JSON ainsi que pour générer les fichiers XML, NT et TTL sont disponibles sur ce même dépôt.

1. https ://github.com/Borderlands-4/DBPanda

Chapitre 2

Choix et récupération des données

2.1 Choix de la base de données à modéliser

Nous avons recherché une base de données qui soit à la fois intéressante au niveau de sa structure mais aussi qui corresponde à nos centres d'intérêt communs. En cherchant sur internet nous sommes tombés sur le site Panda Score qui fournit des informations sur l'eSport (équipes, joueurs, tournois, ...) via l'intermédiaire d'une API. Nous avons été étonnés de trouver si peu de bases de données disponibles sur ce sujet. L'eSport est un domaine en plein essor et en contact direct avec les ressources numériques, nous pensions trouver diverses ressources. Nous avons constaté au contraire que ce n'était pas le cas et c'est pourquoi nous pensons que notre travail de traduction de base de données vers une base de connaissances dans le domaine de l'eSport est novateur. Pour être complet, notons aussi que Panda Score propose une API payante qui permet d'avoir accès à d'autres données, telles que les résultats en direct mais les deadlines imposées et le manque de moyens ne nous ont pas permis d'approfondir ce sujet qui aurait pourtant mérité d'être exploré.

Ainsi, nous avons trouvé un sujet qui nous motive avec une base de données avec une structure riche et complète et qui offre la possibilité d'accéder facilement à une quantité d'information assez importante pour créer notre base de connaissances.

2.2 Récupération des informations

L'API possède différents endpoints qui renvoient le contenu de la base de données sous forme de JSON. En plus de posséder des endpoints pour récupérer la liste entière de nos différents objets à savoir :

— Les leagues,
— Les séries,
— Les tournois,

— Les matches,
— Les équipes,
— Les joueurs,

cette API permet de récupérer directement ces objets pour un jeu vidéo donné.
Nous avons choisi de récupérer les informations de chaque jeu de manière indépendante pour mieux structurer notre base de données JSON et pouvoir effectuer le cas échéant un traitement spécialisé pour chacun. Ce choix permet aussi d'éditer ou de mettre à jour plus facilement une partie spécifique du modèle. De part la structure extrêmement changeante de la scène du jeu vidéo, nous avons essayé de prévoir l'ajout d'un ou plusieurs jeux ou la modification de ceux-ci.

Pour ce faire, nous avons créé un script qui permet de récupérer la totalité d'un type d'objet sur un jeu précis et de le stocker. Nous l'avons ensuite adapté pour pouvoir récupérer l'intégralité de la base en une seule exécution, notamment pour de ne pas avoir à stocker la base de donnée JSON sur git, chacun pouvant l'obtenir en quelques secondes. En effet, ces informations ne nous appartenant pas directement, il nous semblait peu respectueux de les rendre disponibles publiquement sous la forme de fichiers téléchargeables. De plus, la mise à jour des données est aussi facilitée par ce script puisqu'on peut récupérer toutes les nouvelles informations et les modifications en une exécution.

Chapitre 3

Création de la base de données

3.1 Création de l'endpoint

Avant de créer notre modèle, il était important de pouvoir le tester au fur et a mesure de sa création. Pour cela, nous avons utilisé Fuseki (version docker et standalone) pour créer un endpoint que nous utiliserons pour faire des requêtes SPARQL (manuellement ou via sparklis). Nous ne détaillerons pas la mise en œuvre de Fuseki car c'est un processus assez simple et bien documenté. Une fois le serveur Fuseki lancé, il ne nous reste plus qu'à lui donner un fichier (RDF/XML, N-Triples, ...) et le dataset qu'il contient sera disponible via un endpoint.

3.2 Création du modèle

Avant de nous lancer dans la compréhension de Jena, nous avons réfléchi et mis à plat les descriptions des différents objets ainsi que les relations entre eux. Pour cela, nous avons étudié le JSON de chaque objet, puis transcris les attributs qui semblaient intéressants ainsi que la façon dont étaient représentées les relations. Cela nous a permis d'avoir un aperçu de ce à quoi devrait ressembler notre base ainsi que d'étudier la structure du JSON, que nous devons bien comprendre pour l'utiliser dans le code Java.

3.2.1 Modèle utilisé

Tout d'abord, il y a les objets représentant les différents jeux vidéos. Pour cela nous avions besoin d'une classe représentant les jeux de manière globale, qui serait le type de chaque jeu. Un jeu aura aussi un nom et un id.

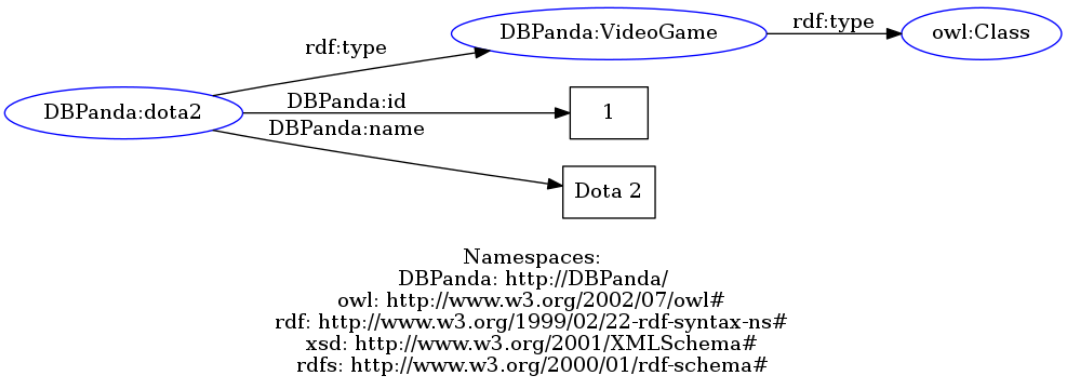

FIGURE 1 – Modélisation d'un jeu vidéo

Ces objets nous permettront de définir à quel jeu se rapporte chacun des objets créé par la suite. Nous avions au départ utilisé pour cela la notion de sous-classe, mais elle ne faisait pas de sens pour la plupart des objets, à l'exception des joueurs qui peuvent avoir des attributs spécifiques en fonction de leur jeu. Ainsi, un joueur possède la classe "joueur", mais aussi une sous-classe de celle-ci, spécifique à son jeu. Il possède aussi un nom et un id ainsi qu'une potentielle liste de matchs auxquels il a participé.

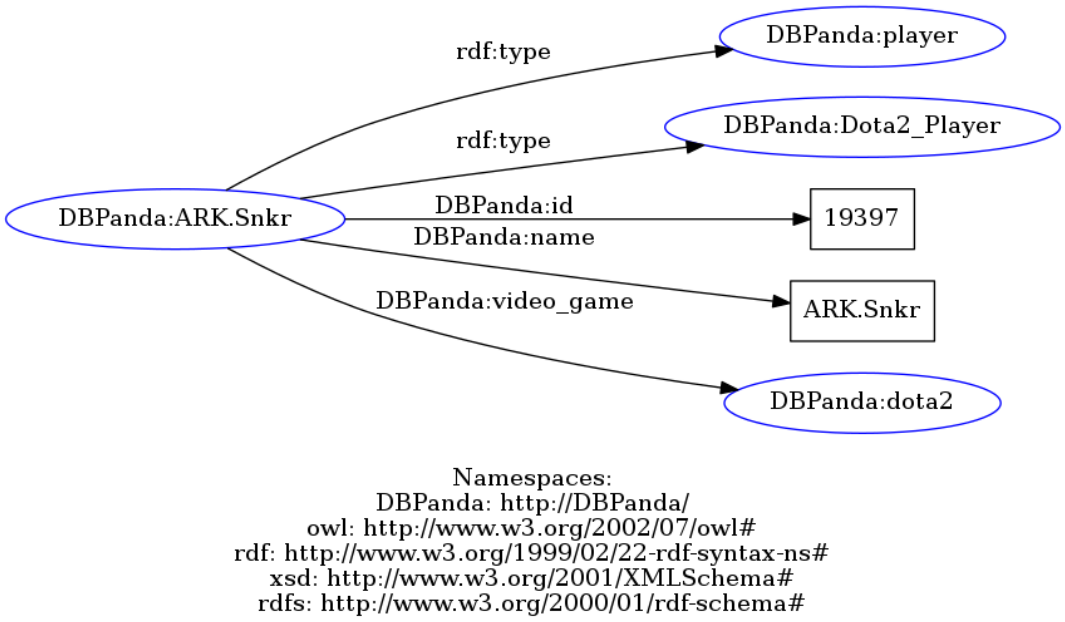

FIGURE 2 – Modélisation d'un joueur

Les équipes sont des regroupements de joueurs, elles possèdent donc une liste de joueurs, ainsi qu'une image représentant son logo et une collection de matchs auxquels elles ont participées.

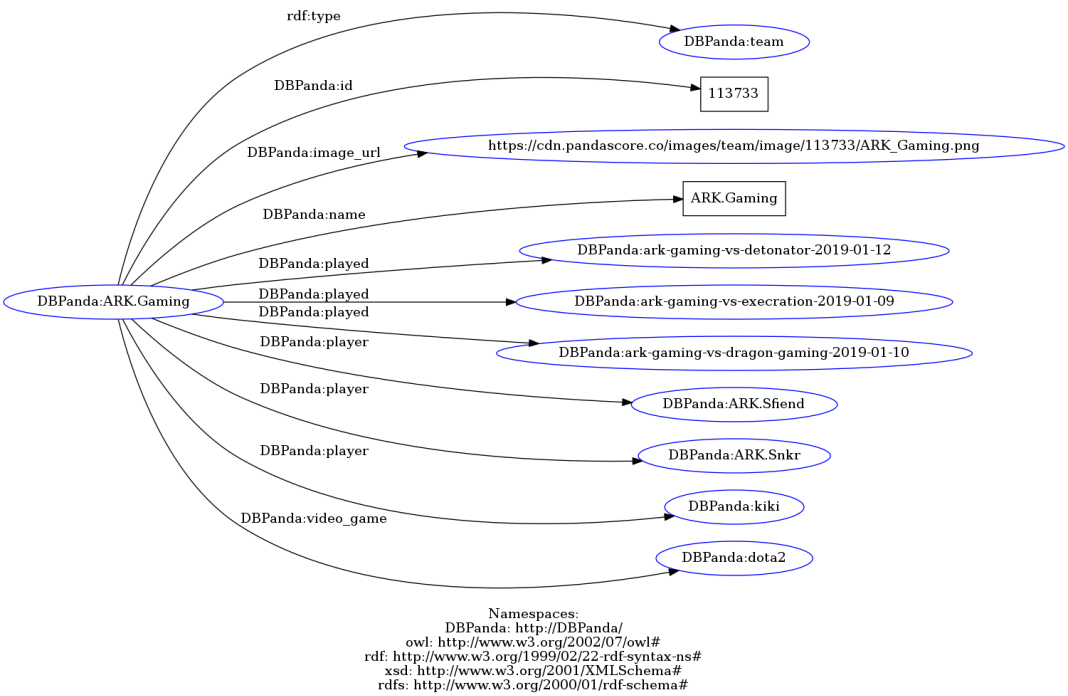

FIGURE 3 – Modélisation d'une équipe

Les matchs quant à eux possèdent une date (littéral typé *xsd :date*), le nom du match et le vainqueur lorsque celui-ci est terminé. Ils pointent également vers le tournoi auxquels ils appartiennent. La propriété *tournament* ayant comme propriété symétrique la propriété match, on peut récupérer la liste de tous les matchs d'un tournoi précis.

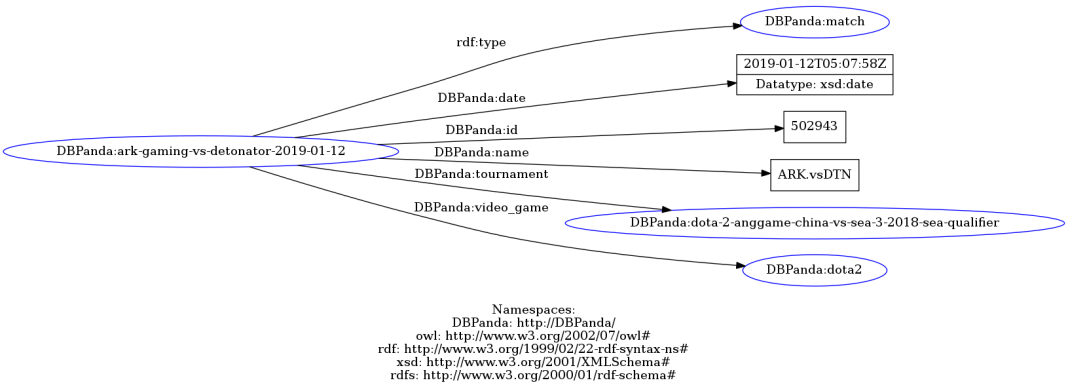

FIGURE 4 – Modélisation d'un match

Ainsi, en plus de la liste des matchs le composant, un tournoi possède un nom et un id, ainsi qu'un pointeur vers la série auquel il appartient. D'une façon similaire, nous avons la propriété symétrique nous permettant de lister les différents tournois d'une série. De plus, un tournoi possède un blank node représentant le "cashprize" gagné par le vainqueur. Ce blank node possède deux propriétés : le montant et la monnaie. Enfin, il peut posséder un vainqueur si celui-ci est terminé, pouvant être une équipe ou un joueur en fonction des règles de la compétition.

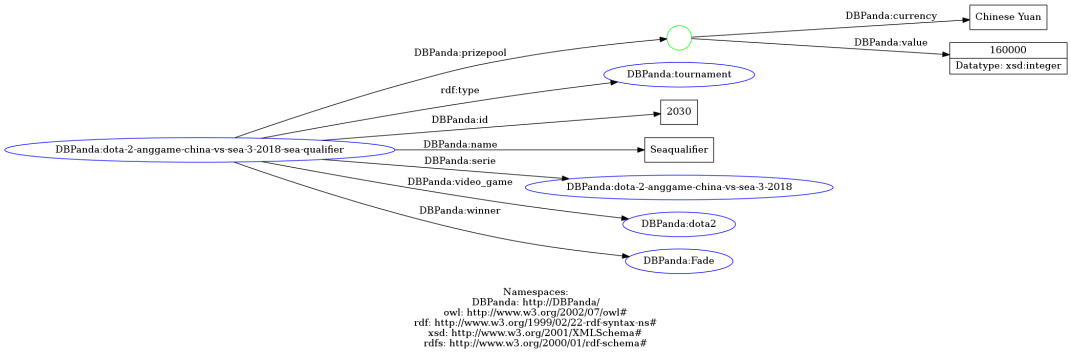

FIGURE 5 – Modélisation d'un tournoi

Une série est assez similaire, dans sa structure, à un tournoi, la seule différence est qu'elle ne pointe pas vers un tournoi mais vers la league à laquelle elle appartient.

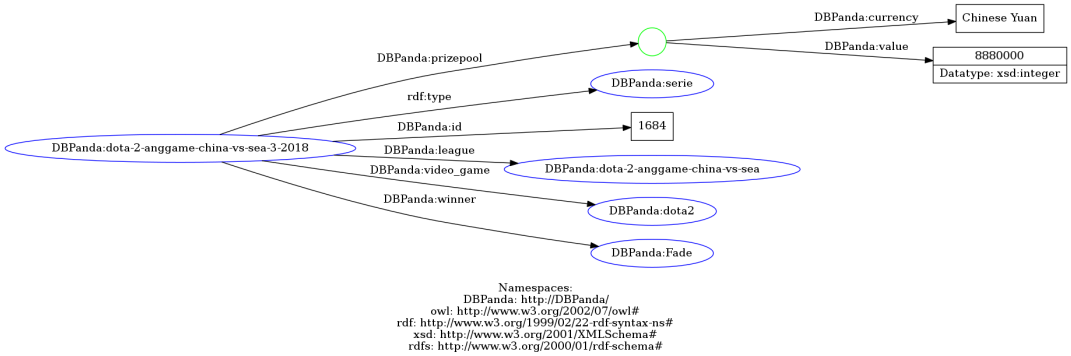

FIGURE 6 – Modélisation d'une série

Enfin, les leagues sont une collection de séries avec un nom, un id ainsi qu'une image représentant son logo.

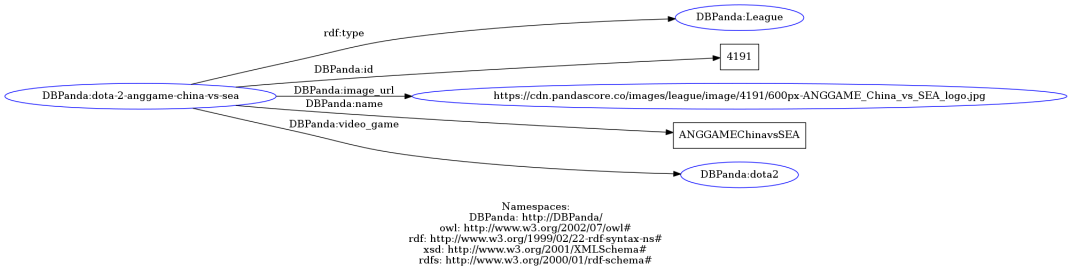

FIGURE 7 – Modélisation d'une league

Une fois la structure en tête, nous avons étudié la documentation de Jena afin de voir comment établir un lien entre nos fichiers JSON et l'API offerte par celle-ci. Nous avons donc parsés nos fichiers JSON pour récupérer les différents éléments et les injecter dans des objets Jena qui alimentent le modèle et nous permettent d'extraire le modèle créé sous le format souhaité.

3.2.2 Modélisation via Jena

Ainsi, nous avons utilisé divers éléments de l'API Jena afin de reproduire la modélisation de notre base :

- Un OntModel, qui représente un modèle ontologique complet auquel nous avons ajouter les divers triplets,
- Un préfixe pour le modèle afin d'avoir une structure moins verbeuse,
- Des OntClass, qui représentent la notion d'OWL class et permettent de structurer notre base en assignant une classe aux sujets principaux ainsi que de gérer la notion de sous-classe,
- Des ressources, qui correspondent aux objets (ou sujets) avec des URI,
- Des ObjectProperty, qui permettent de définir des prédicats avec des URI.

Ainsi, on peut créer des ressources dans notre OntModel, auquel nous ajouterons des ObjectProperty qui auront comme objet un littéral ou une autre ressource afin de créer des triplets. On peut donner les OntClass comme objet d'un triplet avec comme Property "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" afin de définir la classe de notre sujet.

Afin de symboliser les relations entre les sujets de notre base (e.g., les séries d'une league), on utilise la notion de "InverseProperty" afin que lorsqu'on ajoute les diverses séries comme appartenant à une league, la league soit ajoutée comme league de ces séries (e.g : Un joueur fait partie d'une équipe, mais une équipe possède des joueurs.)

Ainsi, nous pouvons maintenant modéliser les différents triplets en assignant à chaque propriété pour une ressource donnée un élément pouvant être :

- Un littéral (on ajoute juste un string comme valeur de la propriété)
- Un littéral typé (on utilise la fonction createTypedLiterral)
- Une autre ressource (en la récupérant ou en la créant)

Nous n'avions donc plus qu'à "parser" chaque fichier JSON et traduire les relations et propriétés présentes en du code Jena correspondant à ce que nous voulions comme représentation dans la base finale. Nous avons décidé, d'une manière analogique à la récupération des données, de traiter chaque objet et chaque jeu indépendamment les uns des autres (en mutualisant les fonctions tout de même) afin de pouvoir mettre à jour ou compléter une partie de la base de donnée plus facilement.

Chapitre 4

Exploration des données

4.1 Exploration SPARKLIS

Tout au long de la modélisation de notre base, nous pouvions extraire l'état actuel de cette dernière dans un fichier XML ou N-TRIPLES et la mettre à disposition via l'endpoint SPARQL. Nous pouvions donc tester en temps réel le rendu donné par SPARKLIS, ce qui nous a permis de debugger et de construire notre base correctement en vérifiant que les relations que nous définissions étaient bien représentées dans SPARKLIS. L'utilisation de cet outil nous a aussi a poussé à certaines méthodes de modélisation, par exemple, nous n'avons pas utilisé la notion de "bag" pour modéliser la liste des joueurs d'une équipe, car la visualisation sur SPARKLIS était moins intuitive qu'en assignant divers ressources au même couple (ressource, propriété). Enfin, l'utilisation de SPARKLIS nous a permis de parcourir notre base de façon intuitive et de créer des requêtes SPARQL utiles plus facilement.

4.2 Requêtes SPARQL

Nous avons écrit différentes requêtes SPARQL nous permettant de récupérer différentes informations :

— La somme totale des gains de chaque joueur :

```
SELECT DISTINCT ?player (SUM(?value) AS ?total)
WHERE { ?player a DBPanda:player .
        /* On sélectionne tous les joueurs */
        ?thing DBPanda:winner ?player .
        /* On récupère ce que le joueur a gagné (séries, tournois...) */
        ?thing DBPanda:prizepool ?prizepool .
        /* Ce que le joueur à gagné doit posséder un prizepool */
        ?prizepool DBPanda:currency "United States Dollar" .
```

```
        /* On fixe la monnaie du prizepool afin
        de ne pas sommer des valeurs dans différentes monnaies */
        ?prizepool DBPanda:value ?value }
        /* On récupère les valeurs qu'on sommera */
GROUP BY ?player
        /* On affiche les résultats pour chaque joueur */
```

— Les équipes qui ont joué un match dans une league précise (ici ESL-ONE) :

```
SELECT DISTINCT ?team
WHERE { ?team a DBPanda:team .
        /* On sélectionne les équipes */
        ?match a DBPanda:match .
        /* On sélectionne les matchs */
        ?match DBPanda:tournament ?tournament .
        /* Les matchs doivent avoir faire parti d'un tournoi */
        ?tournament DBPanda:serie ?serie .
        /* Le tournoi doit faire partie d'une série */
        ?serie DBPanda:league <http://DBPanda/esl-one> .
        /* La série doit faire partie de la league ESL-ONE */
        ?team DBPanda:played ?match . }
        /* On récupère toutes les équipes qui ont joué un de ces matchs */
```

— Les équipes multi gaming

```
SELECT DISTINCT ?team
WHERE { ?team a DBPanda:team .
        /* On sélectionne toutes les équipes */
        ?team DBPanda:video_game ?video_game_1 .
        /* On récupère les jeux des équipes (variable 1) */
        ?team DBPanda:video_game ?video_game_2 .
        /* On récupère les jeux des équipes (variable 2)*/
        FILTER(?video_game_1 != ?video_game_2) }
        /* On filtre de sorte à ce que le jeu 1 soit différent du jeu 2*/
```

— Les joueurs qui ont gagné un match depuis une certaine date (ici, le 1er janvier 2018)

```
SELECT DISTINCT ?player
WHERE { ?player a DBPanda:player .
        /* On selectionne les joueurs */
        ?match a DBPanda:match .
```

```
        /* On sélectionne les matchs */
        ?match DBPanda:winner ?player .
        /* On fait le lien entre les matchs et les vainqueurs */
        ?match DBPanda:date ?date .
        /* On récupère la date des matchs */
        FILTER ( (?date > "2018-01-01"^^xsd:date) ) }
        /* On conserve seulement les matchs qui ont eu lieu après la date
```

— Le classement des équipes (par nombre de match gagné)

```
SELECT DISTINCT ?team (COUNT(DISTINCT ?match) AS ?number_of_win)
/* On fait la somme des matchs gagnés */
WHERE { ?team a DBPanda:team .
        /* On récupère les équipes */
        ?match a DBPanda:match .
        /* On récupère les matchs */
        ?match DBPanda:winner ?team . }
        /* On fait le lien entre les matchs et les vainqueurs */
GROUP BY ?team
/* On affiche les résultats pour chaque équipe */
ORDER BY DESC(?number_of_win)
/* On trie par ordre décroissant pour avoir le classement */
```

Bien évidemment, on pourrait mélanger toutes ces requêtes pour avoir, par exemple, le classement des équipes sur une league, une série ou un jeu vidéo, les joueurs ayant gagné un match d'une certaine league depuis une certaine date, ...

Chapitre 5

Conclusion

Nous avons modélisé une base de connaissances sur le thème de l'eSport. Après avoir récupéré les données dans un format JSON à l'aide de l'API fourni par le site, nous avons créé une modélisation qui nous semblait pertinente puis nous avons retranscrit cette modélisation dans le langage utilisé par Jena afin de pouvoir générer une base de connaissance RDF à partir de ces fichiers JSON. Nous avons ensuite mis cette base de connaissance en ligne à l'aide de Fuseki afin de pouvoir la requêter grâce à SPARQL et l'explorer à l'aide de SPARKLIS.
Nous l'avons modélisé en prenant en compte le fait qu'elle puisse être par la suite agrandie et donc complétée. Pour cette raison, nous avons favorisé un traitement pour chaque jeux au lieu d'un traitement global. La structure des différentes parties du projet, à savoir l'extraction en JSON et sa conversion en schéma RDF, permet ainsi de faire évoluer et de mettre à jour le projet au besoin, afin de compléter les parties n'ayant pas été traitées par manque de temps ou de mettre à jour des valeurs obsolètes. Enfin, pour donner un ordre de grandeur de la base de connaissance générée, il est à noté que cette dernière contient 344069 triplets explicites (sans compter les relations liées aux propriétés inverses).