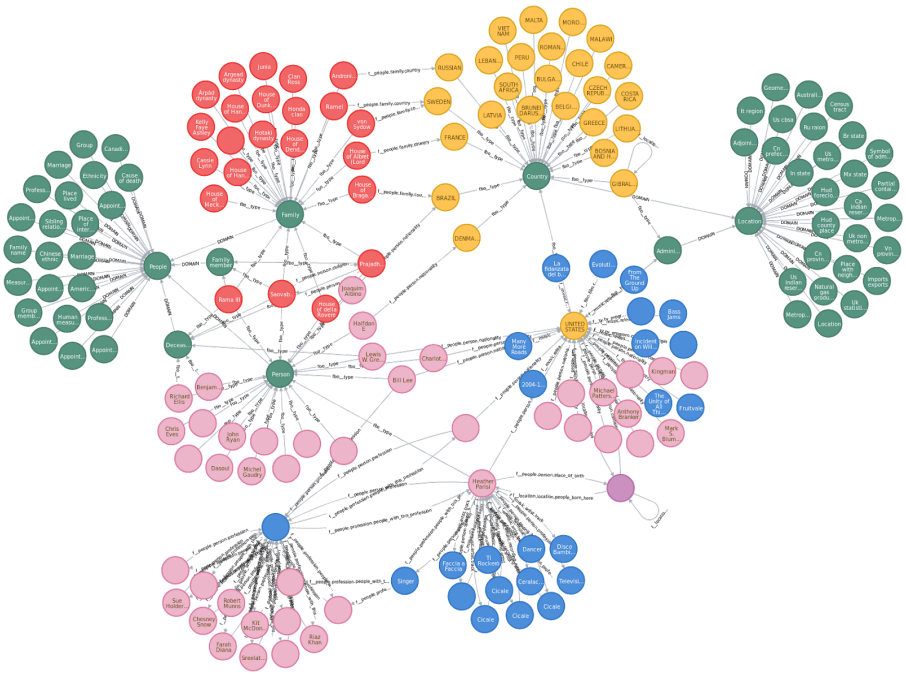
[Big Data ROMA TRE]

AA 2018-2019



Final project

Rebuilding a semantic Knowledge Graph for Link Prediction

bigdata-pals

Michele Ciaffarafà (#529060)

Lorenzo Guidaldi (#473014)

Codice del progetto: https://github.com/sullyD64/bigdata-2019/tree/master/project2

OBIETTIVI DEL PROGETTO

Il nostro progetto ha riguardato l'analisi dei dati e dello schema ontologico del Knowledge Graph **Freebase** a partire dal dump dei suoi dati disponibile in rete; lo scopo è stato quello di ricostruire un grafo semplificato che potesse essere sfruttato per costruire un benchmark per i sistemi di Link Prediction attualmente in ricerca e sviluppo presso il team del Dipartimento di Informatica di Roma Tre. Un ulteriore obiettivo è stato quello di testare una soluzione che consentisse ai ricercatori di esplorare il grafo in modo agile e di facile comprensione.

Per fare ciò abbiamo:

- Effettuato una pulizia e un partizionamento del dump, riducendone le dimensioni oltre il 70%.
- Costruito un'ontologia semplificata *by observation* sulle orme dello schema di Freebase.
- Convertito il dump RDF in un Labeled Property Graph e importato il tutto nel graph database **Neo4J**, grazie al quale abbiamo potuto effettuare query topologiche e semantiche per esplorare il grafo.

Il risultato del nostro lavoro, **FreebaseR3**, sarà a breve disponibile pubblicamente all'indirizzo https://freebase.inf.uniroma3.it e verrà messo a disposizione del team di Roma Tre.

Ci auguriamo che i nostri risultati possano essere d'aiuto!
bigdata-pals

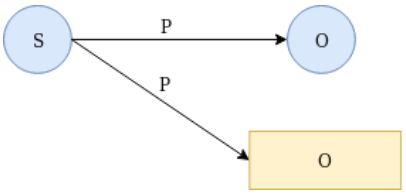
Ambiente di lavoro

Il processamento e lo storage dei risultati è avvenuto su una macchina virtuale Linux messaci a disposizione dall'ateneo. La macchina ha **8 core**, **64GB** di ram e **2TB** di archiviazione.

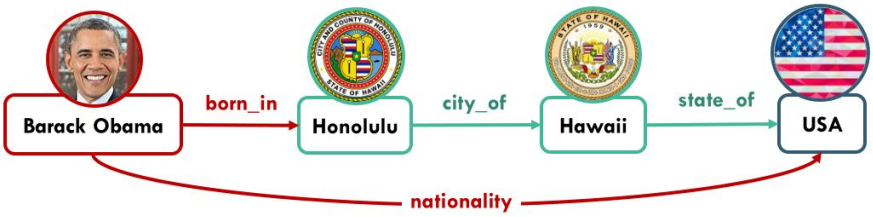
CONTESTO

1.1 Link Prediction su Knowledge Graphs

In un Knowledge Graph, i nodi rappresentano entità del mondo reale o concetti. La conoscenza è espressa mediante **fatti**, ovvero delle triplette che descrivono le connessioni tra le entità. Un fatto è una **tripla (S,P,O)** dove il predicato P può rappresentare una **relazione** tra due entità soggetto e oggetto, oppure un **attributo** dell'entità soggetto; in questo secondo caso, l'oggetto è il valore dell'attributo.



La **Link Prediction** è un task della **Knowledge Graph Augmentation**, una disciplina che si prefissa l'obiettivo di sfruttare le informazioni presenti nelle basi di conoscenza per inferire automaticamente nuova conoscenza sul dominio d'interesse. Nel caso della Link Prediction, questo si traduce nell'osservare il grafo per individuare pattern e predire i fatti mancanti (link) che abbiano maggiore probabilità di formarsi.



Esistono diversi approcci per questo task, tra tecniche supervised, statistiche e Random Walks; un approccio di recente interesse, descritto in diverse ricerche tra cui **TransE** [FB15k] ed esposto da **Andrea Rossi** presso uno dei seminari di AGIW, sfrutta tecniche di Machine Learning basate su Embeddings.

Per rappresentare un buon scenario modelli di Link Prediction, è fondamentale che un Knowledge Graph sia:

- Sufficientemente connesso e popolato.
- Incompleto, così da offrire scenari di previsione.
- Coerente, così che sia possibile giustificare i link predetti.

1.2 Freebase

Freebase è stato un collaborative Knowledge Graph. Creato nel 2007 da Metaweb, con l'intenzione di realizzare una "Wikipedia a grafo", esso consentiva ad utenti di registrarsi gratuitamente e aggiungere o modificare informazioni, creando collegamenti tra le entità e contribuendo alla costruzione di una base di conoscenza. La piattaforma metteva a disposizione un'interfaccia browser e diversi endpoint attraverso cui poter sottoporre query MQL, un linguaggio proprietario basato sullo standard SPARQL per l'interrogazione di basi di conoscenza RDF (Resource Description Framework).

Freebase rappresenta una versione preistorica del Google Knowledge Graph: nel 2010 il colosso di Mountain View ha infatti acquisito Metaweb e la gestione della piattaforma, per poi annunciare, nel 2014, un'interruzione graduale del servizio Freebase che si è concretizzata nel 2016. Nello stesso periodo, gran parte del contenuto e della community di Freebase sono confluiti in Wikidata, il collaborative graph di Wikipedia. Ad oggi, tutti i link a Freebase reindirizzano ad una pagina di Google Developers [**Goog**], dove è stato messo a disposizione uno snapshot della piattaforma in formato **RDF N-Triples**, racchiuso in un unico dump compresso di **32GB**. Il dump, una volta estratto, contiene la rappresentazione di circa **40 milioni di entità** e oltre **3 miliardi di fatti**, per un totale di **400GB**.

Freebase è un ottimo scenario d'impiego per modelli di Link Prediction: il dump contiene numerosi fatti sul mondo reale, di cui gran parte delle informazioni è incompleta (ad esempio, il 70% delle "persone" presenti in Freebase non hanno un luogo di nascita o altri dati anagrafici); inoltre, come vedremo, le entità sono suddivise in domini e classificate in tipi, ciascuno dei quali descrive un set di proprietà che un'entità di quel tipo può esprimere. È quindi presente una struttura che può essere impiegata per la validazione dei risultati ottenuti.

1.3 Schema di Freebase

Come in tutti i KG, anche in Freebase i fatti sono espressi mediante triple (S,P,O); la particolarità di questo sta nella forma assunta dal predicato. Un generico predicato - senza distinzione tra relazioni e attributi come nello standard RDF - è descritto da una struttura a tre livelli come segue [**fbschema**]:

```
<S>    /domain/type/property    <O>
```

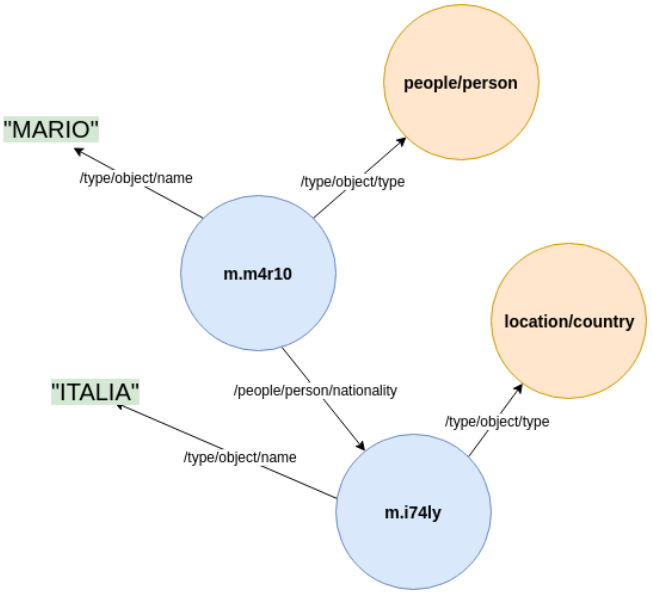
Un predicato identifica quindi una **property** che lega S e O e allo stesso tempo il **domain** e il **type** all'interno del quale ricade la property. Le entità in Freebase hanno quindi uno o più tipi, la cui appartenenza è espressa sia implicitamente (come abbiamo visto, usandone una property) che esplicitamente, legando l'entità che descrive

l'"istanza" a quella che descrive la "classe" mediante due property speciali ***/type/object/type*** (istanza -> classe) e ***/type/type/instance*** (la relazione inversa).

Tutti i nodi del grafo hanno come tipo */type/object,* mentre i nodi dell'ontologia sono istanze dei supernodi */type/type, /type/property, /type/domain*, e così via. Tipi e domini sono quindi delle collezioni categoriche di istanze, mentre le property sono il "vocabolario" di un dato tipo. Il valore di una proprietà è descritto dalla property */type/property/expected_type* e può essere un oggetto o un letterale (stringhe, interi, booleani, etc..).

A ciascuno nodo è associato un identificatore univoco su tutta la piattaforma detto **machine identifier** o **mid**, espresso nella forma /m/{alphanumeric} o /g/{alphanumeric} che, combinato con il prefisso del **namespace**, forma un URI di una risorsa; i nodi ontologici sono identificati conservando la struttura a tre livelli, quindi il nodo che descrive il tipo *person* avrà come ID la stringa *"/people/person*".

Nell'esempio in figura, il nodo m.m4r10 descrive una persona di nome Mario ed è connesso al nodo m.i74ly, che descrive una nazione (l'Italia) mediante la property /people/person/nationality, il che esprime che Mario è Italiano.



Il dominio **/type** racchiude tre informazioni fondamentali sullo schema:

- La descrizione di tipi, domini e property.
- Il **mapping** tra le istanze e i tipi.
- Informazioni generiche sugli oggetti, come il nome/label.

È importante notare come parte di queste informazioni può essere dedotta osservando la sola relazione che collega le due istanze, in quanto implica che m4r10 sia una people/person. Riprenderemo questo concetto più avanti.

Istanze e classi sono inoltre istanze di ***/common/topic*** e ci si riferisce spesso a loro come topic; un **topic** di Freebase è un'entità fisica, un elemento multimediale, una classificazione o concetto astratto; le informazioni legate ad esso, come *display name* e *description*, venivano usate per rappresentare le entità nella UI di Freebase.

Come in RDF, le relazioni che coinvolgono 3 o più entità sono realizzate frapponendo oggetti mediatori, che in Freebase vengono chiamati Compound Value Types (**CVT**). A differenza dei "blank nodes" RDF, però, questi oggetti sono anch'essi identificati da un URI e quindi vengono quindi considerate risorse. Inoltre, i CVT non sono */common/topic*.

1.4 Struttura del dump

Nel dump, le triple sono arrichite da prefissi che costruiscono URI per ciascuna risorsa. Gli URI fanno riferimento a diversi namespace RDF, tra cui quelli delle ontologie W3 e OWL, oltre ai namespace interni come *rdf.freebase.com/ns/* e *rdf.freebase.com/key/*. Per adattare i dati al formato N-Triples, sono stati adottati inoltre i seguenti cambiamenti:

- Tutti i mid e gli ID dello schema sono stati "appiattiti", sostituendo tutti i forward slash successivi al primo con dei punti. Così, */m/abc123* diventa */m.abc123* e */people/person* diventa */people.person*. In questo modo tutto lo schema è rappresentato sotto un unico namespace RDF.
- Ciascun elemento è racchiuso tra parentesi angolate; gli elementi sono delimitati da \t e ciascuna riga è terminata da un punto e accapo.
- I valori letterali delle proprietà attributo sono nella forma di tipi di dato.
- Le stringhe sono racchiuse tra doppi apici e seguite da "@" e un **language code** ISO (ad esempio en, en-gb, it..).
- A date e orari sono appesi URI dello schema XML di w3 (Custom Data Types RDF).

Una tripla del dump si presenta in questa forma:

```
<http://rdf.freebase.com/ns/g.11vjz1ynm>
<http://rdf.freebase.com/ns/measurement_unit.dated_percentage.date>
"2001-02"^^<http://www.w3.org/2001/XMLSchema#gYearMonth>  .
```

1.5 Distribuzione delle triple (domain slices)

Le informazioni di questo paragrafo provengono dal lavoro svolto da **Niel Chah** sull'esplorazione degli slice del dump [nchah] (documentata qui https://github.com/nchah/freebase-triples) e sull'esplorazione della sua ontologia [nchah2]. Chah afferma infatti che un possibile criterio per partizionare il dump in sottoinsiemi di triple consiste nell'estrarre le triple basandosi esclusivamente sul valore assunto dal predicato. Una **slice** è definita come il sottoinsieme di triple dove il predicato è parte di un dominio unico; quindi è possibile effettuare un partizionamento a tre diverse granularità (per dominio, tipo e property).

Partizionando il dump per domini, Chah ha potuto ricavare dati interessanti come la distribuzione percentuale delle triple, individuando inoltre tre categorie principali di slice:

- **Domini dell'implementazione di Freebase:** include le slice */common, /type, /key* e altre informazioni relative sia allo schema (come type) che all'implementazione

di alcune funzionalità della piattaforma, come la ricerca per chiave e la gestione dei link a risorsi esterni (*key*), l'implementazione dei merge (*dataword*) e controlli sulla consistenza (*freebase*). Solamente i domini */common* e */type* costituiscono oltre il 70% dei fatti del grafo.

- **Domini di OWL**: si tratta di slice per le singole property del namespace W3/OWL. Da notare come queste costituiscano una ridondanza con le informazioni contenute in /type: rdf:type coincide con /type/object/type, rdfs:label coincide con /type/object/name, etc.

- **Subject Matter Domains (SMD)**: include tutti i domini top-level su argomenti specifici, come i domini /people, /film, /music, ma anche /sports, /religion e /food. Questi domini contengono tutti i topic del mondo reale che rappresentano la "knowledge" nel KG di Freebase.

Table 3.: Freebase Domains

No.	Name	Domain	Triples	Total %	Group %
Freebase Implementation Domains					
1	common	/common/*	1,429,443,085	45.658%	58.507%
2	type	/type/*	788,652,672	25.191%	32.280%
3	key	/key/*	149,564,822	4.777%	6.122%
4	kg	/kg/*	30,689,453	0.980%	1.256%
5	base	/base/*	24,063,303	0.769%	0.985%
6	freebase	/freebase/*	11,259,415	0.360%	0.461%
7	dataworld	/dataworld/*	7,054,575	0.225%	0.289%
8	topic_server	/topic_server/*	1,010,720	0.032%	0.041%
9	user	/user/*	912,258	0.029%	0.037%
10	pipeline	/pipeline/*	547,896	0.018%	0.022%
11	kp_lw	/kp_lw/*	1,089	0.000%	0.000%
OWL Domains					
1	type	rdf-syntax-ns#type	266,321,867	8.507%	78.520%
2	label	rdf-schema#label	72,698,733	2.322%	21.434%
3	domain	rdf-schema#domain	71,338	0.002%	0.021%
4	range	rdf-schema#range	71,200	0.002%	0.021%
5	inverseOf	owl#inverseOf	12,108	0.000%	0.004%
Subject Matter Domains					
1	music	/music/*	209,244,812	6.684%	60.062%
2	film	/film/*	17,319,142	0.553%	4.971%
3	tv	/tv/*	16,375,388	0.523%	4.700%
4	location	/location/*	16,071,442	0.513%	4.613%
5	people	/people/*	15,936,253	0.509%	4.574%
6	measurement_unit	/measurement_unit/*	15,331,454	0.490%	4.401%
7	book	/book/*	13,627,947	0.435%	3.912%
8	media_common	/media_common/*	6,388,780	0.204%	1.834%

PROBLEMATICHE

Lo studio di Chah identifica delle ridondanze significative nel dump, oltre a provvedere importanti statistiche che ci aiutano a contraddistinguere quali parti di Freebase sono utili da considerare ai fini del nostro scopo, appurata in ogni caso la necessità di dover operare delle riduzioni sul contenuto e sul formato del dump.

Le problematiche principali affrontate nel progetto sono due:

1. **Identificare** quali **informazioni** filtrare e quali estrarre dal dump, inerentemente sia ai dati che dello schema.
2. **Individuare** la **tecnologia** migliore per memorizzare i dati estratti e per poter navigare il grafo sia topologicamente che semanticamente.

2.1 Problematiche del dump

Abbiamo compreso che gran parte della dimensione del dump è in realtà dovuta sia al formato scelto per rappresentare le informazioni, che a innumerevoli informazioni ad oggi prive di possibili applicazioni.

- Per prima cosa, ogni oggetto nel dump è codificato con un URL completo della corrispondente risorsa di Freebase. Non solo questa informazione non è più utile ad oggi - dal momento che il servizio è stato interrotto, tutte le URL sono invalide - ma ciò contribuisce anche ad aumentare notevolmente il peso in byte di ciascuna tripla.
- Osservando il dominio */common*, si nota inoltre la presenza di innumerevoli duplicati delle stesse triple localizzate in lingue diverse, una ridondanza inutile ai fini della leggibilità.
- Infine, alcuni domini consistono esclusivamente di metadati e formalismi che venivano impiegati per garantire la consistenza dei dati e imporre vincoli su ciò che si poteva inserire nel grafo.

Per comprendere al meglio l'utilità di questi metadati e dello schema, ci siamo documentati su altre ricerche condotte su Freebase in cui sono state semplificate le informazioni del dump.

L'approccio più diffuso (visto in [**FB15k**]) consiste nel selezionare un **sottoinsieme di entità** secondo qualche criterio, come ad esempio la centralità nel grafo o il ranking basato sui riferimenti a tali entità su siti-autorità esterne a Freebase; in modo simile, altri hanno attuato un **filtraggio mediante *regex*** della maggior parte dei domini meta e dello schema, **arricchendo** poi le entità del grafo con metriche calcolate durante

l'estrazione come grado in entrata/uscita e conteggio delle relazioni univoche (visto in [**Lissandrini**]).

Entrambi i lavori sopracitati hanno estratto dataset per l'addestramento di sistemi di link prediction e processamento di query, mentre altri studi sono stati compiuti con lo scopo di **ripristinare l'accesso e l'esplorazione dei contenuti** di Freebase, anche qui filtrando le informazioni irrilevanti e arricchendo semanticamente il grafo, ad esempio risolvendo le reificazioni sui mediatori e realizzando la chiusura transitiva (visto in [**fbez**])

Il nostro approccio

Da tutte queste esperienze, abbiamo compreso la necessità di pre-processare il dump per rimuovere le informazioni inutili. Per fare questo abbiamo dovuto studiarne a fondo la composizione ed esplorarlo per raffinamenti successivi al fine di estrarre i domini rilevanti e dividere il dump in **due parti**, una per le informazioni reali e una per le informazioni "meta" e tecniche.

A differenza di [**FB15k**] e [**Lissandrini**], abbiamo scelto di non andare in profondità in "un colpo solo" riguardo le informazioni semantiche da selezionare, optando per riservarci di volta in volta la possibilità di mantenere una **libreria** di fatti da cui poter attingere iterativamente le informazioni dello schema che meglio possono aiutarci a classificare le entità.

Di nuovo, grazie allo studio di Chah, abbiamo tuttavia compreso che gran parte di queste informazioni può essere dedotta direttamente dai predicati delle triple, e che quindi è possibile - dopo aver filtrato le ridondanze e ridotto la dimensione - produrre un'**ontologia** semplificata **"by observation"**, ovvero analizzando i predicati relativi alle istanze delle entità. In questo modo è possibile ottenere una descrizione solamente delle entità realmente contenute nel dump.

Tutto questo processo rappresenta un'inversione di tendenza: avremmo infatti potuto cercare di comprendere, a grana molto fine, quali predicati dello schema aggiungere alla nostra ontologia.

2.2 Scelte tecnologiche

Per l'esplorazione, inizialmente abbiamo valutato l'impiego di diversi triple store, tra cui Virtuoso DB. Su di esso, abbiamo formulato delle query iniziali in **SPARQL** sui dump esemplari [**FB15k**] e [**Lissandrini**], venendo da subito a contatto con la complessità del linguaggio, che male si sposa con la nostra necessità di un paradigma semplice. Abbiamo quindi usato utility standard Unix come **grep, sed, parallel, watch e nohup** per le successive esplorazioni dei dati grezzi dei dump e per la gestione dei job.

La pipeline di data cleaning è una serie di job **Spark** nella variante Python. Abbiamo usato la libreria **rdflib** di python per la validazione delle triple e per lo storage dei risultati intermedi in grafi nativi RDF.

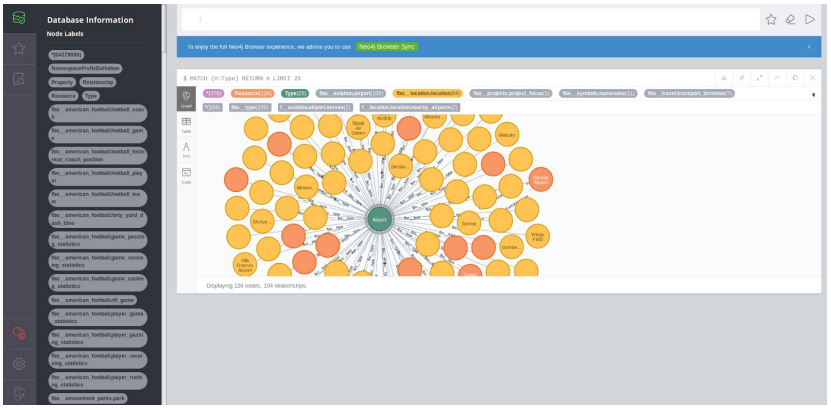
Storage e navigazione: Neo4J

Abbiamo scartato fin da subito l'ipotesi di un'archiviazione su ambiente distribuito HDFS, per la natura intrinseca a grafo del dataset. In generale, lo storage scalabile e il processamento di grosse moli di dati RDF non è un processo triviale e richiede particolare attenzione. Come detto, inoltre, l'esplorazione di un grafo RDF con SPARQL è poco intuitiva e non rende bene con query che vanno in profondità.

Avevamo bisogno di un'interfaccia per poter interrogare il grafo che supportasse un linguaggio di query semplice ma scalabile e che superasse i formalismi di SPARQL. Per questi motivi abbiamo scelto finalmente **Neo4J**, un graph database NoSQL, che offre allo stesso tempo uno storage scalabile e potenti strumenti di visualizzazione ed esplorazione.

Neo4J è un **labeled property graph**, i cui principali elementi sono **nodi**, **relazioni** e **proprietà**, che possono essere associate sia a nodi che relazioni. Nodi e relazioni hanno una struttura interna propria, che è determinata dalle coppie chiave-valore che, a differenza di RDF, rappresentano le proprietà delle entità stesse [**jbarrasa**].

Inoltre, i nodi sono "caratterizzati" da etichette (label) mentre le relazioni da tipi. Le etichette consentono di classificare le entità nel grafo e offrono uno strumento di ricerca che può essere sfruttato, in accoppiata a **cypher**, un linguaggio di query "visuale" molto intuitivo, per formulare ed eseguire query esplorative in modo molto più rapido e scalabile rispetto a SPARQL.



Questo rende l'approccio LPG molto più naturale e logicamente vicino al tipo di lettura umana, evitando l'impiego della reificazione delle relazioni complesse, workaround necessario impiegato nel modello RDF. I normali RDF stores sono inoltre fortemente

index-based, mentre in Neo4J le connessioni tra le entità e nodi vengono salvate su disco.

Questo significa che, quando eseguiamo query su Neo4j, possiamo andare molto in profondità con query semplici, cosa che risulta computazionalmente dispendiosa con un RDF store, la quale struttura ad indice rende molto difficoltose operazioni come l'analisi di path specifici. Neo4j risulta quindi essere un graph storage migliore per query profonde o query che comportino la ricerca di path specifici, ovvero ciò che fa al caso nostro in termini di navigabilità.

Conversione RDF->LPG

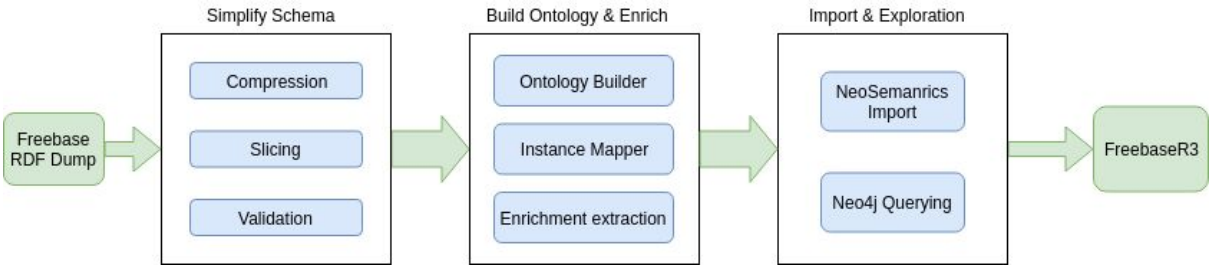
Per la conversione dei dump RDF in LPG e l'importazione in Neo4J, abbiamo scelto il plugin **Neosemantics** (https://github.com/jbarrasa/neosemantics). Si tratta di un insieme di stored procedures cypher che accettano RDF in diversi formati, tra cui N-Triples, e importano in un grafo Neo4J i nodi, gli archi e le label su entrambi.

Neosemantics consente anche l'esportazione dei risultati delle query in formato RDF. Per questo motivo, Neosemantics può essere usato per ottenere i dataset campioni per la link prediction una volta ottenuto il grafo a partire dalle triple.

SOLUZIONE

Data la natura del problema (dover gestire una grossa mole di dati centralizzata), abbiamo dovuto seguire un approccio iterativo, alternando le due fasi di pulizia/estrazione di informazioni e analisi dei risultati intermedi. In questo modo abbiamo potuto sfruttare le utility Unix per compiere diverse esplorazioni (conteggio delle triple con una data proprietà, ricerca di pattern, estrazione di tipi univoci), necessarie a raffinare meglio i nostri obiettivi.

Il risultato di questo processo è una pipeline composta da processamenti sequenziali inizialmente sul dump, poi su parti del dump. La pipeline consiste fondamentalmente di tre fasi:



1. **Semplificazione dello schema**

   Vogliamo ridurre, per prima cosa, la grandezza del dump eliminando rumore, rimuovendo i prefissi dalle triple (che sono una ridondanza non necessaria visto che Freebase non esiste più) e tagliando domini del dump non necessari alla navigazione (*compression*).

   Al fine di processare meglio le singole parti, l'idea è quella di suddividere i domini di schema da quelli del SMD (*slicing*) per poi validare le triple di ciascun dominio che andremo ad utilizzare successivamente (*validation*).

2. **Ricostruzione dell'ontologia**

   Partendo da SMD, intendiamo ricostruire l'ontologia del grafo inferendone le triple dall'osservazione dei predicati (***ontology builder***).

   Vogliamo, poi, collegare i nodi di SMD a quelli dell'ontologia (***instance mapper***) attraverso una ricostruzione fatta tramite l'osservazione del dominio **TYPE** di Freebase.

   Una volta ottenuto un grafo con fatti ed ontologia, vogliamo arricchirlo con ulteriori informazioni come nomi e descrizioni che possano migliorare la navigabilità del grafo e la leggibilità dei risultati delle query (***enrichment extraction***)

3. **Importazione finale ed esplorazione del grafo**

   Finiamo con il caricare le singole parti (SMD, ontologia, enrichment) su **Neo4j** attraverso il plugin **NeoSemantics**, andando a costituire **FreebaseR3.** Attraverso query cypher esemplificative valuteremo la qualità dei risultati ottenuti.

3.1 Semplificazione dello schema

> [S0] Filtraggio e riformattazione delle triple

La prima cosa da fare per gestire e manipolare il dump è quella di ridurne la grandezza, eliminare le molte ridondanze e l'enorme rumore introdotto dalle informazioni non utili alla costruzione del grafo, come quelle che servivano a dare consistenza alle scritture in Freebase quando era ancora attivo.

ANALISI

Per ridurre la grandezza prendiamo in considerazione i prefissi posti davanti agli elementi delle triple. Quando Freebase era in funzione costituivano, insieme alla URI associata, una URL collegata ad una pagina dedicata a spiegarne il significato. Essendo Freebase dismesso, sono una ridondanza non necessaria. Rimpiazzando, quindi, i prefissi con uno più di lunghezza minima, conserviamo lo standard RDF e riduciamo considerevolmente, già con questa operazione, la grandezza del dump più o meno della metà.

Altra ridondanza molto grande è dovuta alle localizzazioni. Molte informazioni sono spesso riportare in diverse lingue. Per la leggibilità a cui puntiamo ci basta la localizzazione in inglese. Scartiamo, quindi, le triple con localizzazione diversa.

Vogliamo, infine, tenere soltanto le triple relative a domini interessanti per la leggibilità del grafo, quelle contenenti informazioni relative alla realtà, ignorando il resto, costituito da dati di schema e metadati.

Del dominio **COMMON** teniamo la parte relativa ai tipi topic, document e notable_for; topic contiene, infatti, tutte le informazioni visualizzabili inerenti alle entità di **SMD**. Il tipo */common/document* contiene riferimenti ed informazioni meno strutturate che conserviamo per il futuro, ma che nella nostra esplorazione del grafo non ci serviranno. Il tipo */common/notable_for* contiene invece molti nodi informativi dal carattere di mediatori che vanno a reificare la proprietà di */common/topic*, */common/topic/notable_for*. Teniamo anche queste per il futuro, nel momento in cui si voglia implementare nella ricerca sulla *Link Prediction* degli embeddings semantici attraverso l'uso di tecniche di NLP.

Teniamo anche una piccolissima fetta del dominio **FREEBASE** che riporta informazioni sulle entità che rappresentano nodi mediatori nel grafo. Le altre triple che dobbiamo tenere sono quelle relative a **TYPE**, per la costruzione dell'ontologia e quelle facenti parte di **SMD**.

SVILUPPO

Per effettuare quanto riportato, abbiamo elaborato un job spark che prende in ingresso l'intero dump caricandolo come ***rdd***.

Attraverso una *filter* basata su pattern regex scarta le triple che non ci interessano ed con una successiva *map* rimpiazza i prefissi degli elementi delle triple con un prefisso di lunghezza minima come segue:

<http://rdf.freebase.com/ns/m.11vjz1ynm>

viene ridotto a

<f:m.11vjz1ynm>

Di seguito vediamo il *pattern regex* con le regole corrispondenti a ciascun dominio e alle triple particolari che vogliamo escludere come quelle, ad esempio, che presentano oggetti con prefisso url diverso da quello di Freebase. Tutte le regole sono concatenate e poste in OR, in un'unico pattern.

```
SKIP_PATTERNS = r"\"@(?!en)|" \
    + r"\"@en-|" \
    + r"/common\.(?!topic|document|notable_for)|" \
    + r"\t<http:\/\/www\.w3\.org[^>]*>\t(?!\.)|" \
    + r"/base\.|" \
    + r"/freebase\.(?!type_hints)|" \
    + r"/dataworld\.|" \
    + r"/user\.|" \
    + r"/pipeline\.|" \
    + r"/kp_lw\.|" \
    + r"/help\.|" \
    + r"/usergroup\.|" \
    + r"/community\.|" \
    + r"/atom\.|" \
    + r"\t<http://(?!rdf)[^>]*>\t\."
```

CONCLUSIONI

Input: l'intero **dump di Freebase**, un pattern regex per la filter, il prefisso url da rimpiazzare ed il nuovo prefisso.

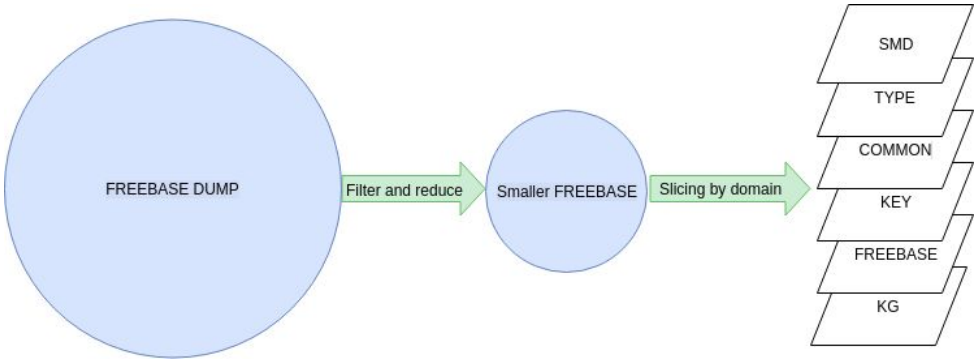
Output: una versione RDF NTriples ripulita e compressa di Freebase.

Il job è composto da una filter (che impiega la regex di cui sopra) ed una map che rimpiazza i prefissi, comprimendo la lunghezza delle triple.

L'intero job ha impiegato **1 ora e 56 minuti** ed ha ridotto la dimensione del dump del **76%**, riducendolo da **397GB** a **95GB**.

> [S1] Estrazione degli slice dei domini

Una volta ridotte le dimensioni del dump in maniera considerevole, la necessità è stata quella di suddividere l'RDF così ottenuto separando lo *schema* dal *subject matter domain* in modo da poter lavorare agevolmente sulle singole parti in seguito.



ANALISI

Per separare i domini di schema da SMD dobbiamo passare in rassegna tutte le triple del dump uscente da s0 e decidere, per ognuna, il dominio di appartenenza dal predicato, distinguendo tra i domini di schema rimasti dopo s0 (TYPE, COMMON, KEY, FREEBASE, KG) e SMD. Farlo attraverso Spark non è immediato se l'obiettivo è quello di generare, durante un unico scorrimento del dump, 6 differenti RDF rispettivamente rappresentanti i domini di schema ed SMD. Dopo aver valutato varie opzioni implementative, tra cui anche quella di un semplice script sequenziale, abbiamo deciso di trattare il problema come un problema di estrazione.

SVILUPPO

Per estrarre i domini di schema ed SMD abbiamo costruito un job che esegue 6 diverse *filter*, utilizzando 6 diversi pattern regex, sul dump uscente da s0. Scorriamo quindi, idealmente, 6 volte il dump, ma, considerato l'hardware a disposizione e l'utilizzo della memoria di Spark, risulta comunque la soluzione più efficiente in termini di tempo di esecuzione.

Di seguito riportiamo i pattern regex impiegati per estrarre ciascun dominio dal dump in output da s0:

```
PATTERN_SMD = r"\t<(k:|f:(common|type|freebase|kg))[^>]*>\t(?!\.)"
PATTERN_KEY = r"\t<k:[^>]*>\t(?!\.)"
PATTERN_COMMON = r"\t<f:common[^>]*>\t(?!\.)"
```

```
PATTERN_TYPE = r"\t<f:type[^>]*>\t(?!\.)"
PATTERN_FREEBASE = r"\t<f:freebase[^>]*>\t(?!\.)"
PATTERN_KG = r"\t<f:kg[^>]*>\t(?!\.)"
```

CONCLUSIONI

Input: Il dump di Freebase post-s0 ed un pattern regex per ogni dominio da estrarre

Output: slice per i domini TYPE, COMMON, KEY, FREEBASE, KG, SMD

Il job ha impiegato **2 ore e 28 minuti** per il completamento. Le dimensioni dei singoli slice sono riportate di seguito:

TYPE	**39GB**	FREEBASE	**3MB**
COMMON	**26GB**	KG	**2GB**
KEY	**8GB**	SMD	**21GB**

> [S2] Validazione delle triple

Dopo il passo s1 abbiamo notato che alcune triple erano mal formattate e non rispettavano lo standard RDF; si tratta di errori presenti nel dump originale di Freebase, che a prima vista ci erano sfuggiti. Occorre che le triple rispettino gli standard RDF NTriples imposti dal W3C, poiché dovremo importarle all'interno di Neo4j tramite NeoSemantics, il quale effettua un check su ciascuna tripla durante l'importazione, bloccando l'operazione qualora venga riscontrata la mancata adesione allo standard.

Abbiamo, quindi, fatto ricorso ad una libreria python costruita per la gestione degli rdf, ***rdflib***. Con questa, abbiamo scritto una funzione di validazione chiamata da un job Spark con una semplice *filter*.

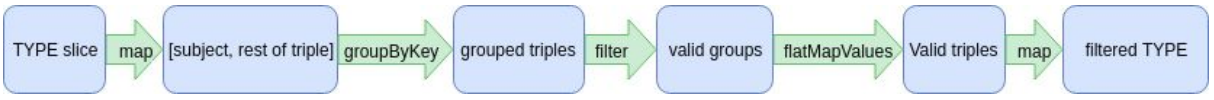
CONCLUSIONI

Input: Qualsiasi file in formato RDF
Output: Il corrispondente file senza le triple non valide
Il job ha validato la suddivisione **SMD** in **1h 33'**, **COMMON** in **1h e 18'** e **TYPE** in **1h 45'**.

> [S3] Filtraggio delle informazioni rumorose da TYPE

Per rendere il dominio /type il più possibile snello per le successive lavorazioni abbiamo deciso di filtrare i type custom di Freebase inerenti all'ontologia di /user e /base. I tipi corrispondenti a questi domini introducono infatti un sacco di rumore nel dump.

Per rimuovere queste triple, con spark, eseguiamo una *map* sul soggetto della tripla ed una *groupByKey*. Controlliamo, poi, con una *filter* se il soggetto è un *mid* e tra le triple ne è presente una che ha come oggetto una stringa che inizia per "/base/" o "/user/". In caso affermativo, l'intero blocco inerente a quel soggetto viene filtrato.
Successivamente, con una *flatMapValues ed* una *map* riportiamo alla forma standard le triple dei vari raggruppamenti per soggetto.



CONCLUSIONI

Input: slice **TYPE** post- s1

Output: slice **TYPE** ottimizzato

Il job ha impiegato **40 minuti** e ha ridotto del **5,4%** lo slice TYPE, riducendolo da **39GB** a **37GB**.

3.2 Ricostruzione dell'ontologia e arricchimento

> [S4] Estrazione di una reference per il mapping tipi-instanze

A questo punto abbiamo la slice SMD sul grafo separata dalla slice TYPE. L'informazione che realizza il mapping tra le istanze e la loro descrizione è contenuta in TYPE, nelle triple con predicato **type.type.instance** e **type.object.type.** Il primo connette i topic type/type alle loro istanze, mentre il secondo connette le istanze ai predicati.

Per evitare ridondanze inutili, ci basta estrarre uno solo di questi predicati, perciò l'obiettivo preliminare è stato comprendere che tipo di relazione ci fosse tra i due.

ANALISI

Già contando le triple di entrambi abbiamo scoperto essere molte di più le triple del primo (per abbrev. TTI, 186 milioni ovvero quasi il 31% di TYPE) rispetto al secondo (TOT, 37 milioni ovvero il 6%), quindi non può esservi una relazione 1:1.

Abbiamo poi estratto le liste di tipi univoci menzionati da entrambi. Confrontando queste liste abbiamo scoperto che TTI menziona 1373 tipi in più rispetto a TOT, che non ne menziona alcuno che non sia già in TTI, quindi, dal punto di vista **ontologico** TOT è un sottoinsieme di TTI.

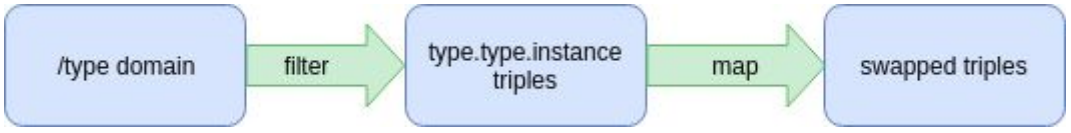
Restava da comprendere se potesse esistere una tripla menzionata da TOT per la quale non esiste la menzione inversa in TTI. L'unico modo per verificare ciò è realizzando l'unione insiemistica.

Usando **rdflib**, abbiamo creato due Graph persistenti (usando Berkeley DB) contenenti i due set di triple; dopodichè abbiamo rovesciato le triple di TOT (l'insieme più piccolo) invertendo il soggetto con l'oggetto e trasformando type.object.type in type.type.instance e abbiamo unito i due grafi ignorando le triple duplicate e contando l'eventuale incremento del numero di triple nel grafo TTI.

SVILUPPO

Non abbiamo riscontrato un incremento nelle dimensioni di TTI, traendo la conclusione che TTI include le relazioni inverse espresse da TOT, pertanto possiamo estrarre il mapping tra istanze e tipi interamente dai predicati TTI.

Per farlo abbiamo realizzato un job che esegue un semplice filter attraverso regex per poi invertire di nuovo, con una map, soggetto e oggetto come espressi in TOT. Abbiamo scelto un nuovo predicato custom **<fbo:type>** per esprimere il mapping e per individuare questa relazione nel nostro grafo.



CONCLUSIONI

Input: slice TYPE post-s3

Output: set di triple <f:mid> <fbo:type> <fbo:domain.type> che chiamiamo **type instance map reference** (TIM_reference)

Il job ha impiegato tempo impiegato **11'** per scorrere /type ed estrarre **100261301 triple** con predicato *type.type.instance*, escluse quelle di /common che non ci servono per collegare le istanze di smd all'ontologia.

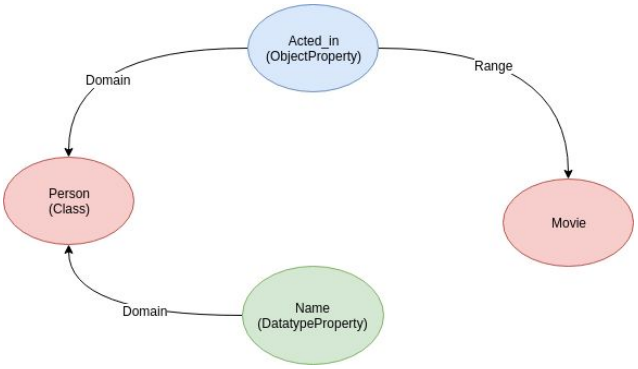
> [S5-0] Ricostruzione dell'ontologia *by observation*

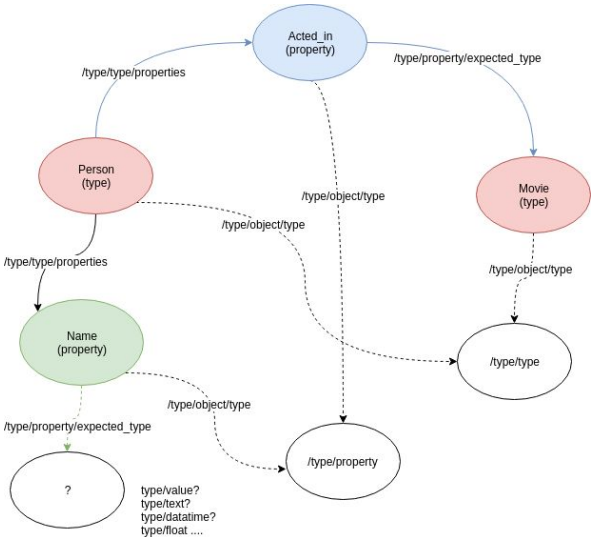
A questo punto abbiamo tutto il necessario per ricostruire lo schema strettamente necessario ad SMD inferendolo dall'osservazione dei predicati delle sue triple. L'idea è quella di generare un RDF contenente triple valide che vadano ad individuare le sole relazioni ontologiche necessarie all'esplorazione semantica di SMD.

ANALISI

La struttura del predicato di Freebase, costruita sui 3 livelli **domain/type/property** permette di osservare il dominio ed il tipo del soggetto di una tripla ma non dell'oggetto. Esiste la possibilità, infatti, che alcuni oggetti delle triple di SMD possano rappresentare entità senza archi uscenti (nodo pozzo, se interpretati su un grafo), ovvero, non si presentino mai come soggetti di una tripla, rendendone impossibile l'inferenza del tipo attraverso l'osservazione del predicato. Escludiamo, quindi, per adesso, di generare le triple che rappresentino il vero e proprio collegamento tra il piano delle entità reali ed il piano ontologico.

Limitiamoci, quindi, alla generazione di un RDF (seguendo lo standard W3C accettato da NeoSemantics) che contenga le triple rappresentative dell'ontologia relativa a SMD.





Osservando, ad esempio, una tripla come quella che segue:

```
</m/m4r10> </people/person/nationality> </m/i74ly>
```

Inferiamo le seguenti triple ontologiche dalla lettura del predicato:

- Una tripla per definire un nodo *tipo*:
  `<fbo:people.person> <rdf:type> <rdfs:class>`
- Una tripla per definire un nodo *dominio*:
  `<fbo:people> <rdf:type> <rdfs:class>`
- Una tripla per definire la relazione tra i due nodi precedenti:
  `<fbo:people.person> <rdf:subclassof> <fbo:people>`
- Una tripla per definire i nodi *proprietà*:
  `<fbo:people.person.nationality> <rdf:type> <owl:objectproperty>`
- Una tripla per collegare la *proprietà* al *tipo*:
  `<fbo:people.person.nationality> <rdfs:domain> <fbo:people.person>`
- Tre triple per dare un nome letterale ai nodi *dominio*, *tipo*, *proprietà*:
  `<fbo:people> <rdfs:label> "People"`
  `<fbo:people.person> <rdfs:label> "Person"`
  `<fbo:people.person.nationality> <rdfs:label> "Nationality"`

SVILUPPO

Abbiamo costruito un nuovo job che, per prima cosa, effettua una *map,* trasformando ogni tripla nella corrispondente lista di triple elencate sopra. Subito dopo effettuiamo una *flatMap* che restituisce un rdd avente una tripla ontologica per riga. Tante di queste potrebbero essere duplicate perché inferite da triple di SMD con lo stesso predicato, quindi, come ultima cosa effettuiamo una *distinct.*

CONCLUSIONI

Input: slice SMD post-s3

Output: set di triple che definiscono l'ontologia di SMD

Il job ha generato l'ontologia in **1 ora e 5 minuti** processando circa **90000 triple/s**.

Il risultato è un RDF di **23 milioni** di triple.

> [S5-1] Ricostruzione del mapping tipi-istanze *mixed*

Abbiamo ricostruito le triple dell'ontologia nel passo precedente. Adesso dobbiamo collegare le entità ontologiche alle istanze di SMD creando un set di triple sulla base di TIM_reference (s4)

ANALISI

La prima necessità è quella di comprendere quali delle entità di **SMD** compaiono soltanto come oggetti delle triple in modo da cercarne nella **TIM_reference** possibili associazioni con l'ontologia. Una volta rilevate queste associazioni possiamo fare la stessa cosa per i soggetti di **SMD** per poi unire le due elaborazioni.

In questo modo avremo generato un set di triple contenente tutte le corrispondenze tra ogni entità menzionata in **SMD** e ogni tipo ad esse corrispondente presente tra le relazioni di Freebase contenute in **TIM_reference**.

SVILUPPO

Il job prende in ingresso **SMD** e ne ricava due liste separate delle entità menzionate come soggetti (**SMD_left**) e come oggetti (**SMD_right**). Per estrarre SMD_left effettua una *map* tra soggetto ed il tipo inferito dal predicato per poi effettuare una *distinct*.

Per estrarre SMD_right viene eseguita una *filter* per escludere le triple che hanno un *literal* come oggetto, una *map* tra entità oggetto della tripla ed un valore nullo, che servirà successivamente come discrimine per la *join*, ed una *distinct*.

Una riga di SMD_left si presenta come di seguito:

[entità1, dominio1/tipo1]
[entità2, dominio2/tipo2]

mentre una di SMD_right avrà:

[entità1, None]
[entità3, None]

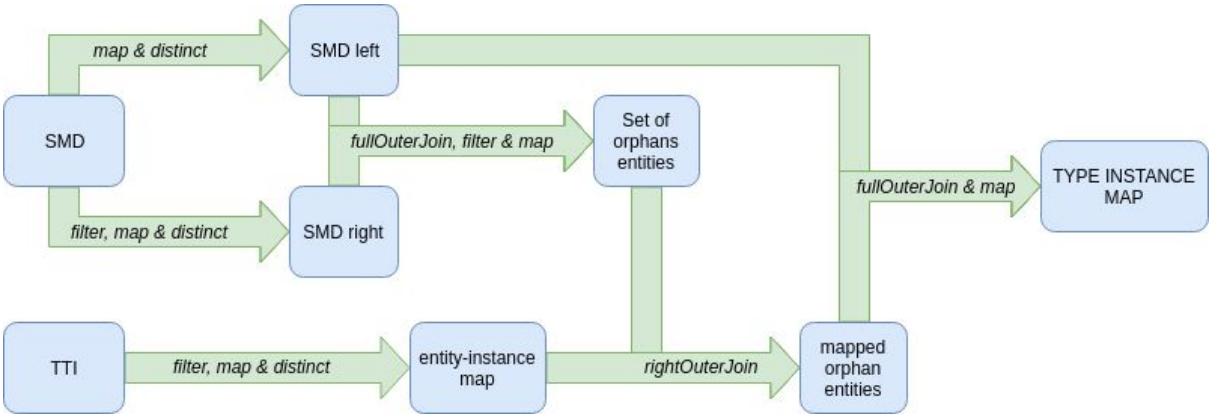
I due rdd così ottenuti vengono uniti attraverso una *fullOuterJoin.* Il risultato sarà un rdd le cui righe avranno una forma del tipo:

[entità1, (dominio1/tipo1, None)]
[entità2, (dominio1/tipo1, None)]
[entità1, (dominio/tipo, None)]
[entità3, (None, None)]

Le righe che presentano un doppio valore nullo sono le entità *orfane* di SMD, *nodi pozzo* nel grafo, per i quali vogliamo ricercare una corrispondenza con un tipo all'interno di TIM_reference. Le estraiamo attraverso una *filter* seguita da una *map* tra entità ed un nuovo valore nullo.

Proseguiamo quindi con l'elaborare TIM_reference attraverso una *filter* che prenda soltanto le triple con un oggetto a 2 livelli (che esprime un associazione ad un tipo) per poi eseguire una *map* tra entità ed il tipo associato; finiamo con una *distinct* per rimuovere possibili duplicati. Il set di triple così ottenuto rappresenta l'intera mappatura tra SMD e l'ontologia (**entity-instance map**). A noi interessa soltanto per risalire alle associazioni dei nodi pozzo; ricostruiremo le restanti associazioni da SMD_left.

Attraverso una *rightJoin* tra la **entity-instance map** ed il **set di entità orfane** otteniamo le associazioni cercate, quelle prive di doppio valore nullo.



Per finire, attraverso un *fullOuterJoin* con SMD_left seguito da una *flatMap* otteniamo l'intero set delle associazioni tra tutte le entità di SMD e la sua ontologia, definendo sia una tripla che definisce la relazione tra entità reale ed entità ontologica, che una che rappresenta il tipo come label per l'entità. Per esempio, per la tripla

```
<entità1> <domain/type/property> <entità2>
```

verranno generate le triple:

```
<entità1> <fbo:type> <dominio.tipo>
<entità1> <rdfs:label> "dominio/tipo"
```

CONCLUSIONI

Input: slice **SMD** post-s3 e **TIM_reference**

Output: set di triple che rappresentano le relazioni tra SMD e la sua ontologia. Type-Instance Map (**TIM**)

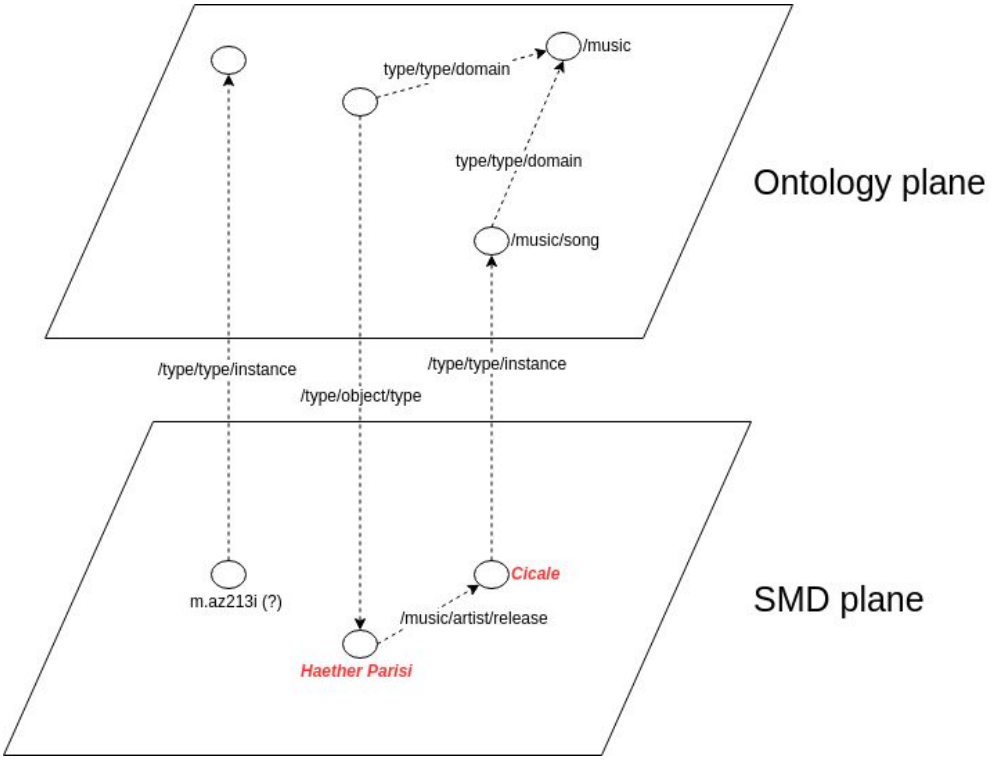
Il job ha impiegato **42 minuti** per giungere a termine.

TIM ha un peso di **10GB.**

Analizzando i risultati intermedi del job abbiamo visto che in SMD esistono **823807** *entità orfane*. Di queste, siamo riusciti a ricostruirne una relazione con l'ontologia in **570503 casi.**

> [S5-2] Estrazione di set di arricchimento *by mid*

A questo punto abbiamo SMD, la relativa ontologia ed il mapping tra le entità dei due piani. Dobbiamo arricchire i nodi con le altre informazioni presenti nei vari slice di schema per rendere leggibile ed esplorabile il grafo.



ANALISI

Per rispondere a questa necessità occorre elaborare un processo di natura generale che permetta di associare alle entità di SMD o dell'ontologia, le informazioni corrispondenti alle stesse entità in altri slice. L'idea è, quindi, quella di cercare corrispondenza per ogni soggetto di ogni tripla di un primo RDF (chiamiamolo **Data**) in un secondo RDF (chiamiamolo **EXT**) filtrato secondo il tipo di informazione che si intende cercare.

Implementare questa procedura significa eseguire un *nested-loop* su file molto grandi e di difficile gestione; spark ci viene incontro nuovamente con la funzione *join*.

Procediamo quindi come segue:

- Ricaviamo da **Data** una lista dei soggetti distinti.
- Da **EXT**, estraiamo un set di triple con predicato corrispondente all'informazione con la quale vogliamo arricchire **Data.**
- Effettuiamo una *join* sui soggetti tra le due elaborazioni.

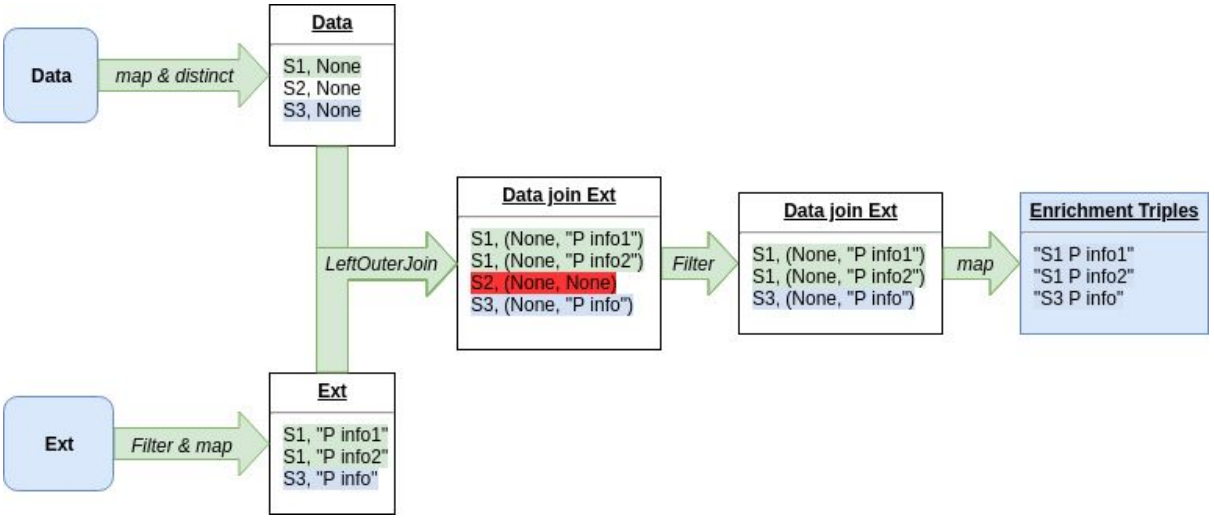
Così facendo otteniamo un set di triple che vanno ad arricchire le entità del primo RDF.

SVILUPPO

Il job Spark che abbiamo implementato processa, per prima cosa, **Data.** Effettua su questo una *map* tra soggetto della tripla ed un valore nullo (questa struttura ci servirà, più avanti, per l'operazione di *join*). Successivamente esegue una *distinct* per ottenere la lista delle entità di **Data**.

Su **Ext**, invece, viene eseguita una *filter* utilizzando un *pattern regex* per estrarre le triple corrispondenti alla proprietà di predicato ricercata. Successivamente viene effettuata una *map* tra soggetto e resto della tripla.

A questo punto viene eseguita una *leftOuterJoin* tra **Data** ed **Ext.** Se un'entità di Data non è presente tra quelle di di Ext, la riga ottenuta dalla join avrà un doppio valore nullo. Per ricostruire, quindi, le triple corrette, eseguiamo una *filter* atta a scartare le righe con doppio valore nullo per poi eseguire una *map* che concatena la chiave della riga (il soggetto proveniente da **Data**) con il resto della tripla proveniente da **Ext**.



Abbiamo impiegato il job per estrarre da **TYPE** le triple corrispondenti ai *name* delle entità di SMD e per estrarre gli *expected value* delle property dell'ontologia.

CONCLUSIONI

Input: Due file RDF (**Data** ed **Ext**) ed il predicato con cui filtrare **Ext**

Output: set di triple che costituiscono l'arricchimento per **Data**

Il job, fatto girare per estrarre i *name* dal dominio **TYPE** ha impiegato **22 minuti**, generando quasi **39 milioni di triple** che rappresentano le associazioni tra le entità di SMD ed i loro nomi.

Per estrarre gli *expected_value* delle proprietà dell'ontologia da **TYPE** ci sono voluti soltanto **8 minuti**, estraendo **6017 triple.**

3.3 Importazione ed esplorazione su Neo4J

> Importazione di SMD

Abbiamo importato SMD in Neo4j attraverso il plugin NeoSemantics.

La procedura consiste in un singolo comando, *importRDF*, che prende in input un file RDF di cui possiamo specificare il formato (*NTriples* nel nostro caso).

Le regole di mapping applicate da NeoSemantics durante l'import sono molto semplici. Per ciascuna tripla (S,P,O) letta:

1. Crea un nodo S con label :Resource con la proprietà uri, contenente l'uri che identifica S.
2. Controlla l'oggetto O della tripla:
    a. Se O è un letterale, aggiungi a S la prioprietà P, con valore O
    b. Se O è un URI, crea il nodo O con label :Resource con la proprieta uri valorizzata come sopra, e connetti il nodo S al nodo O mediante una relazione di tipo P.

La procedura standard di import presenta inoltre una funzione di aggiunta di label personalizzate, qualora sia presente una tripla (S, rdf:type, O), dove rdf:type è il predicato standard http://www.w3.org/2000/01/rdf-schema#type. Ad una tripla così fatta, il mapping associa la label :O al nodo S.

SMD ha un peso di **21GB** ed è composto da più quasi **359 milioni** di triple.
L'intero processo ha richiesto **12 ore e 5 minuti** per arrivare a conclusione.

2019-09-22 04:03:06.481+0000 INFO [o.n.k.i.p.Procedures] Import complete: **348549038** triples ingested out of **348549038** parsed

L'RDF di partenza è stato convertito in un LPG avente **64 milioni** di nodi e **244 milioni** di relazioni.
Considerato che il numero di triple dell'RDF abbiamo dedotto facilmente che ciascun nodo abbia una media di **5,4** tra relazioni e proprietà con il solo subject matter domain.
Abbiamo quindi così un grafo piuttosto informativo e denso considerato il suo volume, il che lo rende perfetto per l'esplorazione tramite LPG.

> Importazione dell'ontologia

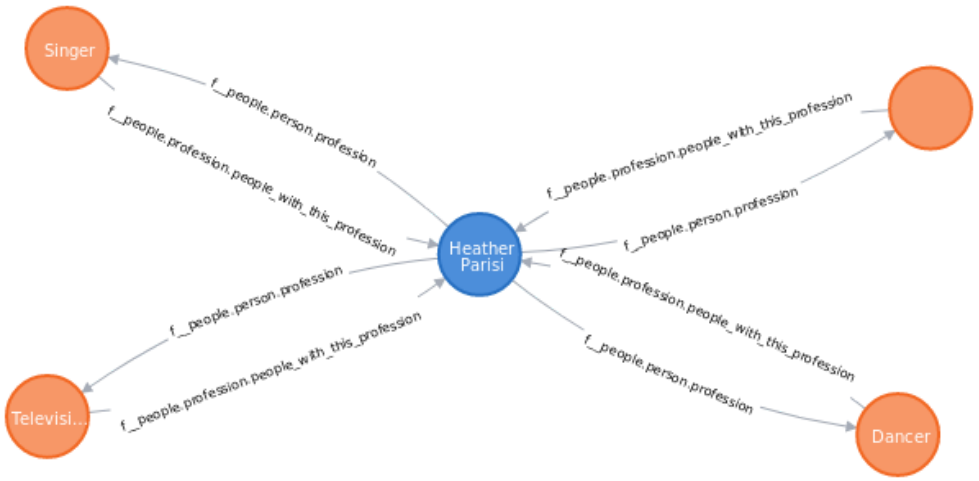
L'esplorazione del grafo costruito in questo modo manca però dei contenuti informativi e di un modo che ne permetta la navigazione semantica attraverso query e visualmente. Il passo successivo è stato, quindi, quello di importare in Neo4j l'ontologia ricostruita nel passo **S5.** Per farlo abbiamo utilizzato la procedura *importOntology* di NeoSemantics, che associa ai nodi importati in questo modo delle classi identificative differenti da quelle impiegate per i nodi inerenti alle normali entità, consentendo una lettura del grafo migliore e generando internamente una struttura indice per query semantiche formulate topologicamente.
Successivamente abbiamo importato attraverso *importRDF* il mapping tra SMD ed ontologia e le informazioni di arricchimento che vanno a dare leggibilità al grafo, ovvero, i nomi delle entità e gli expected value delle proprietà nell'ontologia.

> Esplorazione del grafo definitivo

Di seguito valutiamo con qualche query la leggibilità di **FreebaseR3** osservandone, allo stesso tempo, la coerenza delle informazioni. Il dominio più ampio di SMD è **/music.**

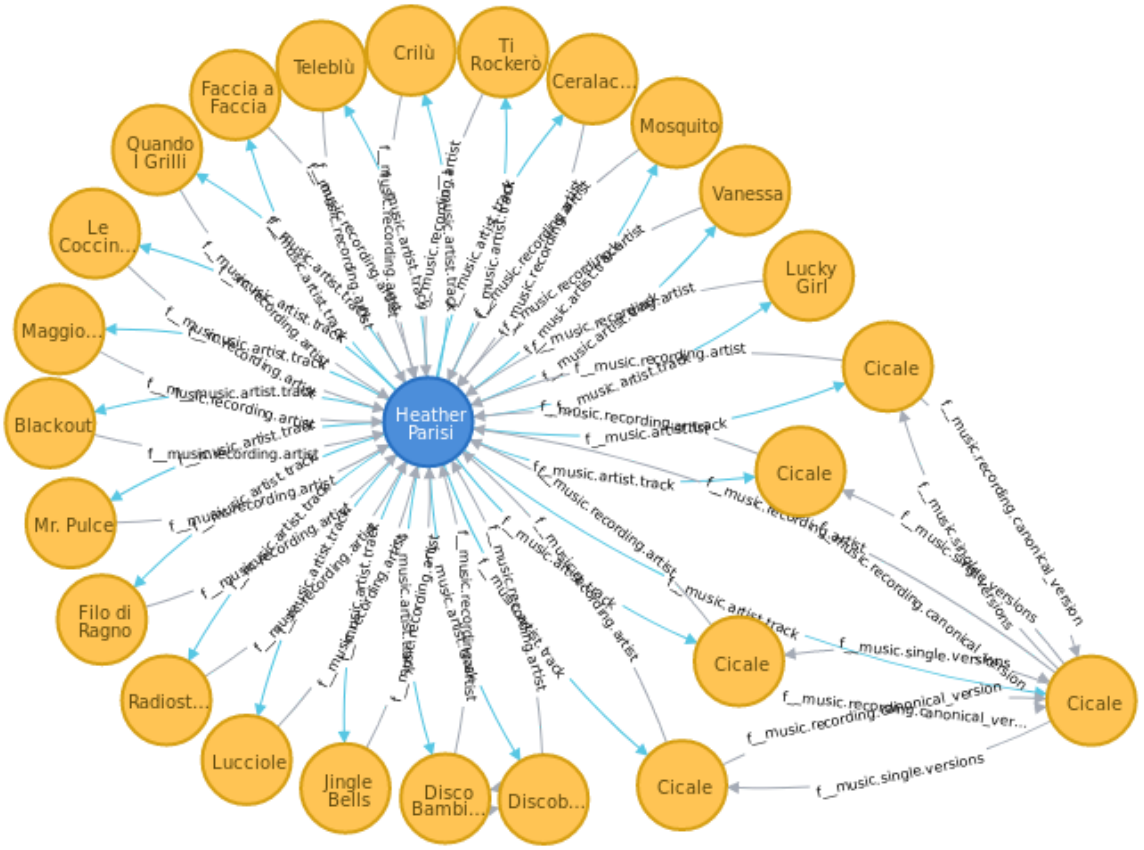
Vediamo qualche query riguardante questo dominio:

```
MATCH (n:fbo__music.artist {fbo__name:"Heather Parisi"})
-[r:f__people.person.profession]->(b) RETURN n,r,b
```



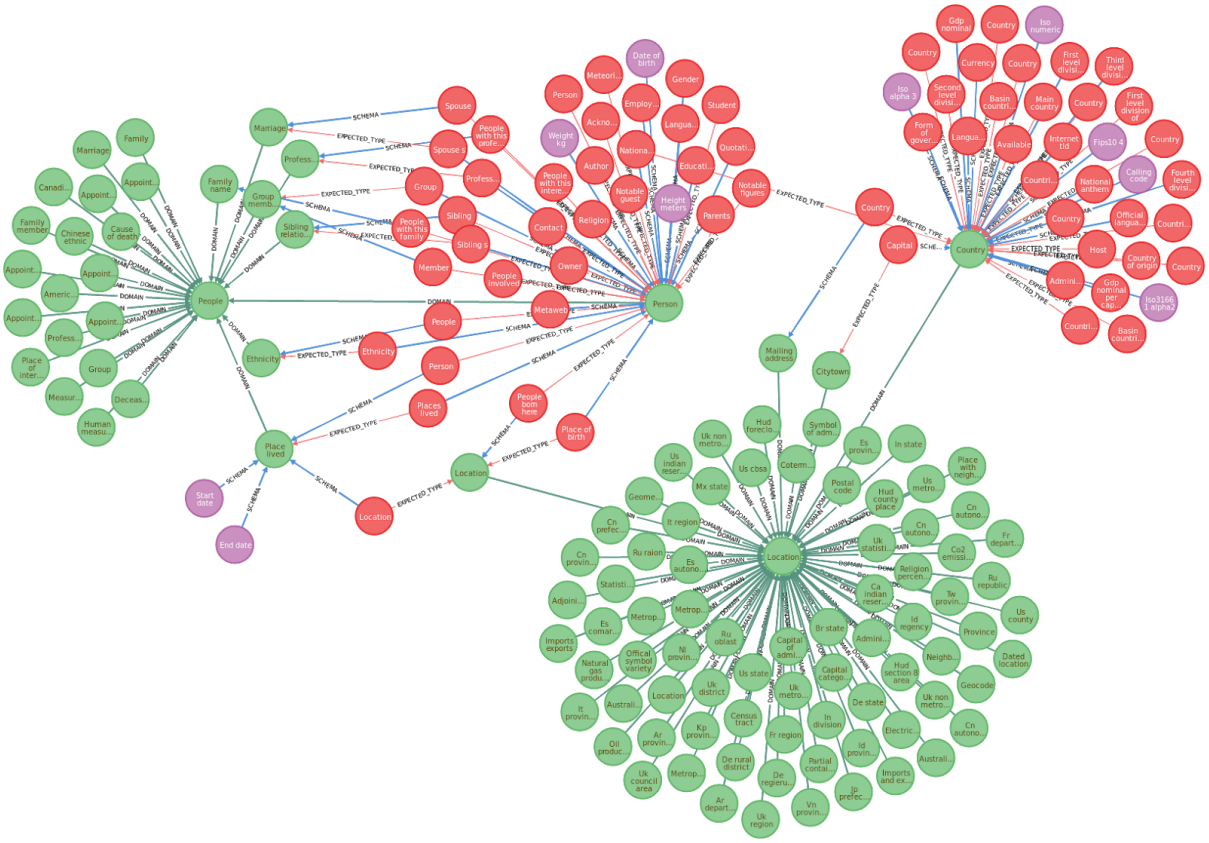
Attraverso la query *cypher* chiediamo a FreebaseR3 di ricercare tutti i nodi di tipo *artisti musicali* (fbo__music.artist) che abbiano come proprietà il nome “Heather Parisi” ed i nodi collegati a questo da una relazione di tipo *professione* (f__people.person.profession).

```
MATCH (n:fbo__music.artist {fbo__name:"Heather
Parisi"})-[r:f__music.artist.track]->(b) RETURN n,r,b
```

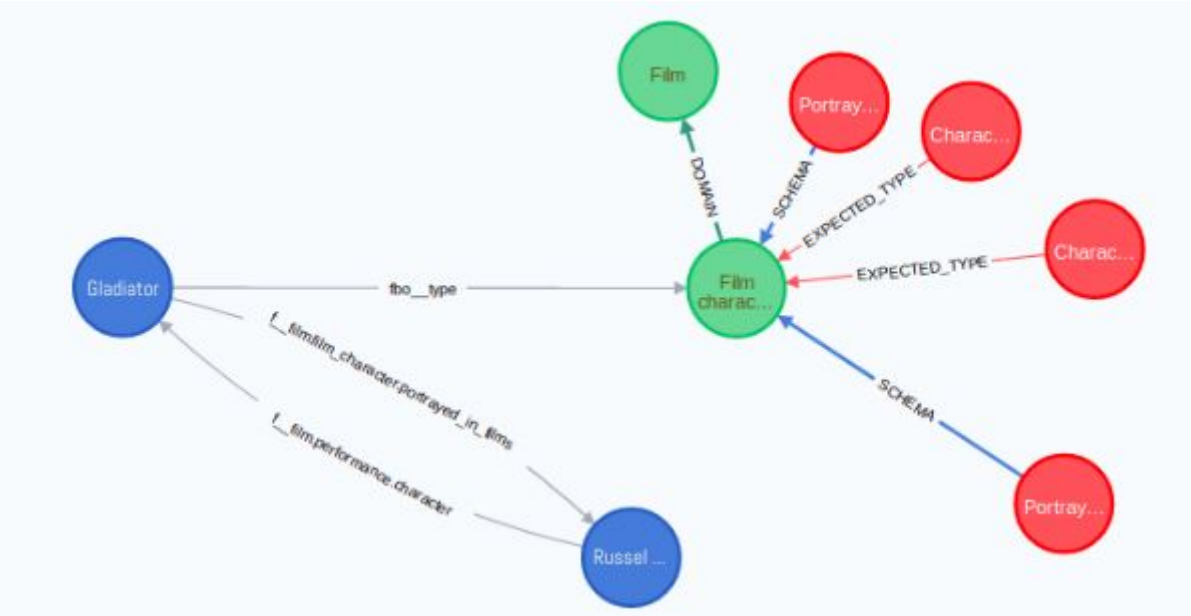


Come la query precedente, ma questa volta ricerchiamo le relazioni *tracce scritte* (f__music.artist.track). Osserviamo come il singolo più famoso di Heather Parisi, *Cicale*, compaia più volte, come traccia di un album e come singolo. Questo è un esempio di ridondanza di informazione; esistono in Freebase molte versioni della stessa entità che si potrebbe pensare di identificare in futuro.

Passando all'ontologia, esploriamo il grafo dal nodo Relationship "nationality". Neo4j permette di esplorare manualmente il grafo espandendo le relazioni dei singoli nodi. Sfruttando questa possibilità, abbiamo espanso i tipi *people.person* e *location.country*, per risalire poi ai domini *people* e *location*, individuando un sottografo ontologico interessante. Con altre espansioni abbiamo, poi, evidenziato le relazioni tra le *ObjectProperties*.



Di seguito vediamo invece come ontologia ed entità reali coesistano sul grafo:



Attraverso questo tipo di esplorazione è possibile costruire in maniera agevole dei sottografi di Freebase la cui coerenza ed integrità sia guidata dalla semantica. Individuare, ad esempio, un sottografo che includa tutto il dominio **/people**, può portare ad avere un dataset coerente e valido nell'ambito della *Link Prediction*, non dovendosi preoccupare dell'eterogeneità dei fatti riportati, fattore normalmente penalizzante.

I dataset richiesti dai sistemi di *Link Prediction* sono solitamente in formato RDF Ntriples. NeoSemantics contiene una procedura per l'export in questo formato, ultimo tassello a renderlo ideale per possibili studi futuri in questo campo.

CONSIDERAZIONI FINALI

Abbiamo visto l'intero processo che ci ha portato a costruire **FreebaseR3**.

Allo stato attuale, lo strumento che abbiamo creato può consentire al team di ricerca di RomaTre di estrarre dataset molto validi per la Link Prediction.

Rimane irrisolta la questione dei nodi CVT (*compound value type*); attualmente questi sono nodi, senza la proprietà name, che gestiscono relazioni reificate. Una soluzione può essere quella di far collassare la reificazione su un singolo arco e nodo di destinazione, sfruttando le proprietà di Neo4j in quanto *labeled property graph*. Il tutto può essere implementato attraverso elaborazioni del tutto simili a quelle presentate in questo studio.

L'impiego di spark, preceduto da una profonda analisi del problema, si è rivelato essere vincente grazie alla sua ottimizzazione della memoria; altri metodi non ci avrebbero, probabilmente, permesso delle elaborazioni tanto efficienti. Un altro approccio poteva essere quello di caricare su un *RDF triple store* l'intero dump di Freebase grezzo ed estrarre i vari domini, dopo aver superato eventuali problemi di ingestion. Tuttavia, anche in questo caso, sarebbero state necessarie successive elaborazioni di riduzione e adattamento per l'ingestion in Neo4j. Affrontare l'implementazione dell'intera pipeline facendo ricorso a job spark sequenziali ci ha permesso, inoltre, di scomporre il problema e di elaborare procedure che potrebbero tornare utili in futuro. Prendiamo, per esempio, l'estrazione delle triple di arricchimento del grafo; possiamo pensare, ad esempio, di estrarre le descrizioni dei singoli *topics* dal dominio COMMON, in modo da avere un campo di testo su ciascun nodo che potrebbe essere impiegato per creare embeddings semantici attraverso tecniche di *NLP*. Questo potrebbe permettere alle attuali tecniche di *Link Prediction* di essere guidate da qualcosa di molto più vicino al nostro modo di pensare e di interpretare, piuttosto che dalla sola topologia.

Neo4j, allo stesso modo, si è prestato perfettamente allo scopo del progetto, essendo in grado di scalare molto bene. Si può arrivare molto in profondità con query molto semplici e attraverso *cypher* possiamo effettuare query semantiche in maniera

topologica, vista la struttura di FreebaseR3. La navigazione visuale rende la comprensione del risultato semplice ed immediata.

I risultati di questo studio verranno impiegati per estrarre sottografi per l'addestramento di diversi sistemi di Link Prediction già addestrati con FB15K in un successivo progetto di studio. L'idea è quella di ricavare delle *ground truth* interpretabili in modo da permettere un miglior tuning di questi sistemi ed una migliore analisi dei risultati.

FONTI

1. [**Goog**] Google, Freebase data dumps
   https://developers.google.com/freebase
2. [**fbschema**] Jamie Taylor, Freebase schema
   https://www.slideshare.net/jamietaylor/freebase-schema
3. [**nchah**] Niel Chah, Freebase-triples: A Methodology for Processing the Freebase Data Dumps
   https://arxiv.org/pdf/1712.08707.pdf
4. [**nchah2**] Niel Chah, OK Google, What Is Your Ontology?
   https://arxiv.org/pdf/1805.03885.pdf
5. [**FB15k**] Bordes et al., TransE
   http://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf
6. [**Lissandrini**] Matteo Lissandrini, Freebase-ExQ
   http://people.cs.aau.dk/~matteo//notes/freebase-data-dump.html
7. [**FBez**] Bast et al,. Freebase-Easy (Easy access to the Freebase Dataset)
   http://ad-publications.informatik.uni-freiburg.de/WWW_FreebaseEasy_BBBH_2014.pdf
8. [**NSMNTX**] Jesús Barrasa, Neosemantics
   http://jbarrasa.github.io/neosemantics
9. [**jbarrasa1**] Jesús Barrasa, RDF Triple Stores vs. Labeled Property Graphs: What's the Difference?
   https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/#RDF-Labeled-Graph-Model