UNIVERSITÀ
DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in Informatica

Elaborato finale

Titolo

Sottotitolo (alcune volte lungo - opzionale)

Prof. Bouquet Paolo

Corte Pause Manuela

Anno accademico 2021/2022

Indice

Sommario

I dati sono diventati una parte sempre più fondamentale della nostra vita e ancor più di quella delle aziende nell'assisterle nel processo di decision-making. Questi dati, provenienti da molteplici fonti come: social network, online tracking, sensori di IoT, . . . , risultano in una mole enorme, non strutturati e di conseguenza complessa da sfruttare per ricavarne informazioni rilevanti. Proprio per questo il mondo della data integration è così importante [17].

Possiamo definire la data integration come il problema del combinare dati provenienti da livelli diversi e fornire all'utente finale una visione unificata di questi [8]. E' facile vedere quindi come questo concetto si adatti bene in un'azienda che utilizza vari sistemi, applicazioni e piattaforme, ognuna delle quali produce o raccoglie dati senza tenere conto degli altri applicativi, creando quelli che vengono definiti data silos.

Esistono approcci diversi alla data integration; quello più tradizionale è certamente quello dei data warehouse, dove tutti i dati sono combinati e memorizzati in un solo posto (tipicamente un database). Questo processo di "combinazione" è definito ETL (Extract, Transform, Load) e permette di rilevare e correggere inconsistenze tra i dati prima che questi vengano uniti. Permette inoltre di integrare tipi di dati eterogenei e visualizzarli poi come un'unica collezione complessiva.

Questa soluzione risulta complessa da utilizzare per dataset che vengono modificati frequentemente e richiedono quindi che il processo di ETL, che è molto costoso, venga re-eseguito molte volte. Anche per questo si è quindi passati ad un paradigma basato sul *loose coupling* ovvero è presente un'interfaccia sulla quale eseguire query che vengono poi mappate ed eseguite sulle sorgenti originali eliminando il problema dell'avere informazioni non aggiornate [16]. Questi due approcci sono descritti in figura 1.

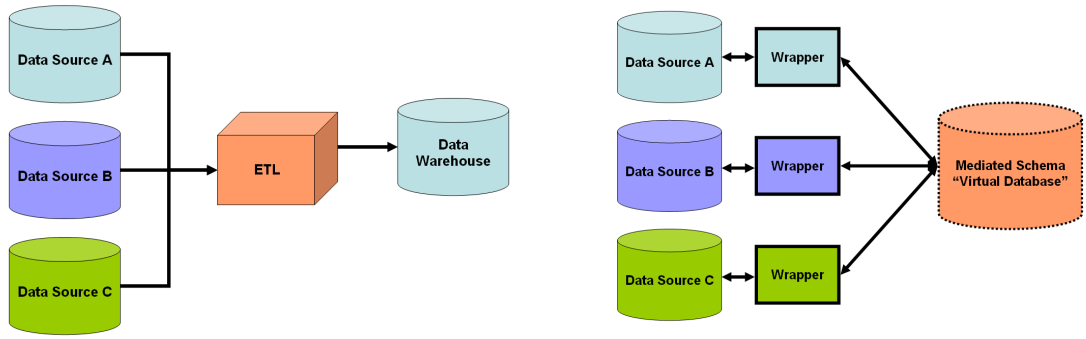

Figura 1. Approcci alla data integration

Indipendentemente dall'architettura del sistema, l'aspetto semantico risulta molto importante al fine di evitare la collisione tra termini uguali usati all'interno delle sorgenti con significati diversi. È proprio da questa considerazione che nascono approcci basati su ontologie definiti come Ontology Based Data Access (OBDA). Queste soluzioni mitigano il problema appena descritto fornendo un vocabolario comune da utilizzare, presentano però il problema di essere complesse e di conseguenza poco fruibili da un ampio pubblico.

Lo scopo di questo elaborato è, per prima cosa, quello di descrivere i principali concetti teorici alla base delle ontologie e dei Knowledge Graph ed in particolare del Virtual Knowledge Graph

system Ontop. Una volta stabilite delle basi teoriche comuni verrà descritta l'esperienza di tirocinio da me svolta presso l'azienda Ontopic al fine di sviluppare un strumento che permetta l'interrogazione di un ontologia tramite strumenti di business intelligence come Tableau. In questo modo l'ontologia può essere interrogata anche da persone non esperte del campo dato che le query possono essere scritte in SQL o addirittura tramite l'interfaccia grafica fornita dagli strumenti di business intelligence.

1 Knowledge Graphs

1.1 Cos'è un Knowledge Graph

Le prime pubblicazioni in ambito di rappresentazione della conoscenza tramite l'aiuto di knowledge base (basi di conoscenza) risalgono alla fine degli anni '50 e nel 1980 ricercatori dell'università di Groningen e dell'università di Twente nei Paesi Bassi usarono per la prima volta il termine Knowledge Graph per descrivere il loro sistema basato sull'integrazione di molteplici sorgenti di dati per rappresentare il linguaggio naturale tramite una knowledge base. Questo primo momento di ricerca iniziale fu poi seguito, all'inizio degli anni 2000, dall'affermarsi del Semantic Web e degli standard W3C, come RDF e OWL ad esso associati, nonché al sorgere di varie ontologie pubbliche come DBPedia, YAGO e Freebase. [4] [6]

Il termine Knowledge Graph viene però diffuso solo nel 2012 da Google che introduce il termine per descrivere il potenziamento tramite tecniche semantiche del proprio motore di ricerca ovvero: le ricerche che vengono effettuate non sono più semplicemente string matching, ma viene aggiunta una componente di ragionamento in grado di riconoscere veri e propri "oggetti" del mondo reale. [4]

1.1.1 Definizione

Esistono molteplici definizioni, a volte anche in contraddizione l'una con l'altra, di cosa sia un Knowledge Graph. In modo molto generale possiamo definire un Knowledge Graph come una struttura in cui la conoscenza è rappresentata come un insieme di concetti e dei quali vengono modellate le relazioni. Se vogliamo invece dare una definizione più formale possiamo definire un Knowledge Graph come una struttura che acquisisce e integra informazioni in una knowledge base e applica un meccanismo d'inferenza per ricavare nuova conoscenza da essa come mostrato in figura 1.1. In molte delle definizioni, la presenza di una quantità elevata di dati, ovvero di un ABox di grandi dimensioni, viene spesso considerata come aspetto caratterizzante di un Knowledge Graph, ma cosa significhi nello specifico "quantità elevata" non è meglio specificato. [4]

La knowledge base è tipicamente implementata tramite un'ontologia, ovvero una struttura a grafo dove i nodi rappresentano gli oggetti e i valori mentre le relazione tra questi e le loro proprietà sono rappresentate tramite archi. Non avendo uno schema definito a priori, questa rappresentazione tramite grafi permette una crescita più flessibile della base di dati ed è quindi adatta per rappresentare domini complessi che attingono dati da fonti molteplici e diversificate tra di loro. Un altro vantaggi di questa rappresentazione è che i linguaggi di interrogazione per strutture basate su grafi sono molto espressivi e contengono la maggior parte dei costrutti usati nei linguaggi di query più tradizionali come join, unioni, proiezioni, ... [5].

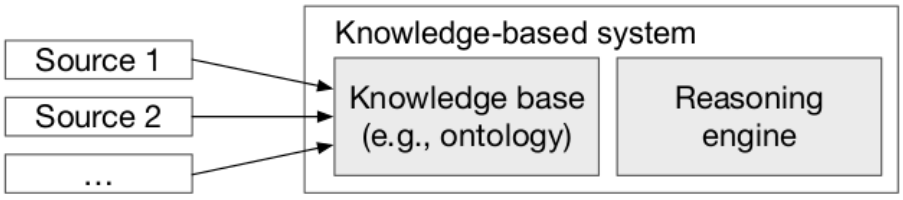

Figura 1.1. Struttura di un KG

1.2 Virtual Knowledge Graph

I Virtual Knowledge Graph (VKG) applicano al concetto di Knowledge Graph quello di data virtualization. Questo significa che in un VKG l'ontologia non viene materializzata, ma viene dichiarato un insieme di dichiarazioni di mapping che permettono di tradurre concetti e proprietà tipiche di un'ontologia in query SQL che vengono eseguite direttamente sulle sorgenti relazionali. Il meccanismo su cui si basa questa riscrittura è descritto anche dalla figura 1.2.

Un Virtual Knowledge Graph presenta il vantaggio di avere informazioni sempre aggiornate nell'ontologia dato che la query è eseguita direttamente sulle fonti sottostanti, ma, allo stesso tempo, ha performance mediamente peggiori rispetto alla sua controparte materializzata, specialmente nei casi in cui i mapping tendano ad essere particolarmente complessi [3]

Da un punto di vista più formale possiamo considerare la specifica di un Virtual Knowledge Graph come una tupla $P = (O, M, S)$ dove abbiamo:

- Ontologia O: rappresentazione a grafo del dominio in analisi con un vocabolario rappresentativo del dominio. Questo grafo è anche arricchito con informazioni riguardanti classi rappresentate nel grafo, le loro pro proprietà e la gerarchia tra esse che permette di inferire nuova conoscenza a partire dalle informazioni che già si hanno. In questo modo viene implementata una separazione tra i dettagli di basso livello delle fonti e la visione d'insieme data dall'ontologia che permette così anche a persone esperte nel campo, ma non nell'integrazione dei dati, di ricavare informazioni.

  In particolare W3C presenta vari standard per la rappresentazione delle ontologie. Tra questi i principali sono RDFS e OWL, entrambi basati sullo standard RDF (Resource Description Network) usato per descrivere grafi e al fine di interrogare questo grafo lo standard è SPARQL.

- Mapping M: insieme di affermazioni che specifica come le classi e le proprietà presenti nell'ontologia siano popolate da dati provenienti dalle sorgenti. Formalmente, dato lo schema di un database S e un'ontologia O, un'affermazione di mapping tra S e O è un'espressione in una di queste forme:

  $$\phi(x) \rightsquigarrow (f(x) \text{ rdf:type A})$$

  $$\phi(x, x') \rightsquigarrow (f(x)\ P\ f'(x'))$$

  dove f è un costruttore di termini RDF ovvero una funzione che mappa una tupla di un database a una URI o un letterale RFD. In altre parole, tutte le tuple del database vengono tradotte fornendo informazioni o sul tipo di dato o su relazioni di tipo soggetto-predicato-oggetto Lo standard per i mapping tra RDF e database relazionali fornito da W3C è R2RML.

- Schema S: struttura delle sorgenti dati, tipicamente database relazionali. Nel caso di sorgenti eterogenee tra loro è sempre possibile utilizzare uno strumento di federazione come Dremio o Denodo che espone tutte le sorgenti come se fossero parte di un unico database relazionale [3].

A questo punto possiamo definire un'istanza di un Virtual Knowledge Graph come la coppia (P, D), dove $P = (O, M, S)$ è la specifica di un VKG istanziata su un database D che rispetta lo schema S. Dati M e D le triple generate applicando M su D costituiscono il grafo RDF che esplicita il significato semantico dell'intero sistema.

Se vogliamo caratterizzare un Virtual Knowledge Graph sotto il punto di vista della logica descrittiva allora possiamo considerare l'ontologia come un TBox e le informazioni ricavate dalle sorgenti tramite i mapping come l'ABox [1] [21].

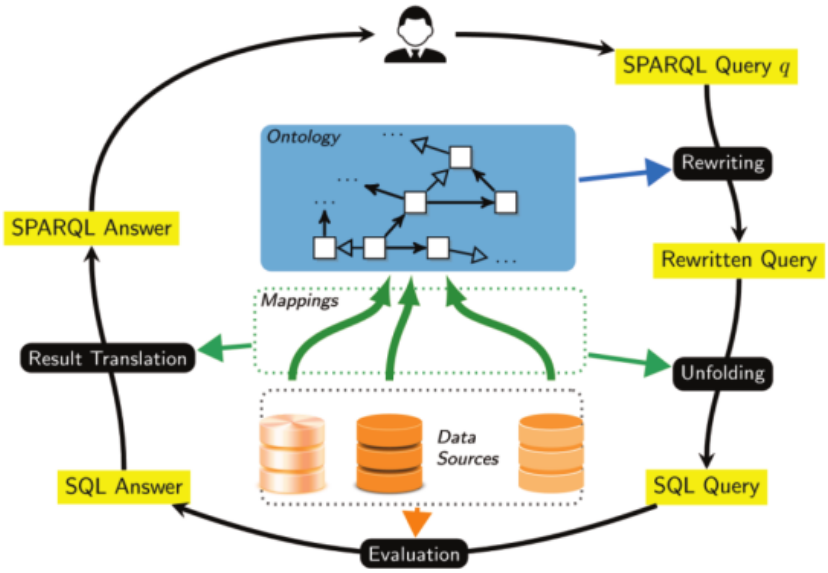

Figura 1.2. Riscrittura di una query in un VGK

1.3 Il Virtual Knowledge Graph system Ontop

Ontop è un Virtual Knowledge Graph system open-source sviluppato dalla Libera Università di Bolzano e supportato commercialmente dall'azienda Ontopic s.r.l. . Riceve inoltre contributi importanti dalla Birkbeck, Università di Londra.

1.3.1 Architettura del sistema

Possiamo considerare il sistema Ontop come strutturato su quattro livelli descritti di seguito e riassunti in figura 1.3.

Input

Ontop supporta gli standard W3C in materia di ontologie e Knowledge Graph; in particolare supporta RDF 1.1 come modello per i grafi, RDFS e OWL 2 QL per le ontologie, la maggior parte dei costrutti presenti in SPARQL 1.1 e R2RML e un sistema di mapping di Ontop, che può essere tradotto in R2RML, per i mapping.

Ontop supporta i maggiori database relazionali tra cui PostgreSQL, MySQL, H2, DB2, Oracle, . . . tramite JDBC e può anche essere utilizzato con federazioni come Dremio o Denodo. Inoltre, nonostante sia un VKG system permette di materializzare il grafo RDF se necessario [2].

Core system

Parte centrale del sistema che si occupa della traduzione, ottimizzazione ed esecuzione delle query. Alcuni dei dettagli di questo meccanismo sono descritti nella sezione successiva 1.3.2.

API

Ontop può essere utilizzato come libreria Java disponibile tramite Maven ed implementa due API:

- OWL API: implementazione di riferimento per la gestione di ontologie OWL.
- Sesame: standard de-facto per la gestione di dati in formato RDF. In particolare, Ontop implementa l'interfaccia Sesame SAIL (Storage And Inference Layer) che supporta inferenza e database relazionali.

Applicazioni

Ontop supporta anche applicazioni che permettono all'utente finale di eseguire query SPARQL in modo facilitato. Tra queste si citano in particolare: il plugin per Protege basato sull'API OWL, che fornisce uno strumento grafico per l'editing dei mapping, l'esecuzione di query SPARQL, materializzazione delle triple RDF, ... e la piattaforma Optique che utilizza Ontop come motore centrale aggiungendo un interfaccia user-friendly per la creazione e visualizzazione di query.

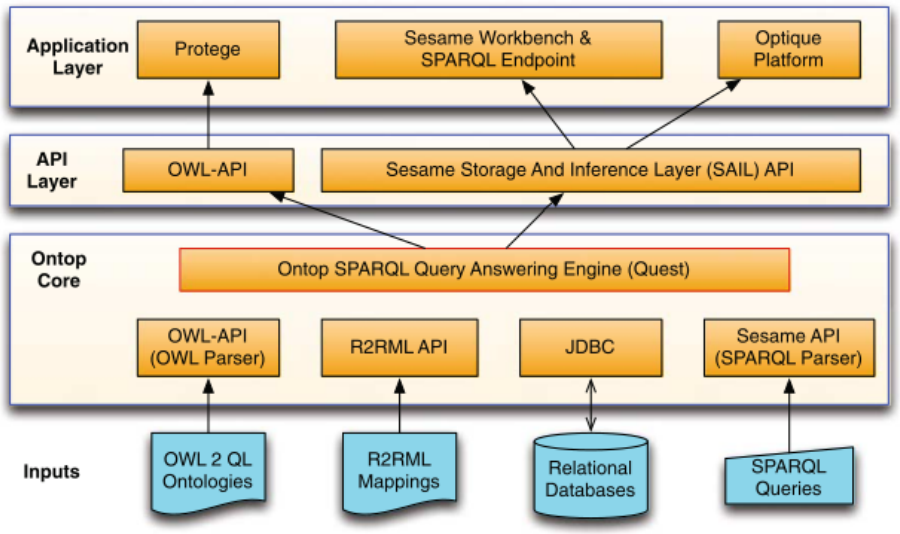

Figura 1.3. Struttura di Ontop

1.3.2 Intermediate Query language

Le prime versioni di Ontop erano inizialmente basate su Datalog come motore interno per la traduzione delle query. Questa soluzione, sufficiente per la traduzione di query semplici basate sull'unione di query congiuntive, risulta però inadatta per la traduzione di un frammento più grande di SPARQL che supporta funzionalità non monotone come OPTIONAL, modificatori di cardinalità come DISTINCT e aggregazione. Per questo con la versione v4, Ontop ha adottato uan struttura dati interna diversa chiamata Intermediate Query (IQ) che fornisce una rappresentazione uniforme sia per le query SPARQL eseguite dagli utenti che per le query SQL dei mapping. Sostanzialmente un IQ è una rappresentazione tramite un albero radicato di un'espressione algebrica dove quest'algebra è un compromesso tra l'algebra di SPARQL da una parte e l'algebra relazionale tipica dei database relazionali dall'altra. [22]

Prima di definire più nel dettaglio quest'algebra è necessario spiegare alcuni altri termini ed in particolare:

- Termine: per termine indichiamo qualsiasi variabile, costante (incluso NULL) o termine funzionale costruito a partire da variabili o costanti usando simboli funzionali di SPARQL, SQL o interni
- Sostituzione: per sostituzione intendiamo un'espressione della forma $x_1/\mu_1, \ldots, x_n/\mu_n$ dove x corrisponde ad una variabile e μ ad un termine usato o per la proiezione o per l'aggregazione.

I nodi che compongono un albero IQ possono essere di due tipi: [9]

Nodi foglia

Tra i nodi che compongono le foglie dell'albero i più interessanti sono:

- Intensional data node: nodo "prototipo" che ci si aspetta venga sostituito da un IQ; è usato principalmente per rappresentare triple e quadruple in RDF.
- Extensional data node: simile a un intensional data node con la differenza che non deve essere necessariamente essere sostituito (ad esempio se rappresenta il nome di una tabella).
- Empty node: può essere visto come uno specifico tipo di data node che rappresenta un insieme vuoto di tuple.
- Native node: è definito a partire da una query nativa (e.g. una query SQL) ed esplicita i tipi delle variabili ritornate dalla query stessa. Questo tipo di nodo viene generato alla fine della riformulazione e non è usata nei mapping.

Nodi interni

I nodi interni dell'albero risultano essere più interessanti rispetto alle foglie in quanto rappresentano quelle che sono le operazioni possibili e che devono essere tradotte tra SPARQL e SQL. Tra queste abbiamo:

- Filter node: filtra il proprio nodo figlio in base ad una condizione booleana specificata.
- Inner join node: natural join tra gli n figli del nodo sui quali può anche essere applicata una condizione booleana di join. Se questa condizione non è specificata allora il risultato è un cross-join.
- Left join node: rappresentazione del left outer join tra i due figli del nodo, eventualmente con condizione di join.
- Union node: unione degli n figli opzionalmente preceduta da un'operazione di proiezione.
- Construction node: rappresenta la sequenza di tre operazioni: la proiezione, l'applicazione di una funzione al valore di ogni tupla restituita e, infine, la rinomina delle variabili dove ognuna di queste tre operazioni è opzionale.
- Aggregation node: definito dall'insieme di variabili su cui viene eseguita l'aggregazione e da una sostituzione per definire le nuove variabili ricavate dalle funzioni di aggregazione. Proietta esclusivamente le variabili su cui è stata eseguita la proiezione e quelle risultanti dalla rinomina delle funzioni di aggregazione.
- Slice node: contiene un limite e/o un offset usati per eliminare un certo numero di tuple dal risultato.
- Distinct node: mantiene una singola occorrenza per ogni tupla.
- Order by node: ordine le tuple presenti in base ad una lista di comparatori. I valori vengono ordinati in base al primo comparatore e, solo in caso di valori uguali, viene usato il secondo comparatore e così via. L'operatore segue una semantica NULL FIRST ASC / NULL LAST DESC.

Durante la traduzione l'albero IQ subisce pesanti operazioni di ottimizzazione basate su tecniche derivate dalla Semantic Query Optimization che sono fondamentali per una riscrittura efficiente. Una loro descrizione più approfondita esula però degli obiettivi di questo elaborato. Oltre alle ottimizzazioni, il sistema è reso più performante dal delegare al DBMS specifico una buona parte della computazione della query. In particolare Ontop modella ogni dialetto SQL individualmente usando Java factory specifiche per ogni ognuno di essei ovvero modella i tipi, le convenzioni nei nomi di attributi e tabelle, la semantica delle funzioni, la struttura dei data catalog, ...

1.3.3 Esempi di utilizzo

Data la distribuzione di Ontop come progetto open-source con licenza Apache2 è impossibile conoscerne tutti i casi di utilizzo. Nonostante ciò esistono comunque casi documentati di uso di questo VGK system sia in ambito commerciale che pubblico. Di seguito ne descriviamo due di particolare interesse. [22]

Open Data Hub-Virtual-Knowledge-Graph

Open Data Hub-Virtual Knowledge Graph nasce come un progetto congiunto tra il NOI Techpark e Ontopic al fine di pubblicare e rendere disponibili i dati del'Alto Adige su turismo e mobilità. Inizialmente questi dati erano accessibili attraverso un'API JSON, ma l'introduzione di un VKG, con annesso endopoint SPARQL, ha reso questo sistema estremamente più potente e flessibile.

UNiCS

UNiCS è una piattaforma sviluppata da SIRIS Academic che integra dati provenienti da fonti come la commissione europea, governi nazionali e regionali e specifiche università al fine di aiutare università ed istituti di ricerca nel prendere scelte informate e supportate da dati. Tutti questi dati sono disponibili tramite un'ontologia che può essere interrogata tramite un endpoint SPARQL e i cui risultati possono poi essere visualizzati tramite grafiche e statistiche di più facile interpretazione [20].

2 Progetto

2.1 Ontopic

Il tirocinio da me svolto è avvenuto presso l'azienda Ontopic s.r.l. Tale azienda è nata nel 2019 come spin-off della Libera Università di Bolzano dal prof. Diego Calvanese insieme al professore Marco Montali, i ricercatori Guohui Xiao e Benjamin Cogrel e l'amministratore delegato Peter Hopfgartner presso il NOI Techpark di Bolzano [19].

L'azienda nasce con lo scopo di proporre soluzioni di data integration ed in particolare basate sull'integrazione semantica come i Virtual Knowledge Graph. Ontopic fornisce sia consulenza diretta ad aziende con soluzioni ad-hoc per integrazione ed analisi dati che una soluzione proprietaria per il mapping di VKG chiamata Ontopic Studio. Questo applicativo è basato sul Knowledge Graph system Ontop, di cui Ontopic è uno dei principali contributori, e permette di realizzare mapping in modo efficiente tramite un interfaccia grafica no-code semplice che ne permette quindi l'uso anche a persone senza un background in tecniche semantiche. [10]

2.2 Il modulo bi-connector

Durante la mia esperienza di tirocinio presso Ontopic, ho lavorato all'interno del progetto del bi-connector. Tale modulo ha lo scopo di permettere all'utente finale di ricavare informazioni su un VKG non solo tramite query SPARQL, come di consuetudine, ma anche tramite strumenti di Business Intelligence (BI) come Tableau, PowerBi o Qlik. Mentre questi applicativi presentano nativamente connettori verso molteplici fonti di dati tra cui: la maggior parte dei database relazionali, database NoSQL come MongoDB, fogli Excel, dati in formato JSON, file PDF, connessioni personalizzate tramite JDBC o ODBC, ..., non forniscono invece connettori nativi per i VKG ed è proprio per questo che nasce bi-connector.

Possiamo pensare a bi-connector come costituito di due parti principali: una che si occupa della costruzione di un database a partire dall'ontologia che possa essere connesso al strumento di BI target e una seconda parte che ha il compito di tradurre le query SQL sul database in una forma comprensibile dal VKG sottostante.

2.2.1 Creazione database

Dalle sorgenti di dati e dai file che utilizzati per creare i mapping di un VKG viene creato un database PostgreSQL. Questo è reso possibile modificando direttamente i cataloghi di sistema resi disponibili da PostgreSQL tramite l'insieme di tabelle *pg_catalog*. La comunicazione da e verso questo database è resa possibile tramite un'implementazione basata sul protocollo Wire che è il protocollo usato da PostgreSQL per lo scambio di messaggi.

Da estendere chiedendo a Benjamin

2.2.2 Parsing di query SQL

È possibile interrogare il database inizialmente creato tramite query SQL, ma queste devono poi essere tradotte in una rappresentazione comprensibile dal VKG sottostante. Questo perché le query non sono eseguite sul database da noi creato, ma sul Virtual Knowledge Graph originale così da poterne sfruttare le capacità di inferenza.

In quanto il VKG system utilizzato è Ontop, le query SQL vengono tradotte in un albero IQ, descritto precedentemente anche nella sezione 1.3.2.

Inizialmente, le query delle quali era possibile fare il parsing erano quasi esclusivamente del tipo SELECT-JOIN-WHERE, ovvero un'unione di query congiuntive. Questo era dovuto al fatto che il parser SQL in uso era quello nativo di Ontop, il quale ha come scopo principale la traduzione dei mapping che sono, nella maggior parte dei casi, rappresentabili tramite questo tipo di semplici query. A causa di questa limitazione moltissime delle query generate automaticamente da strumenti di Business Intelligence dovevano essere riscritte in una forma semplifica per poter essere eseguite. Di seguito un esempio di ciò in cui la query è semplificata eliminando il LEFT JOIN e l'ORDER BY in quanto non supportati.

```
    SELECT n.oid, n.*, d.description
    FROM pg_catalog.pg_namespace n
    LEFT OUTER JOIN pg_catalog.pg_description d ON d.objoid=n.oid
        AND d.objsubid=0 AND d.classoid='pg_namespace'::regclass
    ORDER BY nspname
diventa:
    SELECT n.*
    FROM pg_catalog.pg_namespace n;
```

Date queste limitazioni del parser originale, si è deciso di adottarne uno diverso ovvero JSqlParser. JSqlParser è un parser di istruzioni SQL che trasforma una query in una gerarchia di classi Java. È basato sul visitor pattern il quale permette di separare in modo semplice un algoritmo dalla struttura dati sulla quale è utilizzato. In particolare tale pattern rappresenta un'operazione che deve essere eseguita su un insieme eterogeneo di oggetti e, per ognuno di questi, potrebbe dover essere implementata in modo differente [7].

2.3 Prime attività

La prima attività svolta all'interno del tirocinio è stata quella di familiarizzare con le tecnologie legate ai VKG che mi sarebbero servite in seguito. In particolare ho svolto il tutorial introduttivo ad Ontop tramite la piattaforma Protege così da provare direttamente cosa si intende per tradurre fonti di dati in un'ontologia tramite mapping. (Forse non mettere)

Il mio compito principale è stato quello di estendere il parser SQL in uso così da supportare un numero maggiore di costrutti. Come affermato anche precedentemente, il parser inizialmente utilizzato era esclusivamente in grado di tradurre semplici query il che è risultato essere insufficiente affinché potesse essere usato da strumenti di Business Intelligence in modo non banale.

È stato quindi analizzato quali fossero i costrutti maggiormente presenti nelle query generate automaticamente dagli strumenti di BI, di cui un esempio sopra nella sezione 2.2.2. Tra questi i principali erano sicuramente il GROUP BY per l'aggregazione, dato anche lo scopo degli strumenti di BI di fornire informazioni riassuntive d'insieme sull'intero dataset, e l'ORDER BY che permette di visualizzazione i dati in base a parametri arbitrari.

2.3.1 Database per il testing

Durante la durata del tirocinio, tutte le funzionalità implementate sono state testate su un semplice VKG con un database relazionale H2 come sorgente che rappresenta alcune informazioni riguardanti professori universitari. I test sono stati eseguiti sia all'interno della codebase stessa grazie al framework JUnit che tramite il client SQL DBeaver che ha permesso la visualizzazione del database PostgreSQL creato dal bi-connector e la sua interrogazione diretta. Di seguito si descrivono le tabelle (2.1 2.2, 2.3, 2.4) di questo database che sono inoltre quelle a cui si farà riferimento in tutti i frammenti di codice successivi.

Inoltre, dato il fatto che ogni DBMS tende ad avere un proprio dialetto SQL, è stato molto importante assicurarsi che i costrutti da me implementati ricalcassero il comportamento presente

Tabella 2.1. Tabella prof

id	fName	lName
http://university.example.org/professor/10	Roger	Smith
http://university.example.org/professor/20	Frank	Pitt
http://university.example.org/professor/30	John	Deep
http://university.example.org/professor/40	Micheal	Jackson
http://university.example.org/professor/50	Diego	Gamper
http://university.example.org/professor/60	Johann	Helmer
http://university.example.org/professor/70	Barbare	Dodero
http://university.example.org/professor/80	Mary	Poppins

Tabella 2.2. Tabella prof_stats

profId	totalStudents	countCourse
http://university.example.org/professor/10	21	2
http://university.example.org/professor/30	12	1
http://university.example.org/professor/80	13	1

Tabella 2.3. Tabella course

id	duration	nbStudents
http://university.example.org/course/LinearAlgebra	24.5	10
http://university.example.org/course/DiscreteMathematics	30	11
http://university.example.org/course/AdvancedDatabases	20	12
http://university.example.org/course/ScientificWriting	18	13

Tabella 2.4. Tabella teaching

teacher	course
http://university.example.org/professor/30	http://university.example.org/course/AdvancedDatabases
http://university.example.org/professor/10	http://university.example.org/course/DiscreteMathematics
http://university.example.org/professor/10	http://university.example.org/course/LinearAlgebra
http://university.example.org/professor/80	http://university.example.org/course/ScientificWriting

in PostgreSQL e, nel caso ciò non fosse possibile, fornire messaggi d'errore di facile comprensione. Ad esempio, la funzione *concat()* è null-rejecting, ovvero tutti gli argomenti pari a NULL sono ignorati, mentre in altri dialetti, come ad esempio MySQL, se uno degli argomenti è NULL allora il risultato della concatenazione è anch'esso NULL. Al fine di ottenere ciò, ho utilizzato la documentazione di PostgreSQL e, per test più pratici, ho usato DBFiddle, uno strumento online per testare in modo veloce, e senza bisogno di alcun setup, frammenti di codice SQL.

2.4 Costrutti implementati

La struttura usata per il parsing è basata sulla classe *IQTreeExpression.java* la quale è composta da:

- IQTree: albero composto di nodi IQ che viene costruito partendo dalle foglie e viene esteso aggiungendo nuovi nodi come radici dell'albero stesso. Da un punto di vista più pratico abbiamo come foglie degli extensional node che rappresentano le tabelle elencate all'interno della query e al di sopra di esse sono presenti altri nodi che rappresentano le varie operazioni eseguite.

- RAExpressionAttributes: dizionario che contiene le colonne delle tabelle coinvolte nella query e tutti gli alias ad esse associati. Per alias si intendono i nomi definiti dall'utente tramite la keyword AS o come nome di una tabella e quelli definiti internamente al codice al fine di assicurarne l'univocità.

Di seguito un esempio di un IQTreeExpression generato dalla query

```
SELECT "id", "fName", "lName"
FROM prof p

IQTreeExpression{
    iqTree = EXTENSIONAL "prof_views"."prof"(0:id1,1:fName1,2:lName1),
    raExpressionAttributes=attributes: {"id"=id1, p."id"=id1,
        "fName"=fName1, p."fName"=fName1, "lName"=lName1, p."lName"=lName1}
        with {"id"=[p], "fName"=[p], "lName"=[p]}
}
```

2.4.1 Modificatori di cardinalità

I primi costrutti da essere implementati sono stati quelli responsabili della modifica della cardinalità di una tabella ovvero DISTINCT e LIMIT-OFFSET. La loro implementazione risulta essere abbastanza semplice da un punto di vista logico, ma ciò mi ha permesso di acquisire familiarità con la codebase preesistente e con librerie esterne come JSqlParser in modo più rapido dato che l'implementazione dei costrutti non aveva logiche complesse.

DISTINCT

Il costrutto DISTINCT permette di eliminare righe duplicate ed è stato implementato aggiungendo un nodo IQ di tipo distinct.

La clausola DISTINCT ON non è stata invece implementata in quanto non parte dello standard SQL, ma fornita da PostgreSQL. Inoltre non è stata considerata fondamentale in qunato può essere rimpiazzata tramite l'uso di una subquery o in alcuni casi anche tramite aggregazione [14].

LIMIT e OFFSET

Le clausole LIMIT e OFFSET permettono di restringere il numero di righe restituite da una query e di saltare le prime n righe quando queste vengono restituire rispettivamente. Data le loro funzionalità complementari sono spesso usate insieme anche se è possibile usare singolarmente Ad esempio la forma `LIMIT 5 OFFSET 2` ritorna le prime 5 righe dopo che sono state saltate le prime 2.

Anche se supportata dalla maggior parte dei DBMS, e quindi ampiamente usata, la clausola LIMIT non fa parte dello standard SQL. Una possibile alternativa standardizzata è l'utilizzo della keyword FETCH la cui sintassi è `FETCH { FIRST | NEXT } [ fetch_rows ] { ROW | ROWS } ONLY` dove FIRST/NEXT e ROW/ROWS sono sinonimi e possono quindi essere usati in modo intercambiabile [18].

Al fine di implementarne la funzionalità, indipendentemente se per LIMIT o per FETCH, è stato usato uno slice node il quale permette di specificare esclusivamente un offset o sia un limite che un offset.

Inoltre, quando si utilizza LIMIT è consigliato aggiungere anche un ORDER BY che forza un ordine alle righe altrimenti il risultato non è garantito che il risultato sia lo stesso per esecuzioni successive della stessa query [13].

2.4.2 Ordinamento righe

La keyword ORDER BY permette riordinare le righe risultanti da una query in base a specifiche condizioni, sia in modo ascendente che discendente. Un altro aspetto importante è quello del NULL ordering, ovvero come i valori NULL sono trattati al fine dell'ordinamento. In PostgreSQL i valori pari a NULL vengono considerati come maggiori di qualsiasi altro valore mentre questo è l'opposto per gli IQ tree i quali seguono una semantica simile a quella di SPARQL. Quindi gli ordinamenti del tipo `ORDER BY ASC NULLS LAST` e `ORDER BY DESC NULLS FIRST` non sono supportati [15].

L'implementazione dell'ORDER BY è risultata essere relativamente complessa in quanto può essere usata su un'ampia casistica. Ad esempio si può avere uno o più criteri di ordinamento e ognuno di essi può essere una colonna o una funzione. Inizialmente il costrutto è stato implementato usando un order by node il quale richiede come argomento una lista di comparatori dove ogni comparatore è costituito dal termine rispetto al quale si sta effettuando l'ordinamento e se quest'ultimo è ascendente o meno.

Una volta svolti i primi test, è subito emerso un problema riguardante la rinomina delle variabili in quanto una volta che ad una variabile veniva assegnato un alias si andava a perdere quale fosse il nome della variabile originale. Quindi ad esempio un query del tipo:

```
SELECT "fName" as v
FROM prof p
ORDER BY "fName"
```

avrebbe restituito un errore in quanto l'unica variabile presente a seguito della proiezione sarebbe stata v e sarebbe quindi stato impossibile eseguire un ordinamento basato su *fName*.

Si è quindi deciso di estendere l'implementazione per supportare questo comportamento aggiungendo un construction node che proietta tutte le variabili e tiene traccia delle sostituzioni che avvengono. Inoltre al fine di evitare collisioni tra gli attributi non proiettati, che vengono appunto mantenuti in questa versione, e gli alias, tutti gli attributi sono rinominati aggiungendo un suffisso casuale al nome della variabile stessa. La query vista sopra produce questa IQTreeExpression:

```
IQTreeExpression{
    iqTree=CONSTRUCT [v] []
     ORDER BY [ASC(fName)]
```

```
        CONSTRUCT [v, fName, fName1f1611f1-6d9d-4347-9c60-f51bbdc89a09,
          lName, lName0c931a3d-93fb-406c-b33c-6f4743af211c, id,
          idf859a5d9-d246-42a2-a62f-e722a1175dc4] [fName/v,
          idf859a5d9-d246-42a2-a62f-e722a1175dc4/id,
          lName0c931a3d-93fb-406c-b33c-6f4743af211c/lName,
          fName1f1611f1-6d9d-4347-9c60-f51bbdc89a09/v]
         CONSTRUCT [id, v, lName] []
          EXTENSIONAL "prof_views"."prof"(0:id,1:v,2:lName),
      raExpressionAttributes=attributes: {v=v} with {v=[]}
}
```

È inoltre possibile una forma del tipo `ORDER BY numerical_constant` dove la costante numerica presente viene utilizzata come indice della corrispondente colonna proiettata. Tale uso è scoraggiato in quanto il risultato potrebbe non essere deterministico se si va a cambiare la struttura del database stesso, ma è stato implementato ugualmente in quanto viene ampiamente usato nelle query generate in modo automatico da Tableau.

2.4.3 Combinazione tabelle

Un'altra parte importante al fine formulare query con risultati non banali, è la possibilità di combinare le informazioni presenti in molteplici tabelle. Personalmente, mi sono occupata esclusivamente dell'implementazione del LEFT JOIN in quanto CROSS e INNER JOIN erano già entrambi funzionanti nel momento in cui ho iniziato il mio tirocinio.

In particolare, l'operazione di LEFT JOIN estende tutte le righe della prima tabella con i valori di una seconda tabella basandosi su una determinata condizione booleana di join. Tale condizione può essere espressa in due modi:

- `LEFT JOIN ON condition`: in questo caso viene specificata una serie di condizioni, tipicamente di uguaglianza tra due colonne appartenenti a tabelle diverse
- `LEFT JOIN USING column`: usato se la colonna rispetto alla quale vogliamo eseguire il join ha lo stesso nome in entrambe le tabelle

A fini implementativi, entrambe le condizioni vengono espresse come un ImmutableExpression, ovvero un espressione booleana, la quale viene usata come filtro all'interno di un left join node. Di seguito un esempio dell'IQTreeExpression e del risultato generato dalla query:

```
SELECT p.id, "fName", "totalStudents"
FROM prof p
LEFT JOIN prof_stats ps ON p.id = ps."profId"
```

la quale produce come IQTreeExpression

```
IQTreeExpression{iqTree=CONSTRUCT [id, fName, totalStudents] []
 CONSTRUCT [id, fName, totalStudents,lName, countCourse, profId,
    variabili generate casualmente per univocità] [elenco
    sostituzioni come visto precedentemente]
  CONSTRUCT [id, fName, lName, profId, totalStudents, countCourse] []
   LJ STRICT_EQ2(id,profId)
    EXTENSIONAL "prof_views"."prof"(0:id,1:fName,2:lName)
    EXTENSIONAL "prof_views"."prof_stats"(0:profId,1:totalStudents,
        2:countCourse),
raExpressionAttributes=attributes: {id=id, "fName"=fName,
    "totalStudents"=totalStudents} with {id=[], "fName"=[],
    "totalStudents"=[]}}
```

Tabella 2.5. Tabella LEFT JOIN

id	fName	totalStudents
http://university.example.org/professor/10	Roger	21
http://university.example.org/professor/20	Frank	NULL
http://university.example.org/professor/30	John	12
http://university.example.org/professor/40	Micheal	NULL
http://university.example.org/professor/50	Diego	NULL
http://university.example.org/professor/60	Johann	NULL
http://university.example.org/professor/70	Barbare	NULL
http://university.example.org/professor/80	Mary	13

ed ha come risultato la tabella 2.5.

Questa implementazione risulta non essere però completa in quanto presenta comportamenti non corretti in alcuni casi nei quali le colonne sulle quali viene eseguito il join hanno lo stesso nome. Ad esempio, la query

```
SELECT p.*, ps.*
FROM (SELECT id as "profId", "fName", "lName" FROM prof_views.prof p1) p
LEFT JOIN prof_views.prof_stats ps ON p."profId" = ps."profId"
```

ritorna un errore in quanto i nomi di due colonne all'interno del SELECT sono uguali. Questa è una limitazione di Ontop stesso il quale non permette di avere variabili con lo stesso nome all'interno di una proiezione.

Una volta realizzato il LEFT JOIN, l'implementazione del RIGHT JOIN è risultata essere banale in quanto è sufficiente invertire i ruoli delle due tabelle sulle quali viene effettuato il join per ottenere il comportamento corretto.

2.4.4 Operazioni insiemistiche

Gli operatori insiemistici uniscono i risultati di due query in un unico insieme finale. I due insiemi che devono essere uniti devono rispettare alcune condizioni: devono restituire lo stesso numero di colonne e colonne corrispondenti devono avere tipi compatibili [11]. Inoltre, una limitazione aggiuntiva introdotta dalla nostra implementazione prevede che colonne corrispondenti debbano avere lo stesso nome.

UNION

L'operazione di UNION aggiunge alle righe risultanti dalla prima query quelle ottenute da una seconda query eliminando i duplicati. Se si vuole mantenere le righe duplicate è possibile usare il costrutto UNION ALL.

La sua implementazione prevede l'utilizzo di uno union node, il quale mantiene i duplicati, seguito eventualmente da un distinct node nel caso sia necessario eliminare le copie.

EXCEPT

L'operazione di sottrazione, chiamata in PostgreSQL EXCEPT invece dello standard MINUS, restituisce tutte le righe risultanti dalla prima query che non sono presenti anche nei risultati della seconda.

La sua implementazione risulta essere particolarmente interessante in quanto non esiste un IQ node che ne imiti di comportamento. Viene quindi eseguito un left join tra le due tabelle e tramite un filtro vengono eliminate tutte le righe per le quali le colonne provenienti dalla tabella di destra, identificate dalla colonna ausiliaria *rightProv*, non sono pari a NULL. Infine viene usato un construct node così da mantenere solo le variabili desiderate. Ad esempio la query

```
SELECT "id"
FROM prof p
EXCEPT
SELECT "profId" AS "id"
FROM prof_stats ps
```

ha come risultato la tabella 2.6

Tabella 2.6. Tabella MINUS

id
http://university.example.org/professor/20
http://university.example.org/professor/40
http://university.example.org/professor/50
http://university.example.org/professor/60
http://university.example.org/professor/70

e produce come l'IQTreeExpression:

```
IQTreeExpression{iqTree=CONSTRUCT [id] []
 FILTER IS_NULL(rightProv)
   LJ
    CONSTRUCT [id] []
     CONSTRUCT [id, fName, lName, variabili generate casualmente per
      univocità] [elenco sostituzioni]
        CONSTRUCT [id, fName, lName] []
         EXTENSIONAL "prof_views"."prof"(0:id,1:fName,2:lName)
    CONSTRUCT [id, rightProv] [rightProv/"TRUE"^^BOOLEAN]
     CONSTRUCT [id] []
      CONSTRUCT [id, totalStudents,countCourse, variabili generate
       casualmente per univocità] [elenco sostituzioni]
        CONSTRUCT [id, totalStudents, countCourse] []
         EXTENSIONAL "prof_views"."prof_stats"(0:id,1:totalStudents,
         2:countCourse),
raExpressionAttributes=attributes: {"id"=id} with {"id"=[]}}
```

Inoltre non è stato possibile implementare l'operazione di EXCEPT ALL in quanto non supportata dalla versione in uso di JSqlParser.

2.4.5 Aggregazione

L'aggregazione è una funzionalità estremamente importante in quanto permette di raggruppare i dati in base a parametri arbitrari ed ottenere da questi informazioni riassuntive sui dati.

Funzioni di aggregazione

Le funzioni di aggregazione (SUM, COUNT, MIN, MAX, AVG) permettono di esprimere caratteristiche di una colonna, o di un'intera tabella, tramite un singolo numero riassuntivo.

Il principale ostacolo alla loro implementazione è stato il determinare il tipo della colonna sulla quale la funzione era applicata. Per questo, inizialmente, si è assunto a priori che tutte le funzioni di aggregazione fossero eseguite su colonne di tipo intero. In un secondo momento si è implementato un meccanismo grazie alla classe *NotYetTypedAggregationFunctionSymbol.java* la quale permette di posporre la decisione sul tipo di dato fino a quando questo non è reso disponibile.

Ad esempio, le due query seguenti applicano la funzione SUM alle colonne nbStudents e duration le quali sono rispettivamente un numero intero e uno decimale ed è possibile vedere ciò nel tipo che viene dato alla funzione all'interno dell'aggregation node.

```
SELECT sum("nbStudents") AS sum
FROM course c
```

produce l'IQTreeExpression:

```
IQTreeExpression{iqTree=CONSTRUCT [sum] []
 CONSTRUCT [sum] []
  AGGREGATE [] [sum/SUM_BIGINT(nbStudents1)]
   CONSTRUCT [id1, duration1, nbStudents1] []
    EXTENSIONAL "prof_views"."course"(0:id1,1:duration1,2:nbStudents1),
raExpressionAttributes=attributes: {sum=sum} with {sum=[]}}
```

mentre la query

```
SELECT sum("duration") AS sum
FROM course c
```

produce l'IQTreeExpression

```
IQTreeExpression{iqTree=CONSTRUCT [sum] []
 CONSTRUCT [sum] []
  AGGREGATE [] [sum/SUM_DECIMAL(duration1)]
   CONSTRUCT [id1, duration1, nbStudents1] []
    EXTENSIONAL "prof_views"."course"(0:id1,1:duration1,2:nbStudents1),
raExpressionAttributes=attributes: {sum=sum} with {sum=[]}}
```

GROUP BY

La clausola GROUP BY permette di raggruppare insieme righe di una tabella che hanno lo stesso valore per tutte le colonne per le quali stiamo raggruppando. Una volta avvenuto il raggruppamento determinati gruppi possono essere eliminati in base ad una condizione booleana specificata all'interno del costrutto HAVING, in modo simile a quanto accade per la clausola WHERE [12].

Il costrutto viene implementato tramite un aggregation node al di sopra del quale viene aggiunto un filter node nel caso in cui sia presente la clausola HAVING. La parte più problematica risulta essere la gestione della proiezione degli variabili in quanto solo le variabili rispetto a cui si raggruppato possono essere proiettate, ma allo stesso tempo è necessario mantenere anche tutte le variabili rispetto a quali non si è raggruppato in quanto queste possono essere usate all'interno di funzioni di aggregazione nel SELECT e nel HAVING.

Inoltre il GROUP BY permette di raggruppare non solo in base a colonne nella tabella, ma anche con funzioni. Al fine di rendere l'implementazione omogenea con il caso base descritto sopra, prima di gestire il GROUP BY viene aggiunto un construction node dove ogni funzione distinta presente viene sostituita con una variabile; in questo modo il GROUP BY "vede" esclusivamente variabili. Di seguito una query di esempio:

```
SELECT teacher, MIN(course) AS "minCourse"
FROM teaching t
GROUP BY teacher, LENGTH("teacher")
HAVING LENGTH(teacher) > 20
```

la quale ha come risultato la tabella 2.7 e produce l'IQTreeExpression

Tabella 2.7. Tabella GROUP BY

teacher	course
http://university.example.org/professor/10	http://university.example.org/course/DiscreteMathematics
http://university.example.org/professor/30	http://university.example.org/course/AdvancedDatabases
http://university.example.org/professor/80	http://university.example.org/course/ScientificWriting

```
IQTreeExpression{iqTree=CONSTRUCT [teacher, minCourse] []
 CONSTRUCT [teacher, minCourse, teacher1c52a1c4-11f6-43c1-8311-9e908578b9]
 [teacher1c52a1c4-11f6-43c1-8311-9e908578b9/teacher]
  FILTER GT(CHAR_LENGTH1(teacher),"20"^^BIGINT)
   AGGREGATE [fun6079a412-7f91-4004-9bcc-f3e2f545d3ce, teacher]
   [minCourse/MIN_TEXT(course1)]
    CONSTRUCT [teacher, course1, fun6079a412-7f91-4004-9bcc-f3e2f545d3ce]
    [fun6079a412-7f91-4004-9bcc-f3e2f545d3ce/CHAR_LENGTH1(teacher)]
     EXTENSIONAL "prof_views"."teaching"(0:teacher,1:course1),
raExpressionAttributes=attributes:{teacher=teacher, "minCourse"=minCourse}
with {teacher=[], "minCourse"=[]}}
```

Come detto prima è appunto possibile vedere il construction node al di sotto dell'aggregation node che esegue la sostituzione rispetto alla funzione LENGTH e il filtro usato per l'implementazione del HAVING al di sopra dell'aggregation node.

Così come per l'ORDER BY, anche per il GROUP BY è possibile indicare le colonne rispetto a cui si vuole raggruppare utilizzando un indice numerico.

2.5 Risultati ottenuti

Con l'introduzione dell'aggregazione (e anche dell'order by) è stato possibile rimuovere buona parte delle query automaticamente create da Tableau –¿ accesso a Tableau e prime dashboard create su dataset non banali (chiedere a Benjamin se ha qualche screenshot)

3 Conclusioni

Importanza tirocinio dal punto di vista formativo e personale (prima esperienza di lavoro con un gruppo di programmatori, esperienza su una codebase di una certa dimensione, importanza testing in ambito aziendale, interesse personale nel mondo del Data Integration, ...)

3.1 Possibili sviluppi futuri

Allargare l'insieme delle query supportate (funzioni su date, cursori, meccanismi di autenticazione, ...)

Supportare altri strumenti di BI (PowerBI, Qlik, ...)

Bibliografia

[1] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, and Guohui Xiao. Ontology-based data access – beyond relational sources. *Intelligenza Artificiale*, 13:21–36, 2019. 1.

[2] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[3] Diego Calvanese, Davide Lanti, Tarcisio Mendes De Farias, Alessandro Mosca, and Guohui Xiao. Accessing scientific data through knowledge graphs with ontop. *Patterns*, 2(10):100346, 2021.

[4] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. *SEMANTiCS (Posters, Demos, SuCCESS)*, 48(1-4):2, 2016.

[5] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. Knowledge graphs. *Synthesis Lectures on Data, Semantics, and Knowledge*, 12(2):1–257, 2021.

[6] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and S Yu Philip. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 33(2):494–514, 2021.

[7] JSqlParser. Jsqlparser sourceforge. `http://jsqlparser.sourceforge.net/`.

[8] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, 2002.

[9] Ontop. Intermediate query (iq). `https://ontop-vkg.org/dev/internals/iq.html`.

[10] Ontopic. Ontopic studio. `https://ontopic.ai/en/ontopic-studio/`.

[11] PostgreSQL. Combining queries (union, intersect, except). `https://www.postgresql.org/docs/14/queries-union.html`.

[12] PostgreSQL. The group by and having clauses. `https://www.postgresql.org/docs/current/queries-table-expressions.html#QUERIES-GROUP`.

[13] PostgreSQL. Limit and offset. `https://www.postgresql.org/docs/14/queries-limit.html`.

[14] PostgreSQL. Select lists. `https://www.postgresql.org/docs/14/queries-select-lists.html`.

[15] PostgreSQL. Sorting rows (order by). `https://www.postgresql.org/docs/14/queries-order.html`.

[16] SAP. What is data integration? https://www.sap.com/insights/what-is-data-integration.html.

[17] Bhavana Sayiram. Importance of data integration in the data decade. https://education.dellemc.com/content/dam/dell-emc/documents/en-us/2021KS_Sayiram-Data_Integration.pdf.

[18] sqltutoria.org. Sql fetch. https://www.sqltutorial.org/sql-fetch/.

[19] Unibz. Ontopic. nasce il primo spin-off di unibz. https://www.unibz.it/it/news/132449-ontopic-nasce-il-primo-spin-off-di-unibz.

[20] UNiCS. Unics website. https://unics.cloud/.

[21] Guohui Xiao, Linfang Ding, Benjamin Cogrel, and Diego Calvanese. Virtual knowledge graphs: An overview of systems and use cases. *Data Intelligence*, 1(3):201–223, 2019.

[22] Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalaycı, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. The virtual knowledge graph system ontop. In *International Semantic Web Conference*, pages 259–277. Springer, 2020.