SQLogy: Integrated Query Engine for Relational and Ontology Databases

Submitted by: Xiaoshuai Li
Department of Data Science
Worcester Polytechnic Institute
May 2021

A Thesis
Submitted to
the faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Data Science

APPROVED:

Prof. Mohamed Y. Eltabakh, Thesis Supervisor

Prof. Nima Kordzadeh, Thesis Reader

Prof. Elke A. Rundensteiner, Department Head

Acknowledgements

First and foremost, I would like to show my gratitude to my advisor Prof. Mohamed Eltabakh, who has been on the whole journey with me ever since the day one of this research and thesis work. Prof. Eltabakh is a respectful, responsible, and resourceful professor and scholar. I appreciate all the thoughtful and genuine assistance and guidance he has ever offered, which helps me overcome different kinds of challenges and difficulties encountered in this journey. He is one of the kindest people I have ever known in my life. I have grown so much studying and working with him.

Then I also want to express my gratitude to my thesis reader, Prof. Nima Kordzadeh, for his inspiring encouragement as well as patient assistance in this thesis research. His valuable revisions and suggestions help me so much to improve the quality of this thesis continuously.

I also would like to thank Prof. Chun-Kit Ngan for offering me a great opportunity to know the study field of Knowledge Graph and Ontologies. He is very kind and supportive in my study and research at WPI.

My sincere thanks also go to my dear friend Akim Ndlovu who has been there encouraging me to accomplish this thesis, as well as my parents who are always there during my research and my entire life.

Last but not least, I would thank WPI for everything she has provided me in these ten years. She has totally changed my life and made who I am now. I am so grateful we chose each other in this big world.

Abstract

Nowadays, applications collect and generate huge amounts of data, but it is often isolated and siloed away in different formats under various data systems. It is such a challenging task to query both structured tabular data stored in relational databases and associated semantic data stored in ontology dataset. With the current techniques, in order to query data from both relational databases and ontologies, users must transfer one data source to the other first to unify the data resources and facilitate interoperability. This approach has some drawbacks. In this thesis, we propose a query engine "SOLogy", and its associated web interface, that facilitates querying from RDBMS and ontology dataset natively without data movement. The engine then manages the integrated processing transparently from the end-user.

Keywords: data lake, ontology, knowledge graph, semantic data, relational databases, Jena, Apache Jena Fuseki, Flask, SQLAlchemy, SQL

Table of Contents

Acknowledgements 1
Abstract 2
Chapter 1 - Introduction 5
1.1 Ontology 5
1.2 SPARQL 6
1.3 Limitation of Previous Work 7
1.3.1 Drawbacks of Ontology-to-Database Transformation 8
1.3.2 Drawbacks of Database-to-Ontology Transformation 9
Chapter 2 - Proposed Solution and Architecture Overview 11
2.1 Architecture Overview 11
2.1.1 The SQLogy User Interface Layer 12
2.1.2 SQLogy Query Parsing Layer 14
2.1.3 The SQLogy Back-end 15
2.2 Advantages of SQLogy against Previous Work 16
2.2.1 Heterogeneity of Data Sources (RDBMS and Ontologies) 16
2.2.2 Popularity of SQL Queries 16
2.2.3 Problematic Data Transformation 17
2.3 User Interface (function explained in detail) 17
2.3.1 SQL Query Input 17
2.3.2 SQL Query Output 18
Chapter 3 - Use Cases 21
3.1 Data Source 21
3.2 Six Use Cases (Five Major Use Cases & One Special Use Case) 22
3.2.1 Use Case 1: Only Query PostgreSQL Database with the Original SQL Query 23
3.2.2 Use Case 2: Only Query Ontology Data in Apache Fuseki Server 25
3.2.3 Use Case 3: Query Database in Postgres DB → Query Ontology on Fuseki Server ... 27
3.2.4 Use Case 4: Query Ontology in Fuseki Server → Query Database in Postgres DB 30
3.2.6 Use Case 6: User Updates the Ontology Dataset (Supporting Ontology Update) 36
Chapter 4 - Experiments & Evaluation 40

4.1 Experiment Setup 40
4.1.1 Data Source 40
4.1.2 Preliminary Jobs 41
4.2 Performance Results & Evaluation 41
4.2.1 Baseline Approach 41
4.2.2 Measurement 41
4.2.3 Sample Query Execution Results (SQLogy & Baseline) 42
4.2.4 Query Execution Time Results 51
Chapter 5 - Related Work 53
Chapter 6 - Conclusion and Future Work 55
6.1 Conclusion 55
6.2 Future Work 55
6.2.1 Improve Compatibility of SQLogy on Input SQL Query 55
6.2.2 Extend Adaptability of SQLogy on Data Sources & Systems 56
6.2.3 Optimize Internal Query Execution Efficiency & Effectiveness 56
6.2.4 Integrate SQLogy into Other Applications 56
Chapter 7 - Reference 57

Chapter 1 - Introduction

Today, almost all applications generate a gigantic amount of data. Each day, millions of structured and unstructured data objects are being generated, processed, but usually isolated and soiled away in different storage systems such as relational databases, data warehouses, and knowledge graphs. Traditional structured data (tabular data), either machine- or human-generated, typically resides in relational database management systems (RDBMS) in a very structured fashion, which makes it quickly accessible and easily searchable by human or algorithm generated queries. Structured Query Language (SQL) enables querying over structured data inside relational databases.

1.1 Ontology

Ontology is originally a field of metaphysics that studies and concerns the topic of the essence as well as the relations of existence or being (Simons, 2015). Ontology is also defined by Oxford Languages as "a set of concepts and categories in a subject area or domain that shows their properties and the relations between them". In the field of Artificial Intelligence (AI), the term ontology is regarded as "a representation vocabulary, often specialized to some domain or subject matter" (Gasevic & Djuric, 2006). Ontologies are widely used in several fields of AI and computing including knowledge representation and engineering, database design, information retrieval and so on (Gasevic & Djuric, 2006). In this thesis, , we focus on the domain of *Semantic Web*, which is applicable to a broad range of applications. Semantic Web can be referred to W3C's vision of the Web of linked data and it is an essential part of the W3C standards.. The technology of Semantic Web makes it accessible for people to create data stores on the Web, build vocabularies, and write rules for handling data (W3C.org, 2015).

Ontologies share an abundance of great features which make them available to be used by knowledge representation and engineering, intelligent systems as well as other fields of study in AI and computing. First, an ontology offers a vocabulary for referring to the concepts in a specific subject area. When compared to human-oriented vocabularies, an ontology is capable of providing logical statements that describe the definition or identity of the terms as well as the relationships among terms. Those logical statements of an ontology provide unambiguous meanings for terms inside itself and they are independent of readers as well as content. Given the nature of an ontology,

an ontology provides a vocabulary as well as a machine-processable common understanding of the topics.. The meanings of the terms in an ontology can naturally be communicated between all users as well as various applications (Gasevic & Djuric, 2006). The main purpose of employing ontologies is to share and reuse the knowledge by applications (Sugumaran, 2016). Since we could freely share and reuse ontologies in a domain of knowledge, it is possible to build certain knowledge bases which offer specific descriptions within the domain of specific knowledge. A formerly developed ontology that captures structure of certain knowledge domains can be later reused by others without any need to perform ontological analysis to build the same ontology (Gasevic & Djuric, 2006). A Semantic Web is a web of linked data to represent various kinds of knowledge domains. Within the context of Semantic Web, ontologies , as “a partial conceptualization of a given knowledge domain, shared by a community of users, have been defined in a formal, machine readable language in order to share semantic information across automated systems” (Jacob, 2003). An ontology defines concepts in terms of both the types (or classes) and properties of things as well as the semantic relationship between those defined concepts (Jacob, 2003).

1.2 SPARQL

RDF, a framework for the purpose of describing information or resources on the web, was designed to be read and interpreted by computers (W3 Schools, 2015). Features of RDF make it capable of facilitating data merging regardless of schemas as well as supporting the evolution of schemas over time (W3C.org, 2014). RDF takes the format of URIs (Uniform Resource Identifiers) to denote the relationship between things as well as both ends of the link, which is normally known as a “subject-predicate-object” triple. This specification of RDF defines “the syntax and semantics of the SPARQL query language for RDF” (W3C, 2008).

SPARQL (SPARQL Protocol and RDF Query Language) is the standard semantic query language as well as protocol for both linked data and RDF databases (RDF triplestores) (Ontotext, 2015). SPARQL was not originally designed to fulfill the needs of querying relational data (DuCharme, 2013) but as a special query language that gives users freedom to query inside RDF dataset, a whole set of triples (subject-predicate-object). SPARQL has SQL-like querying syntax structure but with several distinct differences. RDF Schema (RDFS) is a language for writing ontologies. Beside RDF, SPARQL can query an owl ontology file as well. RDF defines how to write stuff but

OWL (W3C Web Ontology Language), a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things (W3C.org, 2013), defines what to write and it adds semantic meanings to the RDF structures. OWL specification defines the semantic content to be written in with RDF in order to construct valid ontologies (Stackoverflow, 2018). Apache Jena Fuseki was originally designed and developed to function as a SPARQL server. It is capable of running "as an operating system service, as a Java web application (WAR file), and as a standalone server" (Apache Jena, 2011). Apache Jena Fuseki server is compatible with OWL and supports uploading owl ontology files and hosts it to be queried by SPARQL query submission.

1.3 Limitation of Previous Work

For a specific domain of knowledge, information could be stored across both relational databases and ontologies. The content in relational databases and ontologies are usually complementary to each other. In order to retrieve complete information, we might face the challenge of querying information from both relational databases and ontologies. Achieving efficient interoperability in between relational databases and ontologies encourages many solutions to unify data sources by making transformation in between relational databases and ontologies. Research on data mapping or transformation in between relational databases and ontologies (Database-to-Ontology or Ontology-to-Database), has gained much attention over the years. Normally, there are two types of approaches for handling Database-to-Ontology mapping: (a) approaches that require the generation of ontologies from existing databases (b) approaches that require the mapping databases to existing ontologies. Ghawi and Cullot (2007) argue that transforming structured data like databases, semi-structured data like XML documents, and/or non-structured data like web pages or data of other types to local ontologies could maintain the semantic of these data source. They proposed an approach along a tool "DB2OWL" to automatically generates an ontology from a relational database. The approach they proposed is looking for some particular cases of database tables and according to them the method decides which ontology component is created from which database. As for the approaches to map an existing database to an existing ontology, Bizer (2003) proposed D2R, a declarative language (XML-based) to describe mappings between RDBMS schemata and OWL ontologies. In D2R, class maps, which assign concepts of ontology to the

database sets are used to define basic mapping of concepts. All attributes of classes and relations are mapped. A significant feature of D2R is that D2R gives the access to flexible mapping of RDBMS structures by integrating SQL queries in the mapping rule. An approach to transform an ontology to relational databases is to make use of OWL API to manipulate OWL and save it as a format that relational databases can take (The OWL API, 2016). The goal of both kinds of research is to unify relational and ontology datasets into one, which enhances semantic interoperability as well as makes it more efficient and much easier to query the data. However, data mapping or transformation between relational databases and ontologies could suffer from several major drawbacks. One of the major drawbacks shared by both Ontology-to-Database and Database-to-Ontology transformation is that the data transformation or mapping requires a dedicated offline and expensive step to be performed before having the system ready for users. Other drawbacks are highlighted below.

1.3.1 Drawbacks of Ontology-to-Database Transformation

1.3.1.1 Loss of relationships

When transforming ontologies into RDBMS, one of the major drawbacks is the potential loss of most interlinks inside an ontology or in between ontologies. Ontologies are in the format of rich graphs, and it captures a lot of structured relationships with rich semantics. “Ontology aims to substantiate the rich variety of semantic relations” (Zhang, 2007). However, RDBMS could only keep a few types of relationships, such as “IS-A” relationships after the data transformation from ontologies, but some other types of relationships that were originally included by ontologies, such as symmetric relationships, could not be restored.

1.3.1.2 Loss of ontology complexity & rich semantic meaning

Another major drawback associated with data transformation from ontologies to RDBMS could be the loss of ontology complexity. The schema of an ontology is large and complex, while RDBMS normally has a relatively simple and smaller schema. In other words, the focus on formal semantics in ontologies is much stronger than in RDBMS (Konstantinou et al., n.d.). Ontology languages are most expressive in the sense of expressing more semantic concepts than database languages which only include constructs for defining or extracting data (Martinez-Cruz et al.,

2012). Transformation from ontologies to RDBMS could sacrifice the complexity of ontology as well as result in loss of rich semantics from original ontologies.

1.3.1.3 Inconsistency and maintenance issue

Mapping from ontologies to RDBMS would also result in inconsistency as well as maintenance issues. Ontologies enjoy open world assumption, and an ontology is capable of being connected to other ontologies. However, after being transformed from ontologies to RDBMS, which has a relatively closed world assumption, the original ontology is no longer consistent with the rest of other ontologies in the same domain of knowledge. The connections with other ontologies would break. Any new ontologies inside the domain of knowledge must be transformed into matching relational databases for the purpose of maintenance.

1.3.2 Drawbacks of Database-to-Ontology Transformation

1.3.2.1 Scalability

Transformation from RDBMS to ontologies also comes with some major concerns. When compared to RDBMS, ontologies suffer from the problem of scalability. An ontology, the data instances of which are normally represented in plain RDF or OWL file, is not very efficient in managing data instances. This nature of ontologies makes maintaining large amounts of data in a single ontology file is a practice with low efficiency. Transforming RDBMS into ontologies will be at the price of losing scalability as well as efficiency.

1.3.2.2 Familiarity

First defined in 1970, relational databases enjoy a long and prosperous history of research and development in both academia and industry. When compared to relational databases, ontologies in the domain of *Semantic Web* are young and have not been explored in full potential. As for the query language, people are more familiar with query language for relational databases, SQL than SPARQL which helps query the schema or data from ontologies. Besides, Query answering in ontologies is more difficult than that in relational databases, which could also lead to scalability problems.

1.3.2.3 Inconsistency

The transformation from relational databases to ontologies will cause some problems of inconsistency. Some key features of relational databases like indexing or query optimization would be disregarded during the data transformation process from RDBMS to ontologies.

Chapter 2 - Proposed Solution and Architecture Overview

2.1 Architecture Overview

In order to avoid the potential drawbacks discussed in Chapter 1, we propose the SQLogy, an integrated query engine for relational and ontology databases. SQLogy is driven by the need from end users to query both RDBMS and ontologies by the most popular SQL queries without data transformation or mapping in between RDBMS and ontologies. SQLogy makes it accessible for an end user to query data by a single standard SQL query without concerning the data sources in the back. SPARQL queries are not required for querying an ontology file.

Given an example in the domain of biology (illustrated in Figure 1), data about gene and protein are invariably interlinked and sometimes could be both stored in the format of ontology, however other biological data could be stored and maintained by RDBMS. Our overarching goal here is to make it available for an end user to query bio data from the heterogeneous data systems involving both RDBMS and ontologies with only SQL queries without concerning the original source format of the data or providing corresponding SPARQL to query data stored inside ontologies.

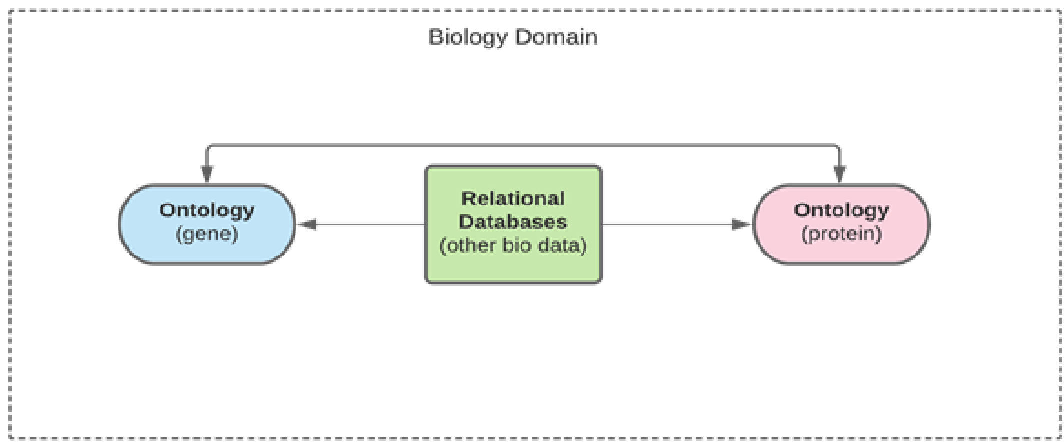


Figure 1. An Example of Biological Data Stored in Both RDB and Ontologies

The integrated query engine SQLogy is implemented as a web application that offers users access to query data from both relational databases (Postgres) and the ontology (stored on Apache Jena Fuseki server) with standard SQL queries. The overall architecture of this proposed integrated

query engine web application SQLogy is illustrated in the following figure which consists of three major parts: the user interface layer, the query parsing layer, and the back end.

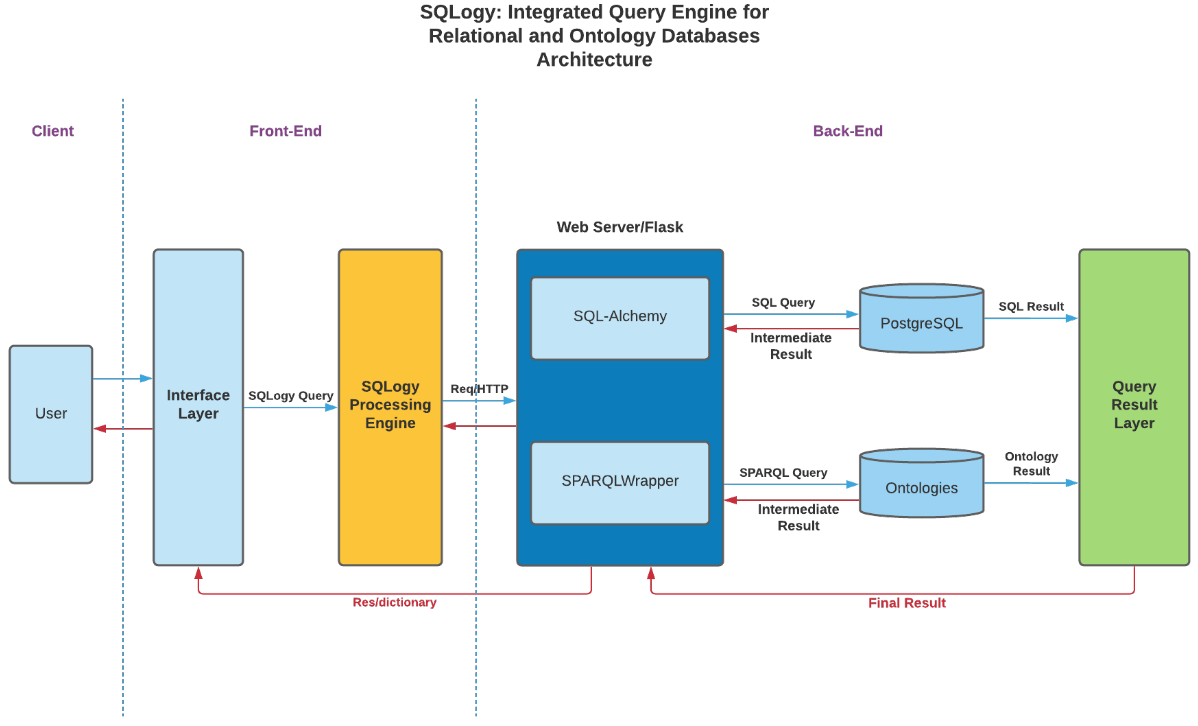


Figure 2. The Overall Architecture of Proposed SQLogy

2.1.1 The SQLogy User Interface Layer

A user interacts with the *Interface Layer of SQLogy* by providing a SQLogy query (which is an extended version of SQL query) and receiving results. Figure 3 illustrates the user interface of SQLogy after it gets loaded. The end user inputs a standard SQL query at the “SQL Query Input” box. After entering the standard SQL query, the user can click the button “Click for Result” to submit the SQL query to the next *Query Parser Layer* to get the input query evaluated and parsed. Newly constructed sub-SQL query or SPARQL query is used to query the data in order at corresponding data systems. The final query result is returned to the front-end of SQLogy and then gets presented as a tabular result in the “Query Output” section.

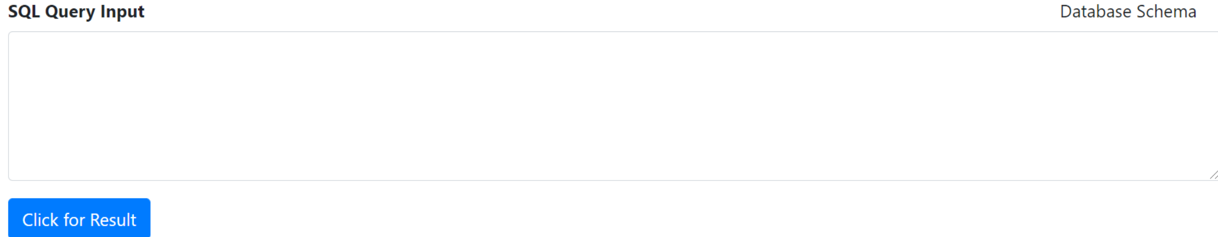


Figure 3. The User Interface of SQLogy Query Engine Web App

"Database Schema" is provided right above the right upper corner of the "SQL Query Input" box. It offers an end user an access to understand the schema of the experimental databases. The user can simply hover the mouse over the "Data Schema" section to get the details of the schema presented right below the "SQL Query Input" box. The database schema shows all related data tables from both Postgres database and ontology dataset on Apache Jena Fuseki server as tables within a single database. Users assume all the data is stored inside a Postgres relational database and don't need to concern over querying an ontology dataset with a SPARQL query. The Figure 4 illustrates the "Database Schema" of the experimental database for SQLogy.

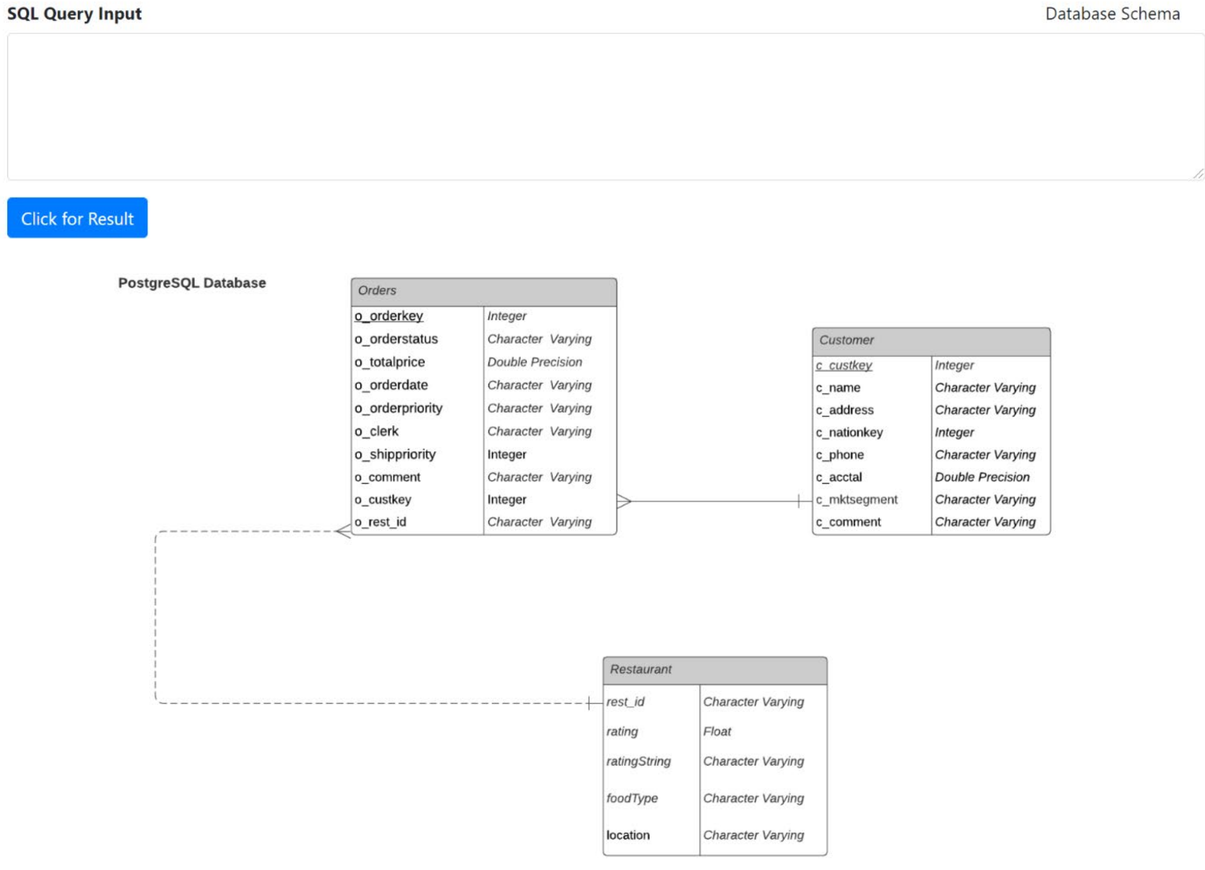


Figure 4. Database Schema of SQLogy Query Engine

2.1.2 SQLogy Query Parsing Layer

The *SQLogy Query Parsing Layer* is responsible for taking and evaluating the SQLogy query input, parsing and then constructing corresponding data queries, and finally distributing them to appropriate data sources in a certain order. A new standard SQL query gets generated to retrieve the information stored in RDBMS (PostgreSQL databases) if needed. A SPARQL query is composed to query the ontology dataset on Apache Jena Fuseki server if the ontology data is involved in the process of getting the final result. SQLogy in this thesis research project takes care of five major use cases:

- *Use Case 1*: SQLogy only needs to query data stored in the PostgreSQL database.

- *Use Case* 2: SQLogy only needs to query data stored as an owl ontology file on Apache Jena Fuseki server.
- *Use Case 3*: a) SQLogy needs to query data from both PostgreSQL database and ontology (.owl file) on Apache Jena Fuseki server. b) Only the data of attributes from the owl file on Apache Jena Fuseki server is required in the final result. c) It must first go to PostgreSQL to retrieve all the qualified restaurant ids (stored as a foreign key attribute in PostgreSQL database) with a new sub-SQL query and then go to Apache Jena Fuseki server querying the restaurant ontology with a SPARQL that includes all retrieved restaurant ids from PostgreSQL database.
- *Use Case 4*: a) SQLogy needs to query data from both PostgreSQL database and ontology (.owl file) on Apache Jena Fuseki server. b) Only the data of attributes stored in the PostgreSQL database is required in the final result. c) It must first obtain all the qualified restaurant identifiers (URIs) from the restaurant ontology file with a SPARQL query on Apache Jena Fuseki server. After transforming all obtained restaurant identifiers into restaurant ids, SQLogy is ready to query needed results with a sub-SQL query in the PostgreSQL database.
- *Use Case 5*: a) SQLogy needs to query data from both PostgreSQL database and ontology (.owl file) on Apache Jena Fuseki server. b) Data from both PostgreSQL database and the restaurant owl ontology file on Apache Jena Fuseki server participates in the final result. c) Similar to *Use Case 4*, SQLogy must first retrieve all qualified restaurant ids along all other required attributes by a sub SPARQL query on Apache Jena Fuseki server and then use all qualified restaurant ids to query needed columns in PostgreSQL database by a sub-SQL query. d) Combine results from both data sources through a shared key (restaurant ids) into a final result.

2.1.3 The SQLogy Back-end

Newly constructed sub queries from in each of the major use cases mentioned above get submitted to the corresponding query execution layer to obtain the data. We take advantage of *SPARQLWrapper* to talk to Java based Apache Jena Fuseki server from a popular Python web framework *Flask* as well as one of its extensions *Flask-SQLAlchemy*. *SPARQLWrapper* is developed as a Python wrapper around a SPARQL query service to execute queries remotely. It

assists in creating the query invocation and, possibly, converting the result into a more manageable format (Herman et al., 2016). We query the owl ontology file stored on Apache Jena Fuseki server with a newly constructed SPARQL query. A sub-SQL query will also be constructed and processed by the web server to eventually query the PostgreSQL database if needed. Intermediate results from either PostgreSQL or the ontology file on Fuseki server can be used to further query other information. *Query Result Layer* collects query result(s) from either or both data source(s) and then constructs the final result, which eventually gets returned as a response in the format of a Python dictionary and presented to the end user.

2.2 Advantages of SQLogy against Previous Work

The following are the major reasons why SQLogy is very much needed for end database users in certain cases.

2.2.1 Heterogeneity of Data Sources (RDBMS and Ontologies)

In the era of Big Data and Artificial Intelligence, one of the biggest challenges is streamlining data processes as it requires making use of heterogeneous data streams (Surani, 2020). Querying data from disparate structured, unstructured, and semi-structured sources can be a challenging work due to the different nature of data sources as well as corresponding query language. SQL query is used to query RDBMS and it is well integrated and supported by all RDBMS products. SPARQL was designed specifically as a query language to query RDF or ontology data files. Unfortunately, directly querying both RDBMS and ontologies by a single type of query language is not viable. Given this particular situation, SQLogy is a great option to take up the challenge of heterogeneity of data sources when both RDBMS and ontologies are presented as data sources.

2.2.2 Popularity of SQL Queries

SQL (Structured Query Language) is a domain-specific language used in programming and was designed for manipulating data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e., data incorporating relations among entities and variables. When compared to SQL, SPARQL is less popular to general database users and is specifically used to query RDF or ontology data. The syntax of SPARQL is somewhat close to that of SQL.

However, SPARQL is still different from SQL and not quite easy to follow for beginners. It would be much easier and quicker for end users to directly work with SQL that they are much more familiar and more comfortable with. The main purpose of designing and developing SQLogy is to make it accessible for end users to access data from both RDBMS (PostgreSQL) and ontologies with standard SQL queries. Users of SOLogy could assume all the data is stored inside RDBMS and SQLogy takes care of the user input SQL and prepares needful sub queries to properly accomplish the query mission.

2.2.3 Problematic Data Transformation

Most database users are way more familiar working with standard SQL to query RDBMS. However, they might have to face the challenges when data is stored as ontologies. Data transformation between different data systems is such a daunting and challenging task that database developers must handle fastidiously before end users. Data transformation from an ontology to RDBMS potentially has the risk of losing some semantic meanings or relationships inside out ontology files. Another problem associated with data transformation happens when updating an existing ontology. Updating an existing after the initial transformation requires further mapping or transformation. If we keep the original data source the way it is, regardless of what new manipulation or modification applied to the original data source, there is no need for any further transformation, which saves time and makes existing data sources more invulnerable to changes.

2.3 User Interface (function explained in detail)

The general structure and function of SQLogy's interface has been introduced in section 2.1.1. Here in this section, more features of the user interface are introduced and explored.

2.3.1 SQL Query Input

SQLogy provides a text input box for an end user to input a standard SQL query (illustrated in Figure 5). End users can use the "Database Schema" section as a reference to the schema of the database where all tables are assumed to be inside RDBMS (PostgreSQL).

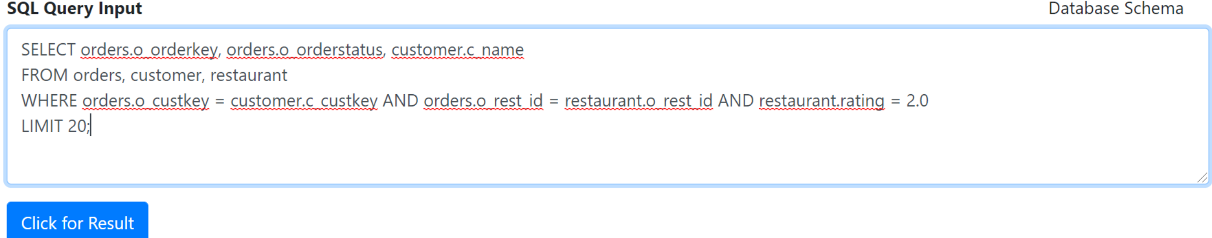


Figure 5. SQL Query Input Section of SQLogy Query Engine

2.3.2 SQL Query Output

After a user inputs the full SQL query and hits the button of "Click for Result", the final query result is presented at the *Query Output* section down below. The result takes a tabular format with all the columns needed based on the original SQL query input. Beside directly showing the result on the webpage, SQLogy also provides an access for a user to export the full result as a csv file simply by clicking the "Export Result" button located above the right upper corner of the *Query Output* section. What is more, SQLogy also provides the total query execution time (in milliseconds (ms)) associated with executing the input query.

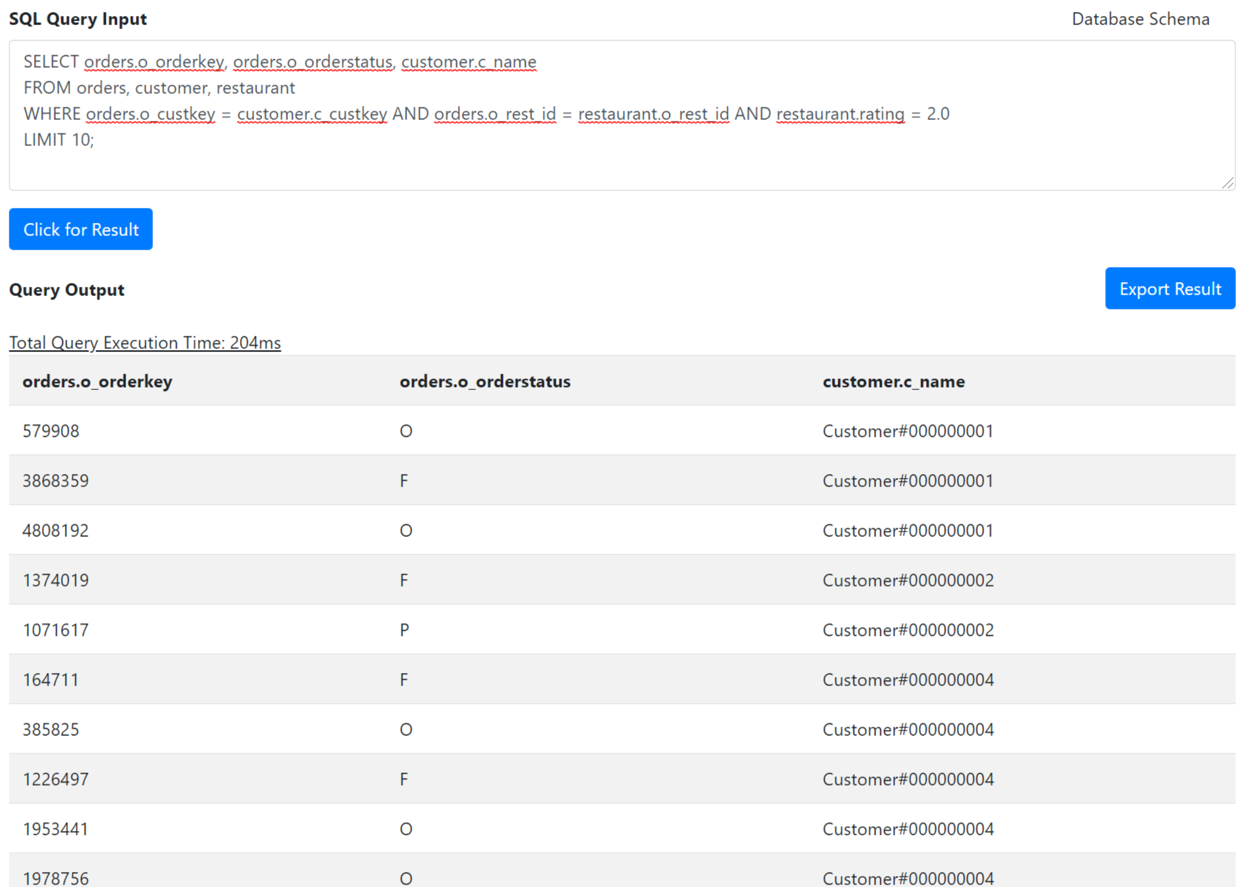


Figure 6. SQL Query Output Section of SQLogy Query Engine

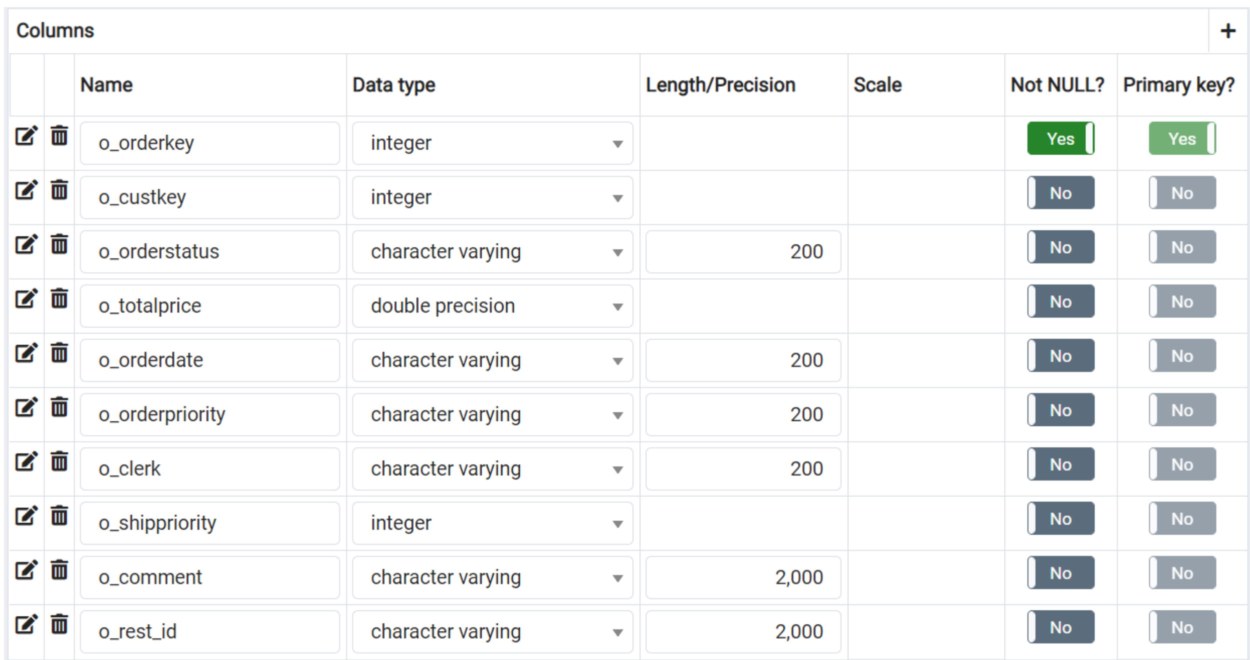
	A	B	C
1	**orders.o_orderkey**	**orders.o_orderstatus**	**customer.c_name**
2	579908	O	Customer#000000001
3	3868359	F	Customer#000000001
4	4808192	O	Customer#000000001
5	1374019	F	Customer#000000002
6	1071617	P	Customer#000000002
7	164711	F	Customer#000000004
8	385825	O	Customer#000000004
9	1226497	F	Customer#000000004
10	1953441	O	Customer#000000004
11	1978756	O	Customer#000000004
12	1944711	P	Customer#000000004
13	1192231	O	Customer#000000004
14	2459619	O	Customer#000000004
15	3251169	O	Customer#000000004
16	4320612	F	Customer#000000004
17	2630562	F	Customer#000000005
18	1959075	F	Customer#000000007
19	1894087	F	Customer#000000007
20	1485505	O	Customer#000000007
21	3211909	F	Customer#000000007

Figure 7. An Example of Exported Query Output CSV File

Chapter 3 - Use Cases

3.1 Data Source

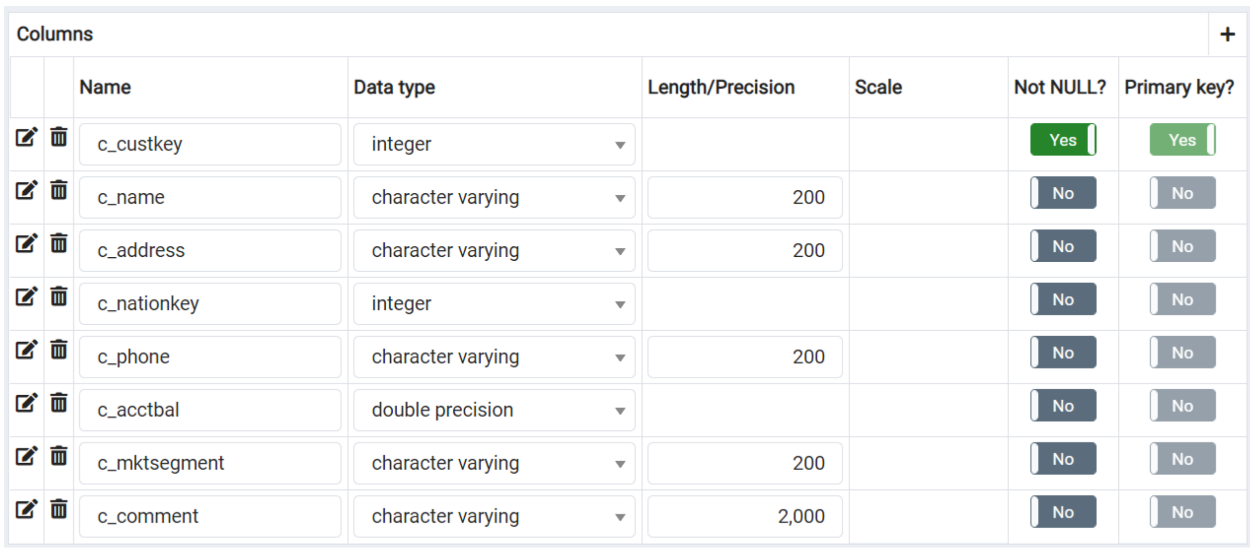
The data set used for this thesis project consists of two parts: tabular dataset about customer and orders and an ontology dataset (owl file) about restaurant. Both the customer and the orders datasets were generated by the database population program DBGEN for the use of TPC-H benchmark and stored inside PostgreSQL database. The *Customer* table stores information about the customers' name, address, phone number, etc., while the *Orders* table provides access to order related information including order price, order date, order priority, etc. Figure 8 and Figure 9 show the column properties of the *Customer* and the *Orders* table, respectively. The primary key of the *Customer* table is "c_custkey" which serves as the foreign key of the *Orders* table of which the primary key is "o_orderkey".



Columns

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
o_orderkey	integer			Yes	Yes
o_custkey	integer			No	No
o_orderstatus	character varying	200		No	No
o_totalprice	double precision			No	No
o_orderdate	character varying	200		No	No
o_orderpriority	character varying	200		No	No
o_clerk	character varying	200		No	No
o_shippriority	integer			No	No
o_comment	character varying	2,000		No	No
o_rest_id	character varying	2,000		No	No

Figure 8. Column Properties of the Orders Table



Columns

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
c_custkey	integer			Yes	Yes
c_name	character varying	200		No	No
c_address	character varying	200		No	No
c_nationkey	integer			No	No
c_phone	character varying	200		No	No
c_acctbal	double precision			No	No
c_mktsegment	character varying	200		No	No
c_comment	character varying	2,000		No	No

Figure 9. Column Properties of the Customer Table

Beside the referential column "c_custkey" from the *Customer* table, the *Orders* table also has a column "o_rest_id" which serves as the bridge between the PostgreSQL database and the *Restaurant* ontology. Information or knowledge about restaurants, originally provided as an owl ontology by Mooney.net, is stored on Apache Jena Fuseki server. The *Restaurant* ontology provides access to information about a restaurant's label, food type, location, numeric rating (from 1.0 to 5.0), and the textual rating (good or bad). Each restaurant instance inside ontology owns a unique identifier (URI) (e.g., "http://www.mooney.net/restaurant#ID_polloSalsa1"). All restaurant instances share the identical leading part of the URIs up to "#", however, the rest of the restaurant URI is unique to individual restaurant instances. In order to build the connection with the *Orders* table inside PostgreSQL database, the unique part of the URI for each restaurant instance (given the URI: "http://www.mooney.net/restaurant#ID_polloSalsa1", the substring needed is "ID_polloSalsa1".) is being used as the restaurant id. It matches the value of the column "o_rest_id" in the *Orders* table. This special "restaurant id" is the key connection between RDBMS and restaurant ontology on Apache Jena Fuseki server.

3.2 Six Use Cases (Five Major Use Cases & One Special Use Case)

SQLogy is capable of handling five major use case scenarios which have been briefly mentioned in Chapter 2. Besides, there is also a special use case scenario that demonstrates how SQLogy

could adapt to changes directly on the existing ontology dataset. Here in this section of Chapter 3, all major use cases will be elaborated with more details and samples.

3.2.1 Use Case 1: Only Query PostgreSQL Database with the Original SQL Query

Use Case Description

A user of SQLogy inputs a standard SQL query which only queries PostgreSQL for information about customers or/and orders. In this specific use case, there is no need to make changes against the original SQL query input. Figure 10 in the following section demonstrates the workflow in the back end of the query engine application in Use Case 1.

Use Case Structure

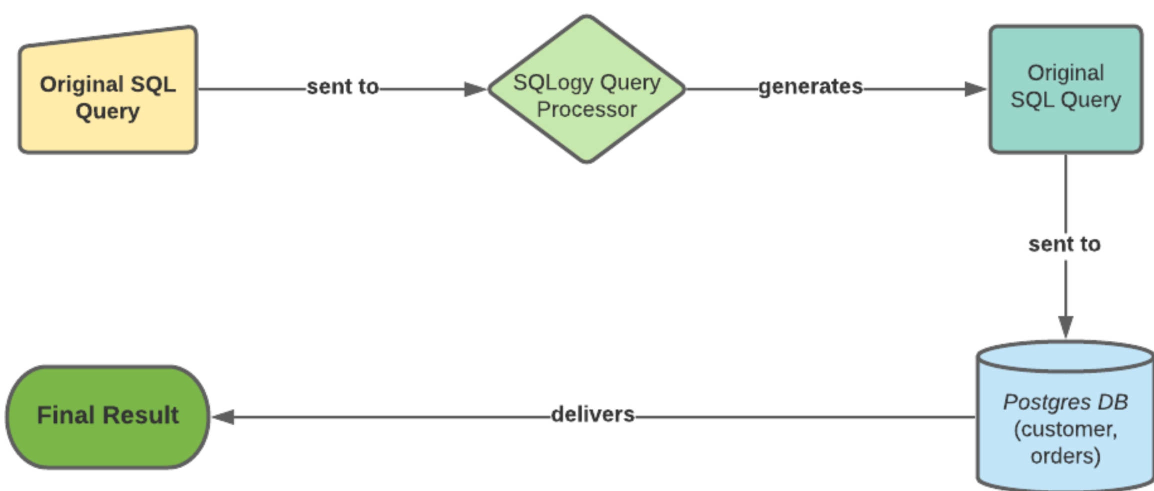
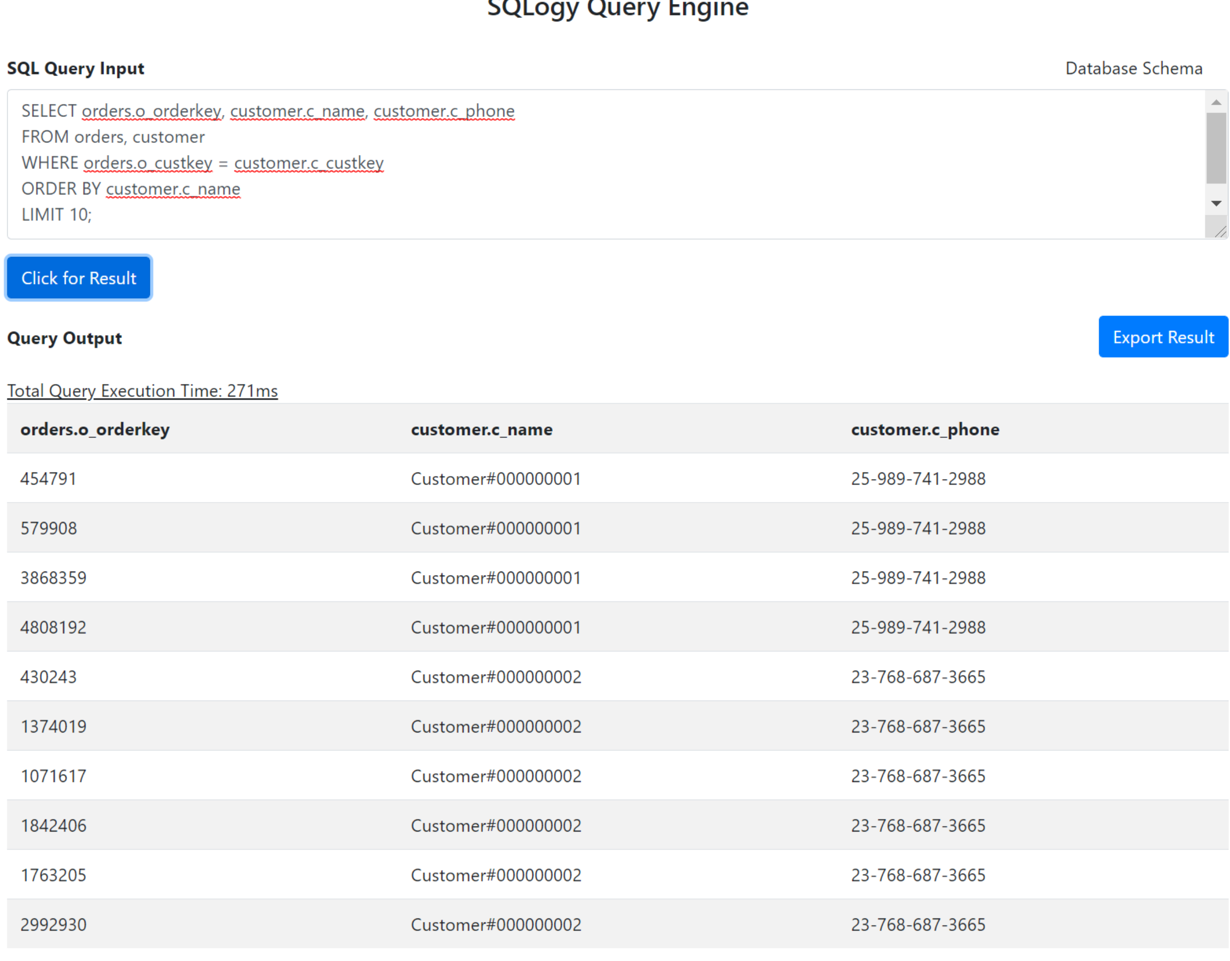


Figure 10. Use Case 1 Workflow Chart

Given the workflow chart of Use Case 1 above, after the original SQL input is transferred to SQLogy Query Processor and categorized to be a case of Use Case 1, since there is no need to make any changes against the original SQL query input or generate any type of sub queries, the original SQL query input gets directly submitted to Postgres to query database for the target result. The query result will be delivered back to the front end of SQLogy and presented to the end user.

Example

Here we provide a working example of Use Case 1. The example SQL query input and the result is shown in the following figure. The user only requests to query data about "o_orderkey", "c_name" and "c_phone" which are all available in Postgres database. No restaurant data is even involved in the "WHERE" section of this input query, which indicates that the ontology owl file on Apache Fuseki server will not be involved in the whole process in this specific use case. The original query input will all be sent to Postgres to query the data. The final query result consists of all three columns of data retrieved from Postgres database and the end user also could download the final result as a csv file.



orders.o_orderkey	customer.c_name	customer.c_phone
454791	Customer#000000001	25-989-741-2988
579908	Customer#000000001	25-989-741-2988
3868359	Customer#000000001	25-989-741-2988
4808192	Customer#000000001	25-989-741-2988
430243	Customer#000000002	23-768-687-3665
1374019	Customer#000000002	23-768-687-3665
1071617	Customer#000000002	23-768-687-3665
1842406	Customer#000000002	23-768-687-3665
1763205	Customer#000000002	23-768-687-3665
2992930	Customer#000000002	23-768-687-3665

Figure 11. Use Case 1 Sample Query Input & Result

3.2.2 Use Case 2: Only Query Ontology Data in Apache Fuseki Server

Use Case Description

An end user inputs a standard SQL query which only queries restaurant owl ontology stored on Apache Jena Fuseki server. For this particular case, since it does not involve any data about customers or orders, the Postgres database is completely disregarded. A new SPARQL query gets constructed to query the restaurant ontology. SQLogy Query Processor evaluates and parses the input SQL query and maps its element to the general structure template of a SPARQL query as well. The "SELECT" part of the input SQL query gets mapped into the corresponding "SELECT" part of the new SPARQL query. "WHERE" the conditional selection part of the SQL input query gets mapped to the "FILTER" section of the new SPARQL query. The Figure 12 in the following section demonstrates the workflow in the back end for Use Case 2.

Use Case Structure

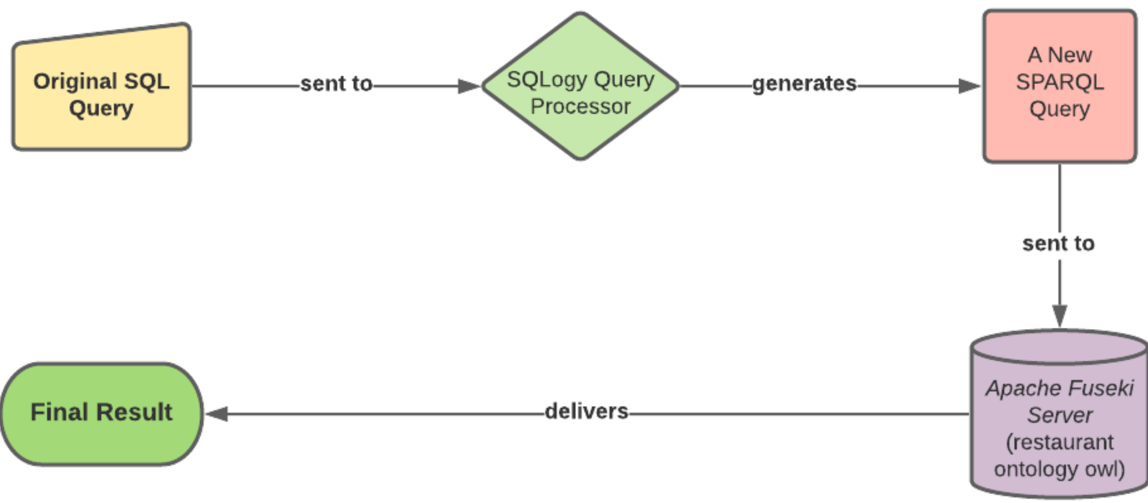


Figure 12. Use Case 2 Workflow Chart

Given the workflow chart of Use Case 2 above, after submitted to SQLogy Query Processor, the original SQL query input is evaluated and categorized to be a case of Use Case 2. Since there is no need to query Postgres and only the restaurant ontology needs to be queried, the original SQL query input gets parsed and transformed into a new SPARQL query to help retrieve information

on Apache Jena Fuseki server. The query result is returned to the front end of SQLogy and presented to the end user.

Example

A working example of Use Case 2 is explained here. The example SQL query input and the result are shown in Figure 13. The user only requests to obtain the "ratingString" and the "label" of all qualified restaurants. Both columns are only accessible from the restaurant owl ontology on Apache Jena Fuseki server. Neither *Customer* nor *Orders* is even involved in the "WHERE" section of this input query, which indicates that only the restaurant ontology is involved in this specific use case. The original SQL query input is first transformed into a new matching SPARQL query and then submitted to Apache Jena Fuseki server to query the restaurant ontology dataset. The final query result consists of data about the restaurant's text rating as well as their labels retrieved from the Fuseki server, and the end user owns the options to download the final result as a csv file as well.

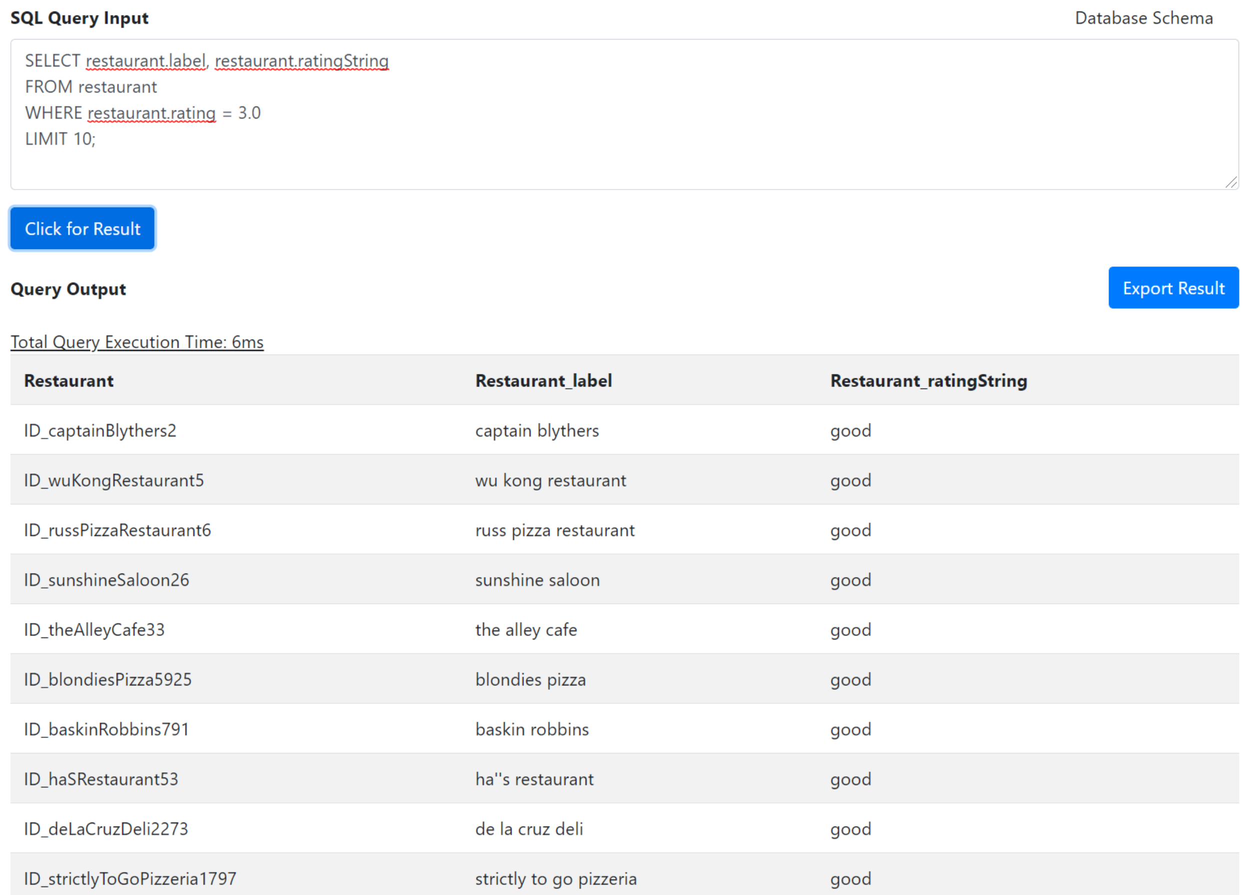


Figure 13. Use Case 2 Sample Query Input & Result

3.2.3 Use Case 3: Query Database in Postgres DB → Query Ontology on Fuseki Server

Use Case Description

In this particular use case, an end user provides SQLogy with a standard SQL query. However, when compared to the first two use case scenarios, SQLogy needs to query both Postgres database and the owl ontology on Apache Jena Fuseki server. A significant characteristic of this type of use case is that only the data from the owl ontology dataset on Fuseki server is needed for the final result. When examining the SQL query input, we only see that column(s) (attribute(s)) of restaurants are involved in the "SELECT" section of the SQL query. Meanwhile, in the "WHERE" section of the input query, there exists some conditional statement involving attributes of customer or/and order in Postgres. Given what we observe from the input query, SQLogy must construct a

new sub-SQL query first to retrieve all qualified restaurant ids (“o_rest_id”) based on the very last conditional statement regarding the *Customer* and the *Orders* table in Postgres. All qualified restaurant ids (“o_rest_id”) are integrated into the new SPARQL query to query the restaurant ontology on Jena Fuseki server for all needed columns of restaurant data (based on the “SELECT” section of original SQL query input). In this specific case, restaurant id, a unique identifier of each restaurant functions as the bridge to connect the Postgres database and restaurant ontology dataset on Apache Jena Fuseki server. Those restaurant ids are stored as a column (“o_rest_id”) in the *Orders* table to refer to the unique URIs of restaurant instances in restaurant ontology. However, string processing is required to transform the value of “o_rest_id” from Postgres into the value of restaurant URI, which is a unique value for every restaurant instance in the restaurant ontology. For instance, the value of the attribute “o_rest_id” in the *Orders* table is “ID_captainBlythers2”. The value of URI of the same restaurant in restaurant ontology is “<http://www.mooney.net/restaurant#ID_captainBlythers2>”. In order to retrieve other restaurant information from the ontology, all qualified “o_rest_id” values that have been queried from Postgres have to be transformed by adding a leading string “http://www.mooney.net/restaurant#” and then wrapped in between a set of two brackets “< >”. The Figure 14 in the following section demonstrates the workflow happening in the back for Use Case 3.

Use Case Structure

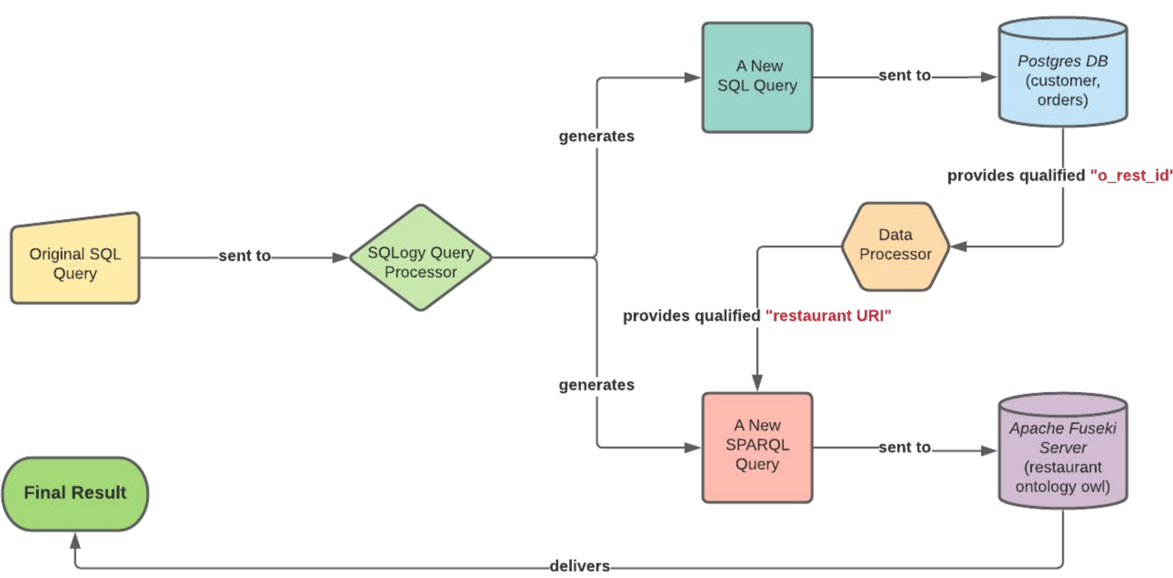
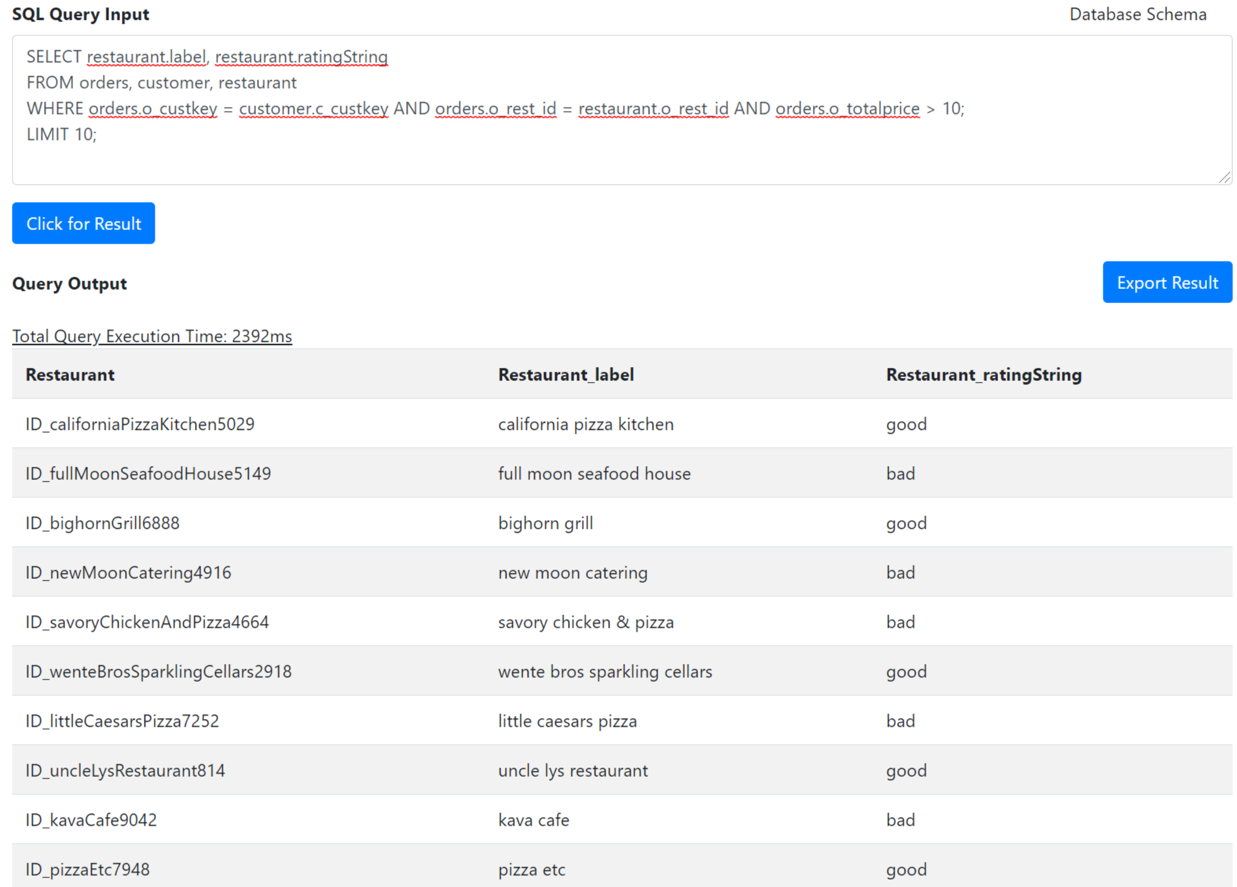


Figure 14. Use Case 3 Workflow Chart

The above flow chart (Figure 14) demonstrates how the back end works in Use Case 3. After being categorized as a case of Use Case 3, a sub-SQL query is constructed based on the original SQL query input and moves on to Postgres to extract all the qualified “o_rest_ids”. The data processor takes all qualified “o_rest_ids” and transforms them into matching restaurant URIs by adding the same prefix. A brand new SPARQL query is generated. It accepts all qualified “restaurant URIs” and gets sent to Apache Jena Fuseki server to query the restaurant ontology for all requested columns (attributes). Finally, the query result is delivered to the front end of SQLogy for review.

Example

Here in this section, we use an example to explain what happens inside SQLogy in User Case 3. The example SQL query input and the result are shown in Figure 15. The user wants to query the label (the name) and ratingString (“good” or “bad”) of all restaurants of which the order total price is over $10 dollars. Since the “SELECT” section of the input SQL query only consists of attributes from the restaurant ontology file on Apache Jena Fuseki server, the final result comes out of Fuseki server. However, the very last conditional statement in the “WHERE” section of SQL query input applies to the column of “o_totalprice” stored in Postgres. SQLogy has to first go to Postgres to retrieve all qualified “o_rest_id” from the *Orders* table with a sub-SQL query and then integrate them into a newly constructed SPARQL query to get the label and ratingString of all qualified restaurants from the ontology. Since we add a “LIMIT 10” condition statement, only ten of those qualified restaurant results are presented at the front end of SQLogy.



Restaurant	Restaurant_label	Restaurant_ratingString
ID_californiaPizzaKitchen5029	california pizza kitchen	good
ID_fullMoonSeafoodHouse5149	full moon seafood house	bad
ID_bighornGrill6888	bighorn grill	good
ID_newMoonCatering4916	new moon catering	bad
ID_savoryChickenAndPizza4664	savory chicken & pizza	bad
ID_wenteBrosSparklingCellars2918	wente bros sparkling cellars	good
ID_littleCaesarsPizza7252	little caesars pizza	bad
ID_uncleLysRestaurant814	uncle lys restaurant	good
ID_kavaCafe9042	kava cafe	bad
ID_pizzaEtc7948	pizza etc	good

Figure 15. Use Case 3 Sample Query Input & Result

3.2.4 Use Case 4: Query Ontology in Fuseki Server → Query Database in Postgres DB

Use Case Description

Like Use Case 3, Use Case 4 needs SQLogy to query both Postgres database and the ontology dataset on Apache Jena Fuseki server. However, the most significant difference between Use Case 3 and 4 is the order of querying Postgres and the ontology on Apache Jena Fuseki server. In this specific case, the user only wants to query the Postgres database for data about orders or/and customers because all columns provided in the "SELECT" section belong to tables in Postgres. Restaurant information like (label, location, food type, etc.) is not involved in the final result. Meanwhile, in the "WHERE" section of the input query, there exists a conditional statement about an attribute of restaurant ontology dataset on the Fuseki server. This nature of the original SQL

query determines the order of query execution: SQLogy must construct a new SPARQL query first to retrieve all qualified restaurant ids from the ontology dataset. Since all of those retrieved restaurant ids (or the identifiers of the restaurant instances) are in the format of URIs, in order to make use of all qualified restaurant ids to compose a new SQL query to get all needed columns in Postgres, SQLogy has to transform all restaurant URIs into rest_ids (by removing all surrounding brackets as well as the identical location string "http://www.mooney.net/restaurant#") and then integrate them into the new sub SQL query which gets submitted and queries the Postgres database. Like Use Case 3, o_rest_id functions as the key to connect both data systems. Figure 16 in the following section demonstrates the workflow in the back for Use Case 4.

Use Case Structure

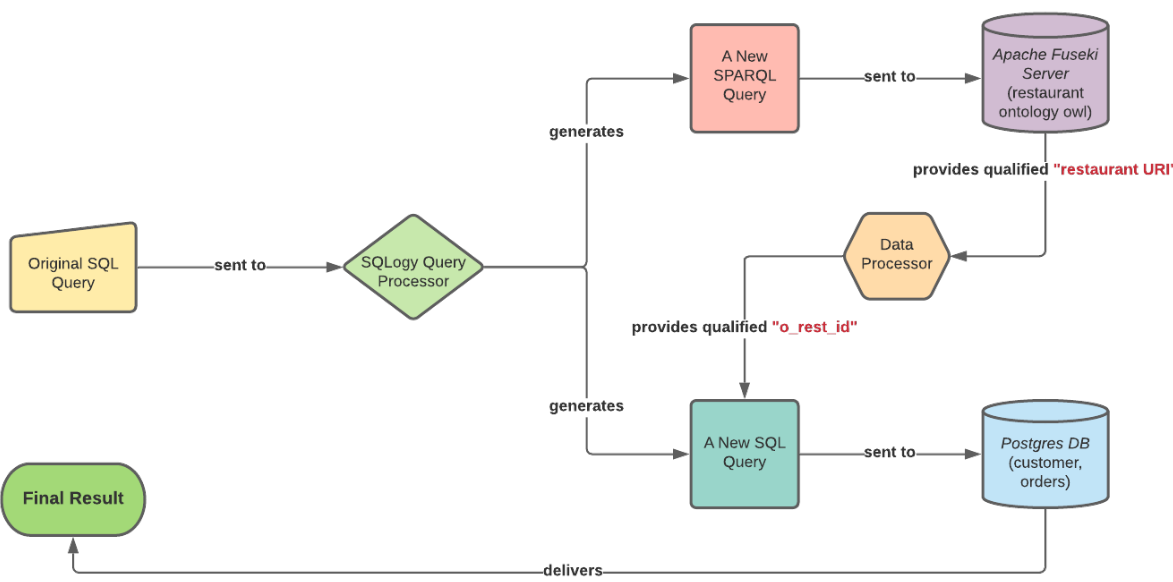


Figure 16. Use Case 4 Workflow Chart

The single original SQL input is initially submitted to the SQLogy Query Processor and falls into the category of Use Case 3 after examining both the "SELECT" and "WHERE" sections of the original SQL query input. Firstly, a new SPARQL query is constructed based on the content of the "WHERE" section in original SQL query input and then the query gets submitted to Apache Fuseki Jena server to retrieve all qualified restaurant ids (or all unique restaurant identifiers in the format of URIs). Since the format of all queried restaurant ids does not match restaurant ids (o_rest_id)

stored in the table of restaurant at Postgres, the Data Processor will assist in transforming them into the appropriate format. All the freshly processed restaurant ids will be integrated into the new sub-SQL query, which then gets sent to Postgres DB to complete the very last data querying job.

Example

The sample SQL query input we will use for Case 4 is illustrated in Figure 17. The user wants to query information about an order (order key and order status) as well as all associated customer names. No attributes of the restaurant ontology are identified in the "SELECT" section of the original query. The only conditional statement included in the original SQL query is "restaurant.ratingString = 'good'". The nature of this sample SQL query input determines that a SPARQL query needs to be constructed first to help query all qualified restaurant ids from ontology dataset on Apache Jena Fuseki server. Then, after being transformed into the correct format which follows the exact same format as all "o_rest_id" in Postgres, all extracted restaurant ids are integrated into the "WHERE" section of the new SQL query. The newly constructed sub-SQL query moves on to Postgres. Since we add a "LIMIT 10" condition, only ten of those qualified results are presented at the front end of SQLogy for the end user to review or export.

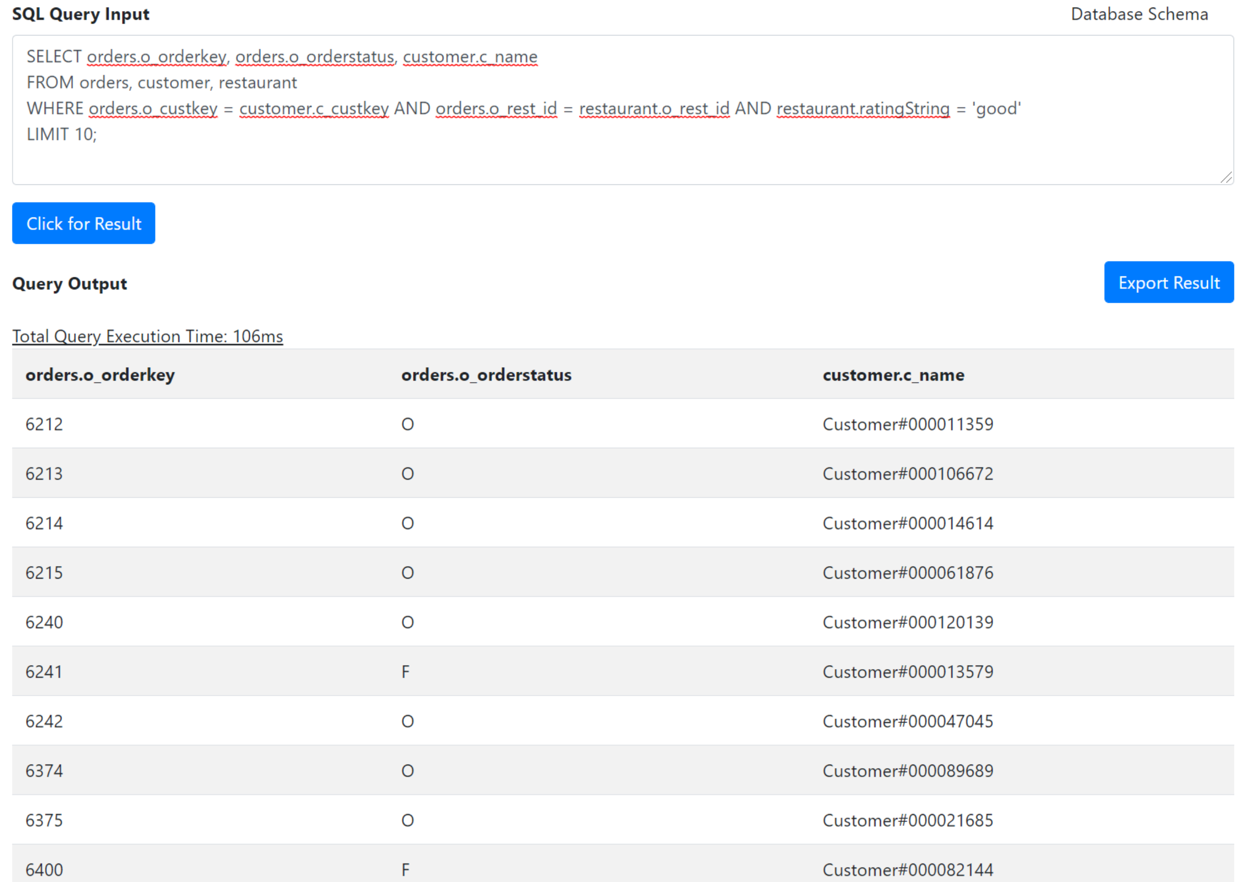


Figure 17. Use Case 4 Sample Query Input & Result

3.2.5 Use Case 5: Query Needed Columns from Both Ontology & Postgres and Combine the Results.

Use Case Description

For Use Case 5, SQLogy must query both Postgres database and the ontology (.owl file) on Apache Jena Fuseki server. Besides, data of columns (attributes) from both Postgres database and the ontology on Apache Jena Fuseki server participate in the final result. Use Case 5 can be regarded as an extension of Use Case 4 where the conditional statement in the "WHERE" section of the original SQL query input applies only to a column of the ontology dataset. Like Use Case 4, SQLogy needs to generate a new SPARQL query first to query the ontology first and then move on to querying Postgres database with a sub-SQL query. However, when compared to Use Case 4,

the most significant difference is that the final result expected by the user consists of columns (attributes) from both Postgres database and the ontology dataset on Apache Jena Fuseki server. Given the nature of input SQL query in Use Case 5, not only does the SQLogy need to first query the restaurant ontology for all qualified restaurant ids (the identifiers of the restaurant instances) by a proper SPARQL query but also any needed columns for the final results have to be queried out. After the transformation of all qualified restaurant identifiers, A sub-SQL query is created with all transformed "o_rest_ids" and then gets submitted to Postgres. The final result will be the combination of results from both data sources. Figure 18 in the following section demonstrates the workflow in the back end for Use Case 5.

Use Case Structure

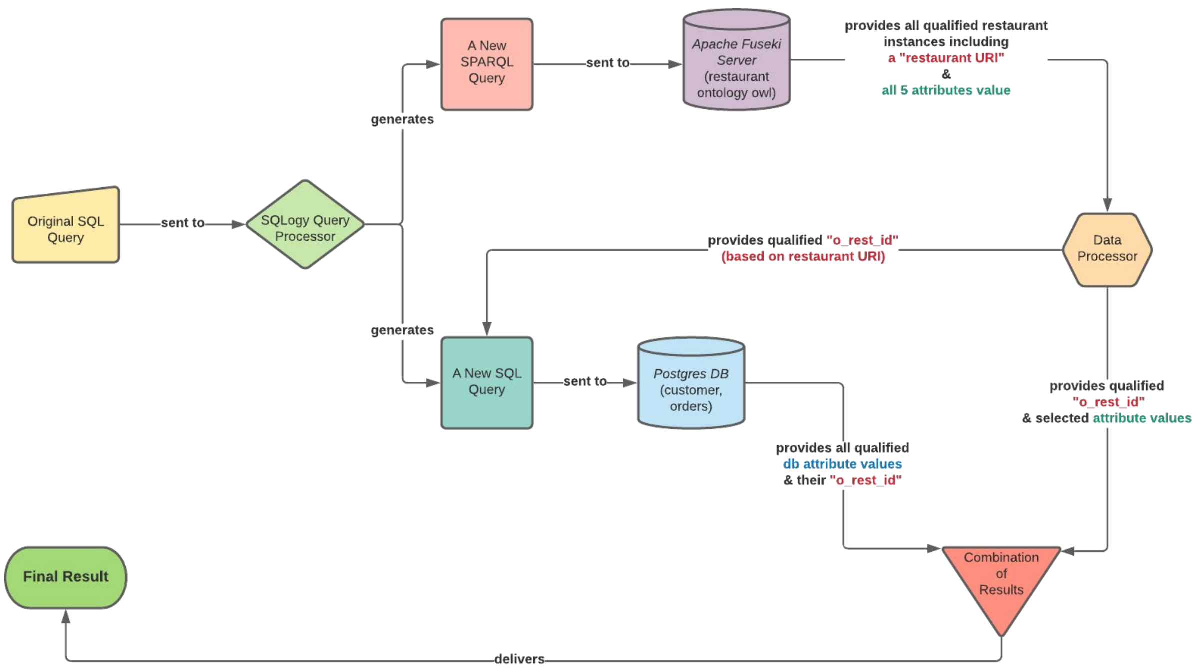


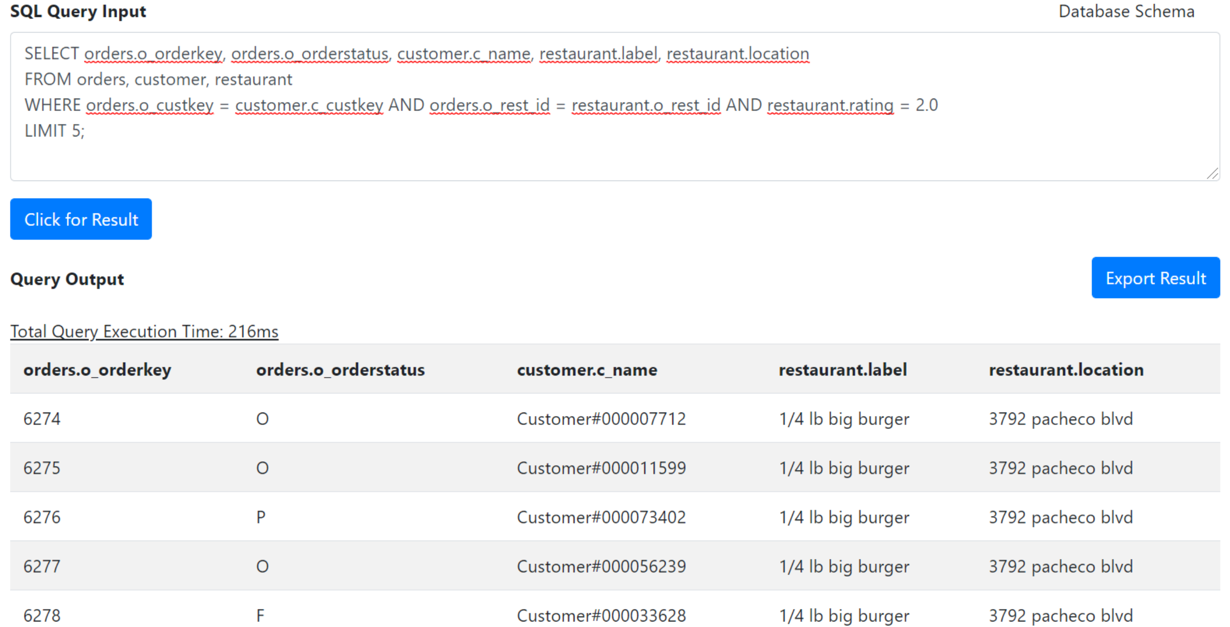
Figure 18. Use Case 5 Workflow Chart

The original SQL input is initially submitted to SQLogy Query Processor and categorized as a case of Use Case 5 after the evaluation of both the "SELECT" and "WHERE" sections of the original SQL query input. Firstly, a new SPARQL query is constructed based on the very last conditional statement on a restaurant's attribute in the "WHERE" section of original SQL query input. Then,

the SPARQL query gets submitted to Apache Jena Fuseki server to query the restaurant ontology for all qualified restaurant data instances (restaurant URIs along all their five attributes: “label”, “foodType”, “location”, “rating”, and “ratingString”). The result will be saved and labeled as “Fuseki Result”. Secondly, restaurant URIs of all qualified restaurants from the “Fuseki Result” need to be transformed into restaurant ids which share the identical format as the value of the column “o_rest_id” in Postgres. Thirdly, a sub-SQL query is created with restaurant ids transformed from the previous step and then used to query the Postgres database. The query result will be saved and labeled as “Postgres Result”. Finally, since both “Fuseki Result” and “Postgres Result” share the same key: restaurant id, both results will be combined based on the restaurant id (or “o_rest_id”) to be the final result where duplicates of records and unwanted columns (attributes) of restaurants are excluded. The final combined result gets delivered back to the front end for the user to review.

Example

Here we use a sample SQL query to illustrate the Use Case 5 in Figure 19. The user wants to query five different kinds of information about all qualified orders and only needs SQLogy to show 10 records. The columns (or attributes) the user expects to query are physically stored on both Apache Jena Fuseki server (both the label and location of the restaurant) and Postgres database (order key, order status, as well as associated customer name). The very last conditional statement inside the “WHERE” part of the original SQL query is about the restaurant's numeric rating. Given all the characteristics of this sample SQL query input, SQLogy determines that it fits the Use Case 5, followed by the generation of a new SPARQL query to help query all qualified restaurant instances (in the format of restaurant identifiers (URIs) + their five attributes) from the ontology dataset on Apache Fuseki server. The query result will be saved for later result combination. Then, the restaurant identifiers are being transformed into the correct format which matches the value of the “o_rest_id” column in Postgres. A SQL query is constructed with all qualified restaurant ids received from the previous step. It queries qualified columns in Postgres and the result is saved to be combined with the result from the ontology on Fuseki server. Since we add a “LIMIT 10” condition, only ten of these qualified results are presented at the front end of SQLogy.



orders.o_orderkey	orders.o_orderstatus	customer.c_name	restaurant.label	restaurant.location
6274	O	Customer#000007712	1/4 lb big burger	3792 pacheco blvd
6275	O	Customer#000011599	1/4 lb big burger	3792 pacheco blvd
6276	P	Customer#000073402	1/4 lb big burger	3792 pacheco blvd
6277	O	Customer#000056239	1/4 lb big burger	3792 pacheco blvd
6278	F	Customer#000033628	1/4 lb big burger	3792 pacheco blvd

Figure 19. Use Case 5 Sample Query Input & Result

3.2.6 Use Case 6: User Updates the Ontology Dataset (Supporting Ontology Update)

Use Case Description

We discussed some major drawbacks of mapping in between ontologies and RDBMS when working with both data systems. One of the major drawbacks when mapping an ontology to RDBMS is that the mapping or transformation could result in some maintenance issues. If we choose to map an existing ontology into RDBMS, any changes on the existing ontology after the transformation must come with a round of mapping or other type of maintenance. SQLogy, proposed in this thesis, helps to avoid such kind of remapping or further transformation every time when the existing ontology gets an update since SQLogy directly communicates with the ontology file and there is no need to map an ontology into an RDBMS element. The ontology dataset saved on Apache Jena Fusek server is accessible for any update, which saves much cost associated with any type of ontology transformation and mapping.

Example

Here we demonstrate how SQLogy supports working with an update of ontology with an example where the restaurant ontology gets inserted with a new restaurant instance. The user is trying to query a restaurant whose label (name) is 'chicken finger lover', which is not included in the original restaurant ontology. The query result is shown in Figure 20 where SQLogy fails to find a matching record in the existing restaurant ontology.

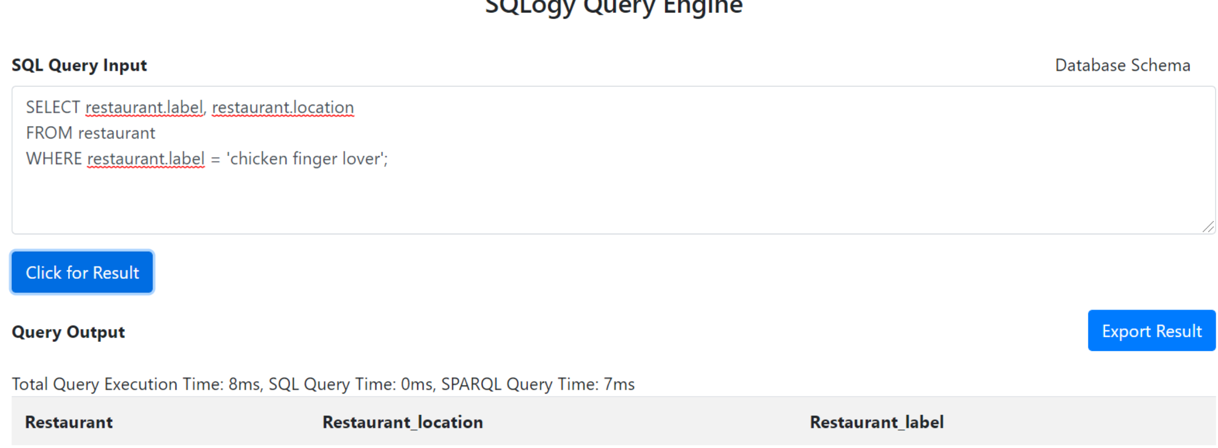


Figure 20. SQLogy Query Result for A Non-existing Restaurant Instance

We insert a new restaurant instance with the label of "chicken finger lover" by an SPARQL insertion query from a supporting SPARQL web application *SPARQL Query Engine*. *SPARQL Query Engine* is a simple web application which shares a similar user interface as our SQLogy. It directly connects to the Apache Jena Fuseki server and supports querying an ontology dataset on Apache Jena Fuseki server with a standard SPARQL query. The following Figure 21 illustrates the SPARQL query we could use to add this new restaurant instance to the existing ontology dataset.

SPARQL Query Engine

SPARQL Input

```
PREFIX mooney: <http://www.mooney.net/restaurant#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

INSERT
{
  mooney:ID_chickenfingerlover001 rdf:type mooney:Restaurant .
  mooney:ID_chickenfingerlover001 mooney:label 'chicken finger lover' .
  mooney:ID_chickenfingerlover001 mooney:rating 2.2 .
  mooney:ID_chickenfingerlover001 mooney:ratingString 'bad' .
  mooney:ID_chickenfingerlover001 mooney:foodType 'chinese' .
  mooney:ID_chickenfingerlover001 mooney:location '186 cambridge st' . }
WHERE
{
  FILTER NOT EXISTS
  {
   mooney:ID_chickenfingerlover001 rdf:type mooney:Restaurant .
  }
};
```

Click for Result

Figure 21. The SPARQL Query to Add a New Restaurant Instance through SPARQL Query Engine

After the insertion, we use the same SQL query in SQLogy to check whether the new restaurant instance "chicken finger lover" has been successfully added to the existing restaurant ontology. Just as what is shown in Figure 22 below, now we can successfully retrieve this new added restaurant instance from the restaurant ontology dataset on Fuseki server. The real update on ontology could be way more complicated than what the example shows here. The cost of mapping the changes could be considerably high.

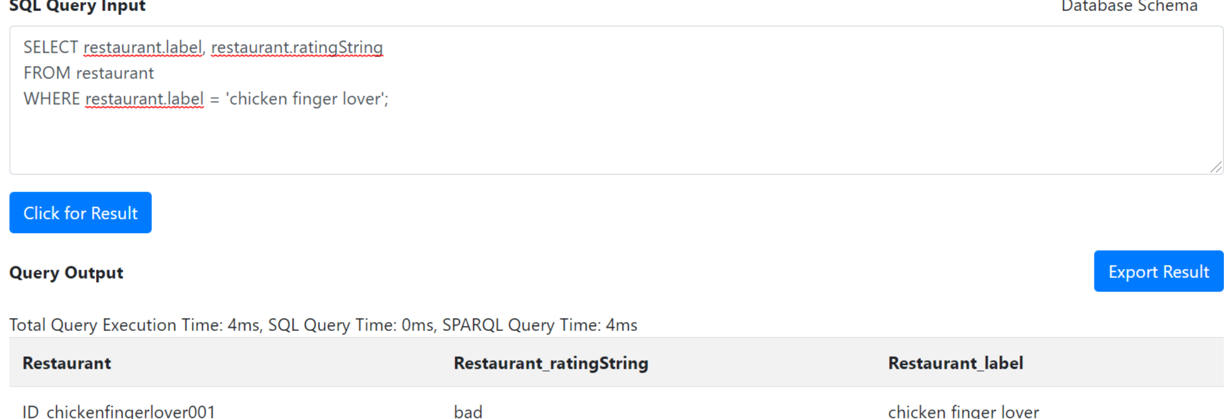


Figure 22. SQLogy Query Result for A Newly Added Restaurant Instance

Chapter 4 - Experiments & Evaluation

4.1 Experiment Setup

Experiments of this thesis project focus on comparing the querying execution time between SQLogy method and baseline method. Here are the details of experiments.

4.1.1 Data Source

Two different datasets are used for experiments as well as the evaluation in this thesis research. Tabular dataset about customers and orders were generated by a database population program DBGEN for the use of TPC-H benchmark and stored inside PostgreSQL database. There are two tables residing in Postgres database: *Customer* and *Orders.* The *Customer* table includes the following columns (attributes):c_custkey (the primary key), c_name, c_address, c_nationkey, c_phone, c_acctbal, c_mktsegment, and c_comment. The *Customer* table consists of a total of 150,000 customer records. The *Orders* table includes a total of ten columns (attributes): o_orderkey (the primary key), o_orderstatus, o_totalprice, o_orderdate, o_orderpriority, o_clerk, o_shippriority, o_comment, o_custkey (which is a foreign key from the table Customer), and o_rest_id (which is a key attribute used for connection with restaurant ontology data). This table consists of 1,048,575 records in total. The data of one of the foreign keys of the *Order* table, o_rest_id, comes from the restaurant ontology dataset. All unique restaurant identifiers (URIs) are initially being processed (Only the end “ID_xxxxx” part of the original URIs is used.) and randomly populated to the column “o_rest_id” of the table *Orders*. This specific column is the bridge to connect both Postgres database and restaurant ontology dataset.

Information or knowledge about restaurants, originally provided by Mooney.net, is stored as an owl ontology file on Apache Jena Fuseki server. The restaurant ontology includes about 9,750 individual restaurant instances and provides access to a total of five kinds of information about restaurants’ name (“label”), food type (“foodType”), location (“location”), numeric rating from customers (“rating”), and string rating (“ratingString”).

4.1.2 Preliminary Jobs

The preliminary jobs handled before the formal experiment include acquiring dataset from the original data sources, preprocessing the original data, uploading the data to the right database systems, building the connection between Postgres database and Apache Jena Fuseki ontology dataset.

4.2 Performance Results & Evaluation

4.2.1 Baseline Approach

In order to evaluate the performance of SQLogy, we need to first implement a baseline approach and then compare the result of SQLogy against that of baseline approach. In this thesis research, we decide the baseline approach to be querying information of all customers, orders, and restaurants directly from Postgres database. The goal of SQLogy is to query data from both RDBMS and an ontology without any data transformation from one data source to the other. However, we are curious to see the comparison between querying data from a single RDBMS source and querying information from both data sources. Given the nature of the original restaurant ontology dataset, we must transform all restaurant instances from the restaurant ontology owl file into a table inside Postgres. First, we used a SPARQL query to retrieve all restaurant instances with their restaurant identifiers (URIs) and all five attributes (label, foodType, location, rating, and ratingString) and saved them all in a csv file. Second, we transformed all restaurant identifiers into restaurant ids like how we handle them for SQLogy in former chapters. Third, we imported the tabular dataset of the restaurant into Postgres database as the restaurant table and made "r_rest_id" (restaurant id) the primary key as well. There are two foreign keys inside the orders table: "o_cuskey" refers to the primary key "c_custkey" of the referenced table customer; "o_rest_id" points to the primary key "r_rest_id" of the referenced restaurant table. The final step is to run all matching testing SQL queries to log the query execution time (in milliseconds (ms)) and record the result, which will be used in comparison against SQLogy's performance.

4.2.2 Measurement

The measurement we use in this thesis to evaluate the results of both SQLogy and baseline method is the query execution time in milliseconds (ms). For the baseline method, we take advantage of

the command "EXPLAIN" in PostgreSQL which displays the execution plan that the PostgreSQL planner creates for the supplied statement (PostgreSQL.org, 2021). Both the query planning time and the execution time will be presented. As for SQLogy, we implement the log of querying planning and executing time inside the code.

4.2.3 Sample Query Execution Results (SQLogy & Baseline)

This section illustrates several sample query execution results for all five major use cases, which we discussed in detail in Chapter 3, and statistics of query execution time.

Use Case 1

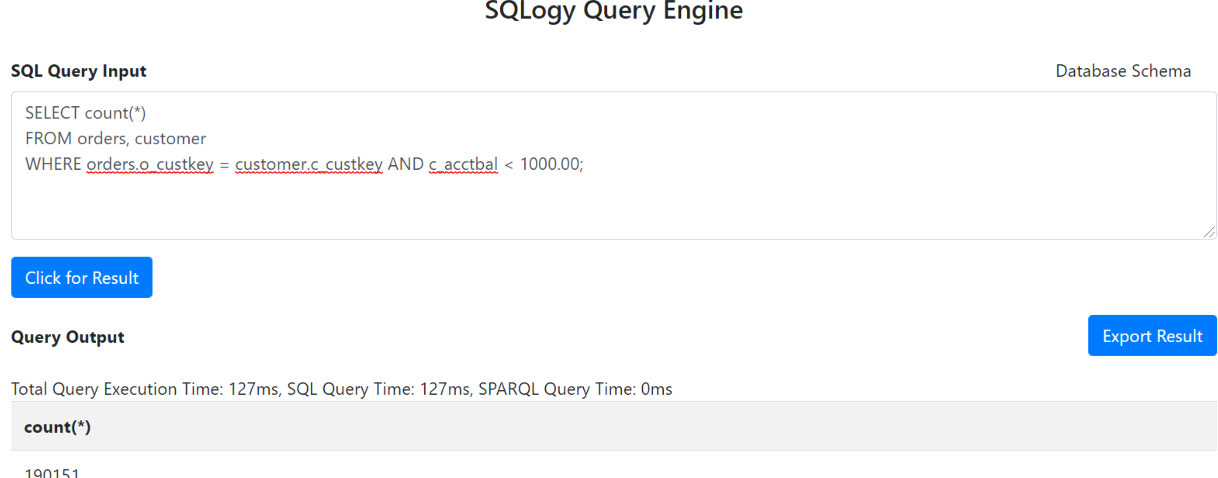


Figure 23. A Sample Use Case 1 SQL Query Input & the Result in SQLogy

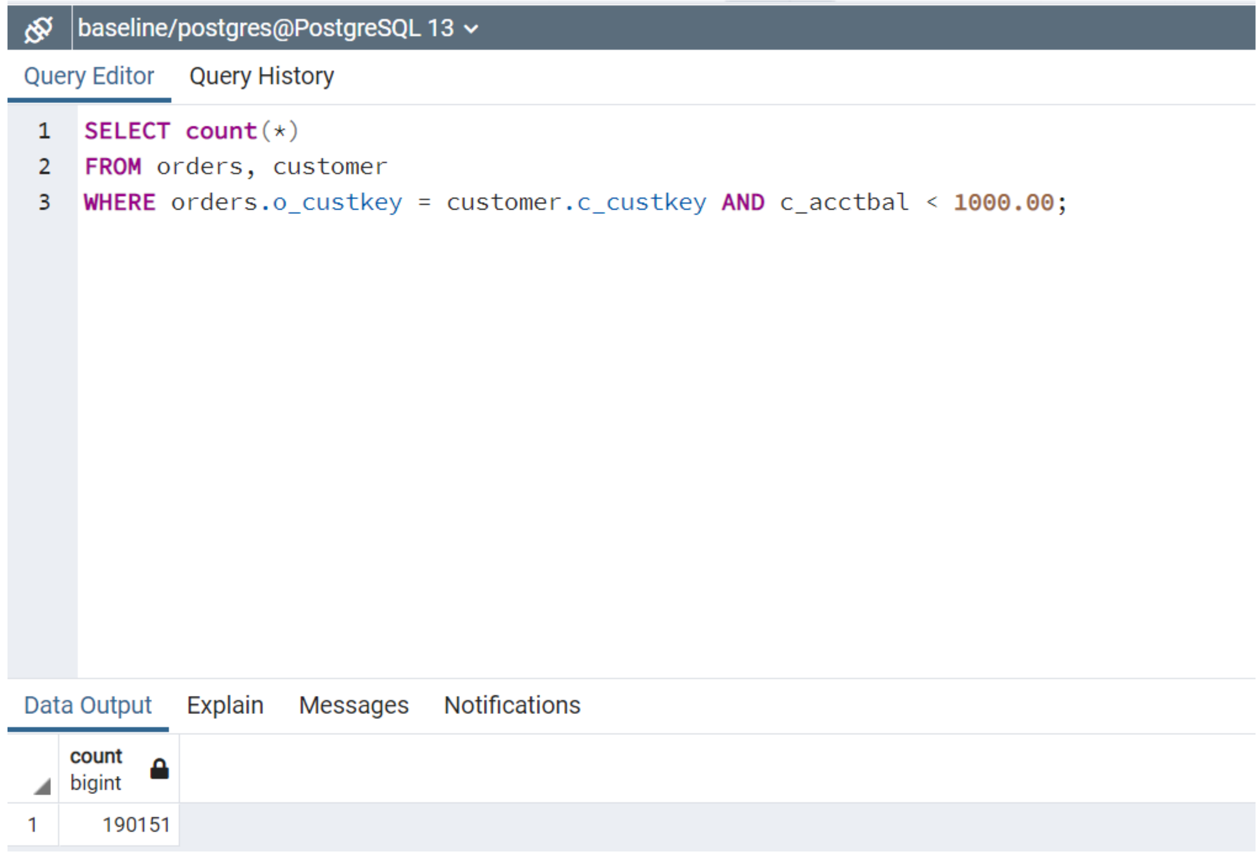


Figure 24. A Sample Use Case 1 SQL Query & the Result in Postgres

Planning Time: 0.135 ms

Execution Time: 122.954 ms

Figure 25. Query Execution Time of a Sample Use Case 1 SQL Query

Use Case 2

SQLogy Query Engine

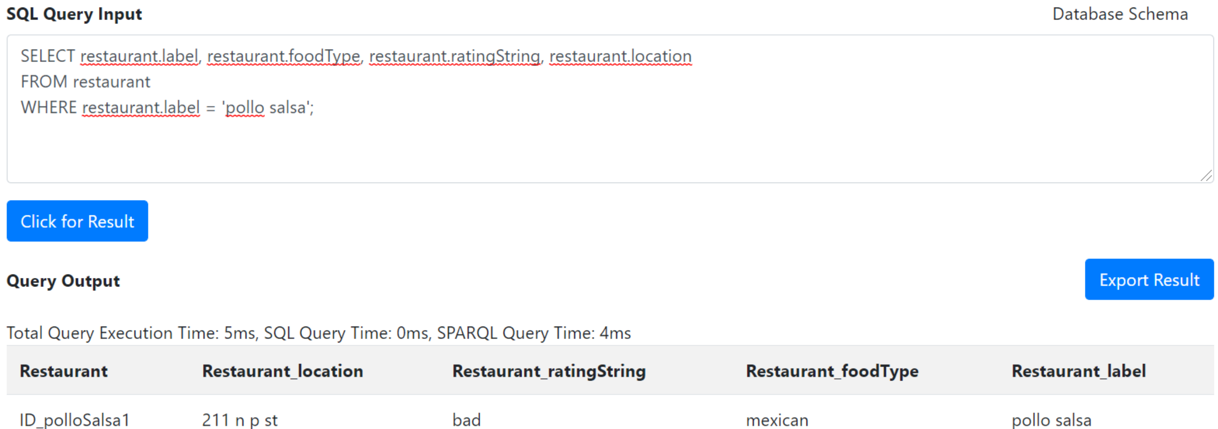


Figure 26. A Sample Use Case 2 SQL Query Input & the Result in SQLogy

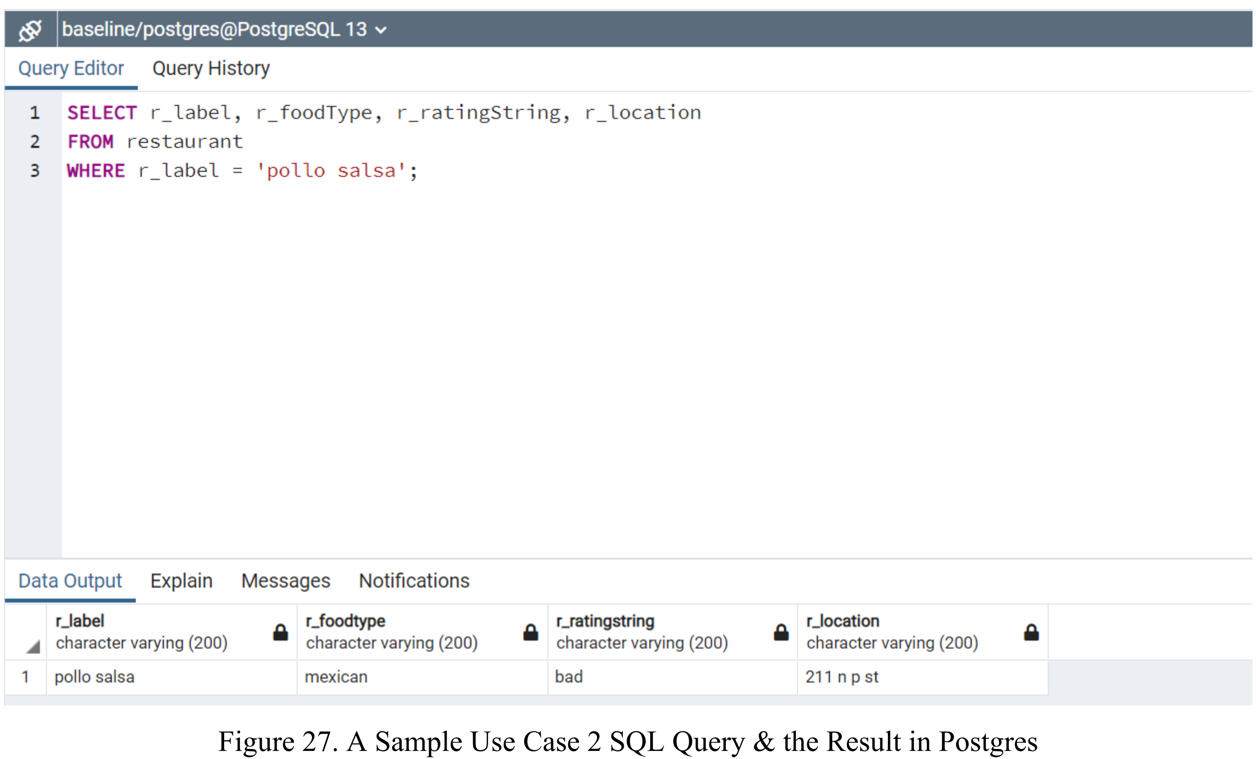


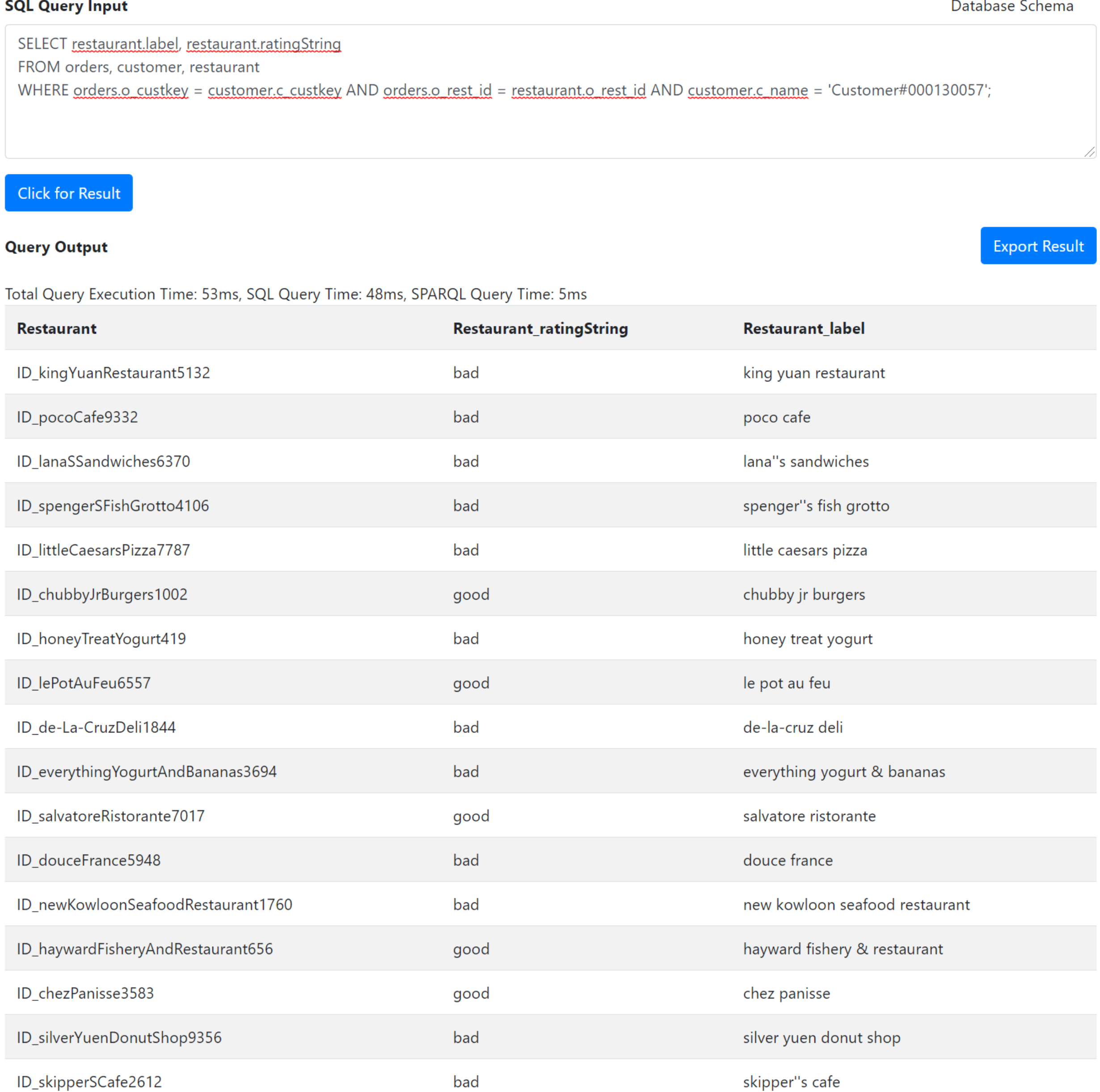
Figure 27. A Sample Use Case 2 SQL Query & the Result in Postgres

Planning Time: 0.046 ms

Execution Time: 0.624 ms

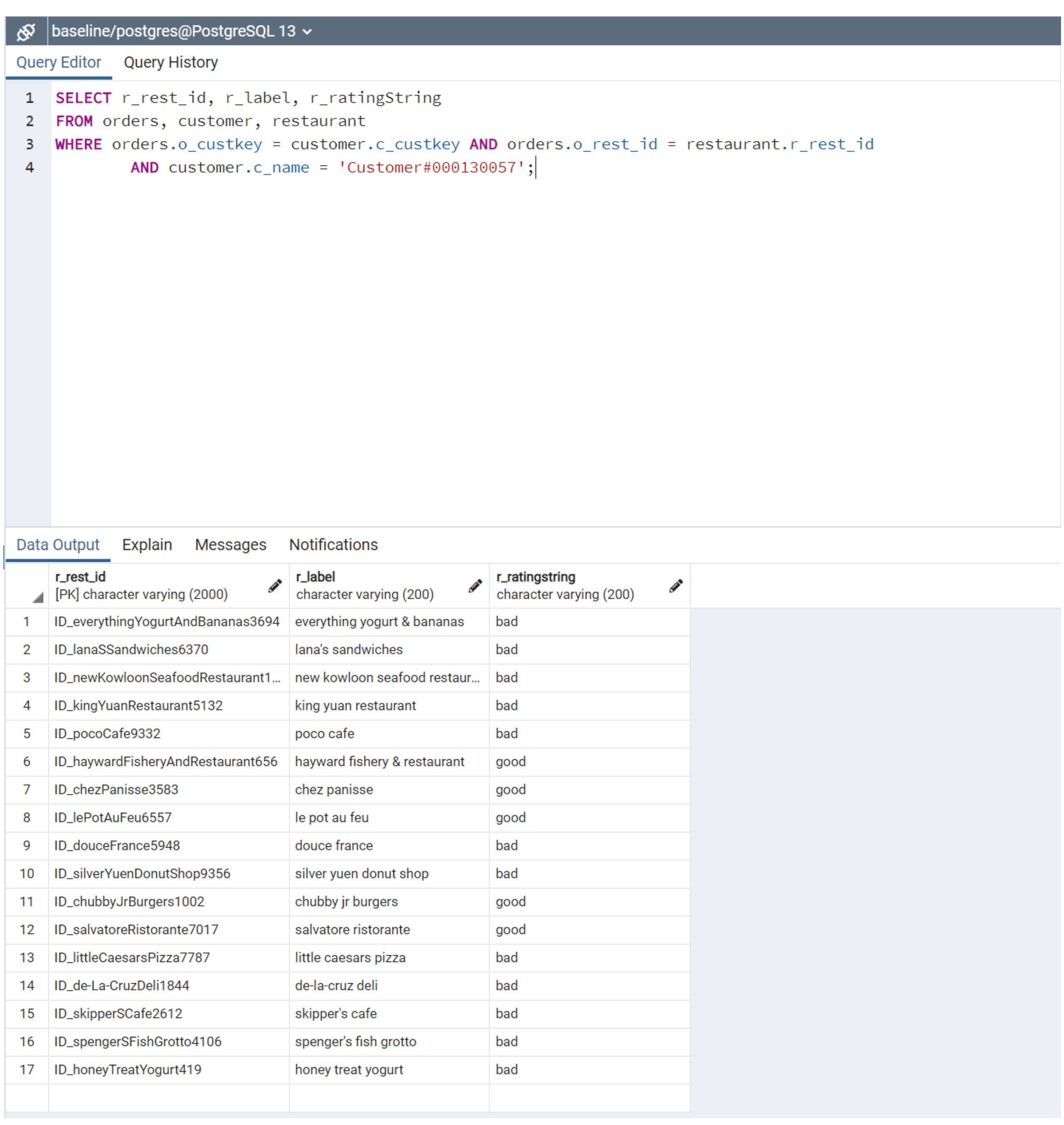
Figure 28. Query Execution Time of a Sample Use Case 2 SQL Query

Use Case 3



Restaurant	Restaurant_ratingString	Restaurant_label
ID_kingYuanRestaurant5132	bad	king yuan restaurant
ID_pocoCafe9332	bad	poco cafe
ID_lanaSSandwiches6370	bad	lana''s sandwiches
ID_spengerSFishGrotto4106	bad	spenger''s fish grotto
ID_littleCaesarsPizza7787	bad	little caesars pizza
ID_chubbyJrBurgers1002	good	chubby jr burgers
ID_honeyTreatYogurt419	bad	honey treat yogurt
ID_lePotAuFeu6557	good	le pot au feu
ID_de-La-CruzDeli1844	bad	de-la-cruz deli
ID_everythingYogurtAndBananas3694	bad	everything yogurt & bananas
ID_salvatoreRistorante7017	good	salvatore ristorante
ID_douceFrance5948	bad	douce france
ID_newKowloonSeafoodRestaurant1760	bad	new kowloon seafood restaurant
ID_haywardFisheryAndRestaurant656	good	hayward fishery & restaurant
ID_chezPanisse3583	good	chez panisse
ID_silverYuenDonutShop9356	bad	silver yuen donut shop
ID_skipperSCafe2612	bad	skipper''s cafe

Figure 29. A Sample Use Case 3 SQL Query Input & the Result in SQLogy



baseline/postgres@PostgreSQL 13

Query Editor | Query History

```sql
SELECT r_rest_id, r_label, r_ratingString
FROM orders, customer, restaurant
WHERE orders.o_custkey = customer.c_custkey AND orders.o_rest_id = restaurant.r_rest_id
        AND customer.c_name = 'Customer#000130057';
```

Data Output | Explain | Messages | Notifications

	r_rest_id [PK] character varying (2000)	r_label character varying (200)	r_ratingstring character varying (200)
1	ID_everythingYogurtAndBananas3694	everything yogurt & bananas	bad
2	ID_lanaSSandwiches6370	lana's sandwiches	bad
3	ID_newKowloonSeafoodRestaurant1...	new kowloon seafood restaur...	bad
4	ID_kingYuanRestaurant5132	king yuan restaurant	bad
5	ID_pocoCafe9332	poco cafe	bad
6	ID_haywardFisheryAndRestaurant656	hayward fishery & restaurant	good
7	ID_chezPanisse3583	chez panisse	good
8	ID_lePotAuFeu6557	le pot au feu	good
9	ID_douceFrance5948	douce france	bad
10	ID_silverYuenDonutShop9356	silver yuen donut shop	bad
11	ID_chubbyJrBurgers1002	chubby jr burgers	good
12	ID_salvatoreRistorante7017	salvatore ristorante	good
13	ID_littleCaesarsPizza7787	little caesars pizza	bad
14	ID_de-La-CruzDeli1844	de-la-cruz deli	bad
15	ID_skipperSCafe2612	skipper's cafe	bad
16	ID_spengerSFishGrotto4106	spenger's fish grotto	bad
17	ID_honeyTreatYogurt419	honey treat yogurt	bad

Figure 30. A Sample Use Case 3 SQL Query & the Result in Postgres

Planning Time: 0.249 ms

Execution Time: 117.304 ms

Figure 31. Query Execution Time of a Sample Use Case 3 SQL Query

Use Case 4

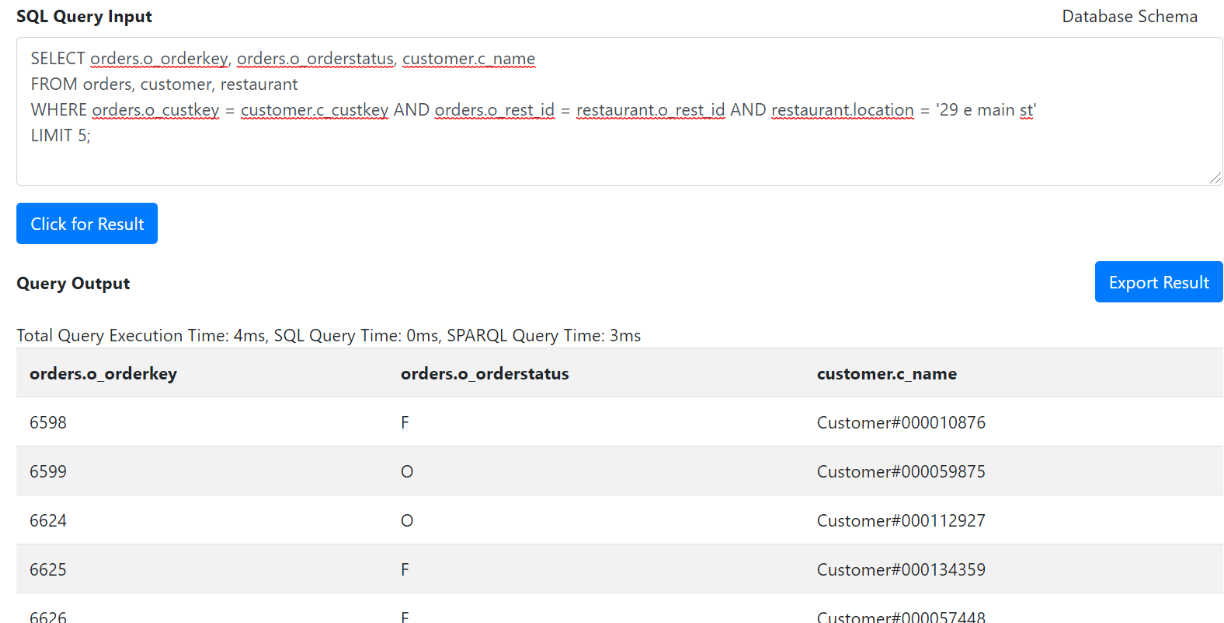


Figure 32. A Sample Use Case 4 SQL Query Input & the Result in SQLogy

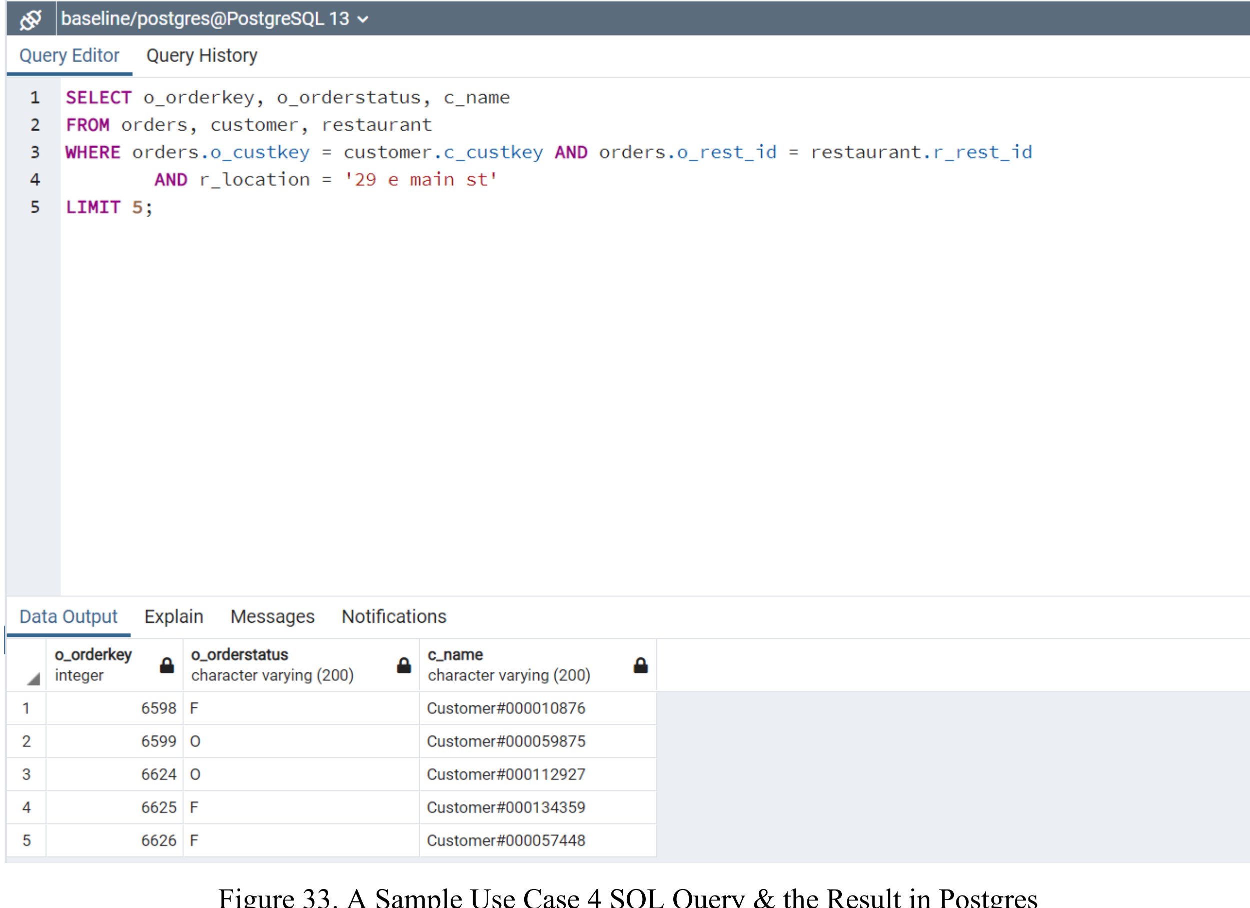


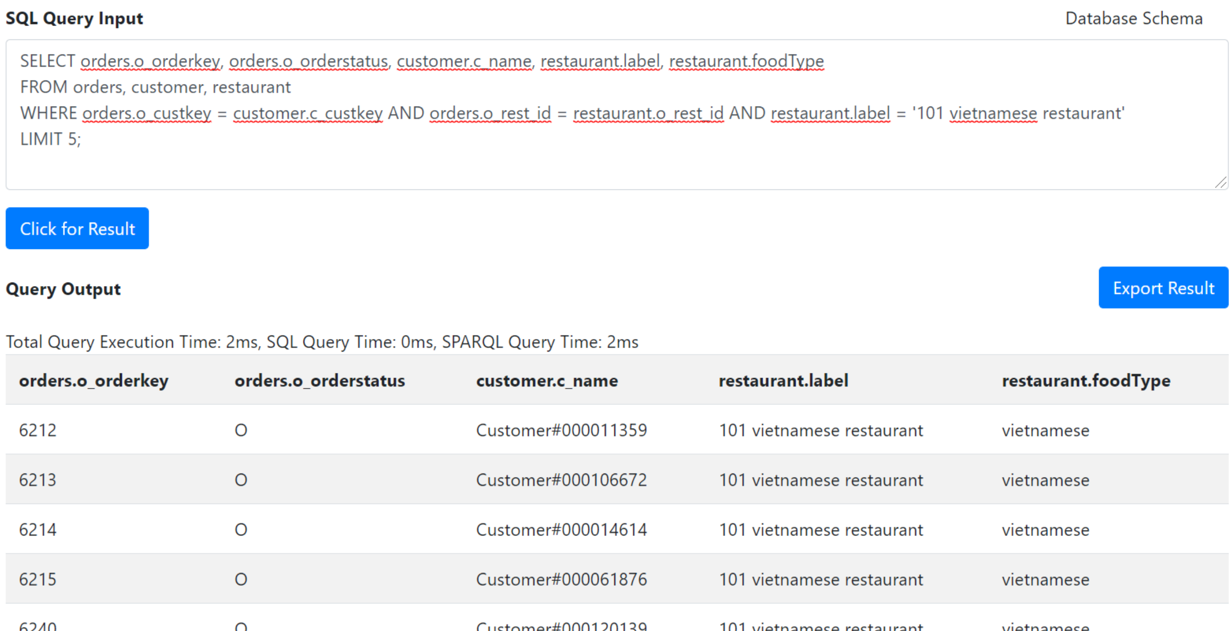
Figure 33. A Sample Use Case 4 SQL Query & the Result in Postgres

Planning Time: 0.252 ms

Execution Time: 0.572 ms

Figure 34. Query Execution Time of a Sample Use Case 4 SQL Query

Use Case 5



orders.o_orderkey	orders.o_orderstatus	customer.c_name	restaurant.label	restaurant.foodType
6212	O	Customer#000011359	101 vietnamese restaurant	vietnamese
6213	O	Customer#000106672	101 vietnamese restaurant	vietnamese
6214	O	Customer#000014614	101 vietnamese restaurant	vietnamese
6215	O	Customer#000061876	101 vietnamese restaurant	vietnamese
6240	O	Customer#000120139	101 vietnamese restaurant	vietnamese

Figure 35. A Sample Use Case 5 SQL Query Input & the Result in SQLogy

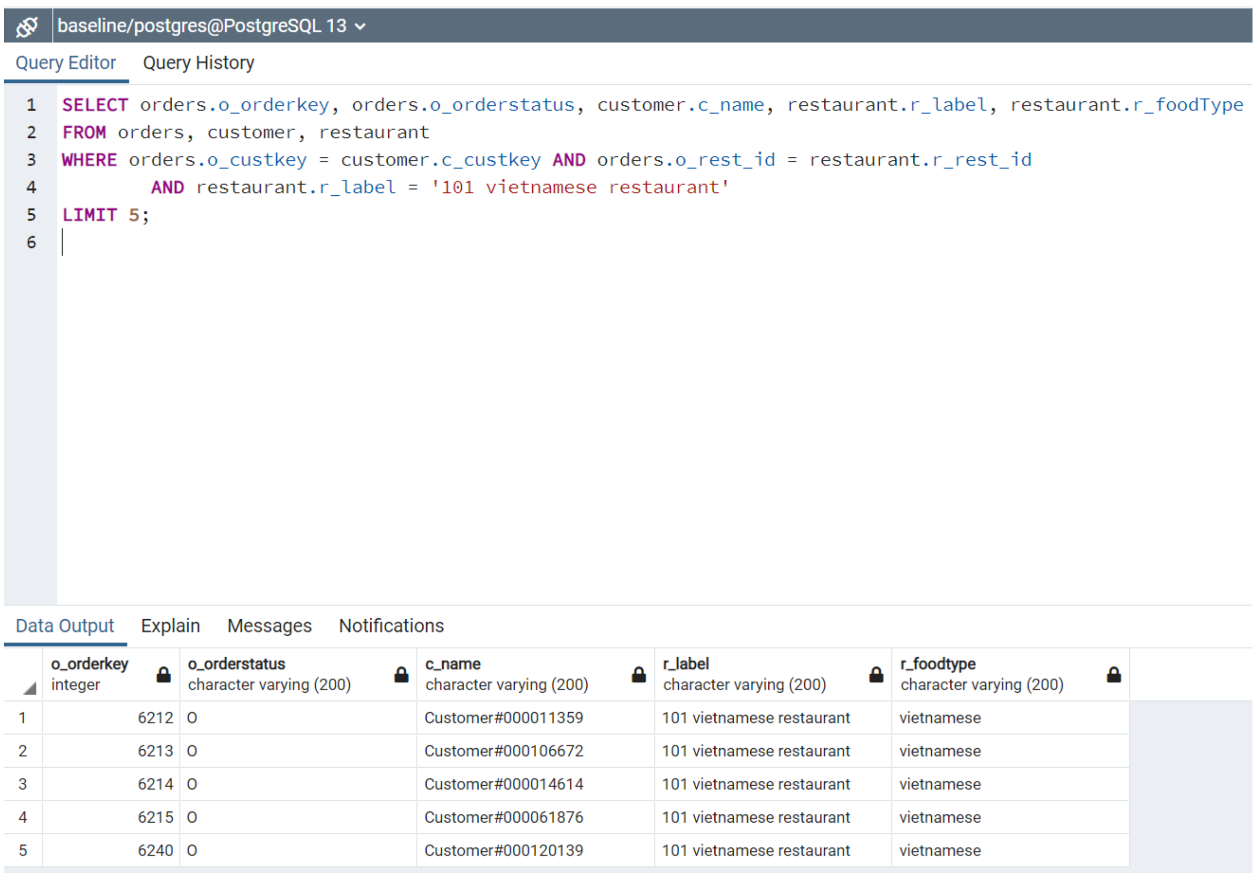


Figure 36. A Sample Use Case 5 SQL Query & the Result in Postgres

Planning Time: 0.340 ms

Execution Time: 0.602 ms

Figure 37. Query Execution Time of a Sample Use Case 5 SQL Query

4.2.4 Query Execution Time Results

	Baseline (*ms)	SQLogy (SQL+SPARQL) (*ms)
Use Case 1	123	127 (127 + 0)
Use Case 2	1	5 (0 + 5)
Use Case 3	118	53 (48 + 5)
Use Case 4	1	4 (1 + 3)
Use Case 5	1	2 (0 + 2)

Table 1. Query Execution Time Result (Baseline vs. SQLogy)

Observation

When examining the result of example query for Use Case 1, we see that results from both methods are close to each. In Use Case 1, SQLogy takes the original SQL query input and submits it directly to the PostgreSQL database to query data. Thus, both SQLogy and the baseline approach use the same SQL query. As for Use Case 2, 4, and 5, we could clearly tell that SQLogy takes longer time to retrieve the information. In Use Case 2, we query the same content but stored in different data systems. It seems that the querying process is more optimized in Postgres, and it consumes a short time to execute the query. Use Cases 4 and 5 in SQLogy follow the same order of logic and process. The query process consists of two parts: first querying the restaurant ontology on Apache Jena Fuseki for qualified restaurant ids (and needed columns for the final result (Use Case 5)) and then querying the Postgres with a sub-SQL integrated of all qualified restaurant ids. When compared to the querying process behind SQLogy, the baseline method queries all tables directly in a single data system, which saves much time on saving partial results in a temporary variable and then combining results from both data systems. What is beyond our expectation is that the execution time in Use Case 3 is noticeably shorter than that in baseline method. Basically, it takes twice as much time to get the final result in the baseline method. After analyzing the query execution details for Use Case 3 in SQLogy method, we could identify that most of the time is dedicated to querying the Postgres database and it only takes about 5ms to query the restaurant ontology on Jena Fuseki server. Both methods must retrieve all qualified restaurant ids and then use those restaurant ids to query restaurant data. We assume that when provided with instance identifiers (subjects of

restaurant instances), querying an ontology is quicker than querying a table inside a Postgres database.

Chapter 5 - Related Work

Integration of semantic web or knowledge data has become an attractive research topic. An ontology is designed for a purpose of enabling "the modeling of knowledge about some specific domain, real or imagined" (Gruber, n.d.). It has become one of the key areas of search in the study filed of semantic web. Ontologies are already included in the W3C standards stack for the Semantic Web. The significant role that ontologies play in the field of database systems is to "specify a data modeling representation at a level of abstraction above both specific logical and physical database designs, so that data can be exported, translated, queried, and unified across independently developed systems and services" (Gruber, n.d.).

In order to work with different data resources and facilitate interoperability across heterogeneous data storage systems, integration of multiple, heterogeneous data systems has become undoubtedly necessary. One of the popular research areas is to unify the data systems by mapping one data storage system to another, which makes it accessible to data search, manipulation, etc. Mapping between relational databases and semantic web ontology is one of the most popular research areas since a large amount of data is normally stored in relational databases, but semantic web cannot directly make use of the data stored in RDB. Hazber, et al. (2016) propose a new approach of transforming relational databases to semantic web ontology by first building ontology from RDB schema and then populating the ontology with instances from RDB following mapping rules. Hazber et al. (2016) believe that this proposed method could make it available for "semantic web applications to access relational databases and their contents by semantic methods".

Ghanwi and Cullot (2007) also propose a general interoperability architecture that makes use of ontologies for explicit description of semantics of information sources, and web services to facilitate the communication among the different components of the architecture. The architecture provides services for mapping information sources to local ontologies and several web services for encapsulating the different functionalities of the architecture (Ghawi & Cullot, 2007).

Instead of first unifying data systems, another prosperous research area: a federated query engine is gaining more and more attention within the research community. A federated query engine is used for querying different data resources across different dimensions of heterogeneity. Bradshaw,

et al. propose a Utah statewide informatics platform “The Federated Utah Research and Translational Health e-Repository (FURHeR)” that includes a key element of a federated query engine for “heterogeneous resources to access and integrate diverse biomedical data and promote discovery of new knowledge” (Bradshaw et al., 2009). The federated query engine is responsible for creating data queries and distributing them to corresponding data resources and then aggregating the query results. Our proposed federated query engine web app also adopts the idea of querying heterogeneous dataset with a single end user input.

Chapter 6 - Conclusion and Future Work

6.1 Conclusion

In this thesis project, We designed a new approach of querying both RDBMS and an ontology dataset in the same domain of knowledge without any data mapping or transformation in between two distinct data sources. We also proposed an integrated query engine with a simple and clean user interface for end-users to query both RDBMS and ontology dataset with only a standard SQL. No SPARQL query is required for querying the ontology dataset. The engine managed the integrated processing transparently from the end-users. We explained how SQLogy works in all major six use cases and provided abundant working examples to illustrate the internal querying process. When compared to other methods that focus on the data mapping or transformation in between RDBMS and ontology, our proposed SQLogy not only avoids several drawbacks associated with the mapping method but demonstrates various peculiar advantages. In the very end, the performance statistics of both SQLogy and baseline methods from experiments were evaluated and reported as well.

6.2 Future Work

The integrated query engine SQLogy has much space of improvement. Our future work mainly focuses on the following parts.

6.2.1 Improve Compatibility of SQLogy on Input SQL Query

Given the discussion we have in previous chapters, the current version of SQLogy provides access to some basic kinds of standard SQL query input. We expect to improve the compatibility of SQLogy. One of the improvements we would like to work on is the SQLogy's compatibility for more advanced or complex SQL query inputs. For example, the current version of SQLogy does not support input SQL queries with aggregations or any nested queries. Another improvement we would like to see in the future version of SQLogy is that the query engine could be compatible in more use case scenarios. The current version of SQLogy is working in almost all major use case scenarios except one where SQLogy needs to query columns from both data sources with a conditional filter applying to columns of orders or customers.

6.2.2 Extend Adaptability of SQLogy on Data Sources & Systems

SQLogy now works smoothly with the restaurant owl ontology file stored in Apache Fuseki server as well as PostgreSQL database. We plan to extend the adaptability by making SQLogy support uploading new datasets. More RDBMS products and more types of knowledge graph or semantic web data, e.g., RDFs are also expected to be included. Currently, SQLogy only works with local ontology files. However, we expect SQLogy to be capable of querying online ontology data or knowledge graphs.

6.2.3 Optimize Internal Query Execution Efficiency & Effectiveness

Given our discussion in Chapter 4 about the comparison of performance of both SQLogy and the baseline methods, we want to work on the optimization of SQLogy's internal querying process SQLogy. When comparing to the general transformation or mapping methods, we feel obliged to optimize SQLogy's internal functionality and potentially speed up the querying execution process.

6.2.4 Integrate SQLogy into Other Applications

SQLogy is working as an individual query engine with a simple web user interface. We plan to extend the functionality of SQLogy and integrate it into other applications. For example, SQLogy could become an integrated part of a web application with a searching function where end users do not necessarily have to provide standard SQL queries to get the needed information. They could simply provide keywords along with other filters to initiate the web app to construct standard SQL queries in the backend.

Chapter 7 - Reference

Apache Jena. (2011). *Apache Jena Fuseki Documentation.* From Apache Jena: https://jena.apache.org/documentation/fuseki2/

Bizer, C. (2003). D2R MAP – A Database to RDF Mapping Language. *The twelfth international World Wide Web, WWW2003.* Budapest.

Bradshaw, R. L., Matney,, S., Livne, O. E., Bray, B. E., Mitchell,, J. A., & Narus,, S. P. (2009). Architecture of a Federated Query Engine for Heterogeneous Resources. *AMIA Annu Symp Proc, v2009; 2009*, 70–74. From https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2815441/

DuCharme, B. (2013). *Learning SPARQL* (2nd Edition ed.). O'Reilly.

Gasevic, D., & Djuric, D. (2006). *Model Driven Architecture and Ontology Development.* Springer.

Ghawi, R., & Cullot, N. (2007). Database-to-ontology mapping generation for semantic interoperability. *Conference: Third International Workshop on Database Interoperability (InterDB 2007), held in conjunction with VLDB 2007At: Vienna, Austria*.

Gruber, T. (n.d.). *Ontology.* From tomgruber.org: https://tomgruber.org/writing/ontology-definition-2007.htm

Hazber, M., Li, R., Gu, X., & Xu, G. (2016). Integration Mapping Rules: Transforming Relational Database to Semantic Web Ontology. *Applied Mathematics & Information Sciences, 10*(3), 881-901. From

https://www.researchgate.net/publication/301886521_Integration_Mapping_Rules_Transforming_Relational_Database_to_Semantic_Web_Ontology

Herman, I., Fernández, S., Alonso, C., & Zakhlestin, A. (2016, 12 5). *SPARQLWrapper: SPARQL Endpoint interface to Python.* From Github: https://github.com/RDFLib/sparqlwrapper

Jacob, E. K. (2003). Ontologies and the Semantic Web. *Bulletin of the American Society for Information Science and Technology*(April/May 2003), 19, 20, 21, 22. From https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/bult.283

Konstantinou, N., Spanos, D.-E., & Mitrou, N. (n.d.). Ontology and Database Mapping: A Survey of Current Implementations and Future Directions. *Journal of Web Engineering (JWE), 7(1)*(1-24). From https://www.researchgate.net/publication/220538166_Ontology_and_Database_Mapping_A_Survey_of_Current_Implementations_and_Future_Directions

Martinez-Cruz, C., Blanco, I. J., & Vila, M. A. (2012). Ontologies versus relational databases: are they so different? A comparison. *Artificial Intelligence Review, 38*, 271-290. From https://link-springer-com.ezpxy-web-p-u01.wpi.edu/article/10.1007/s10462-011-9251-9

Ontotext. (2015). *What is SPARQL?* From Ontotext: https://www.ontotext.com/knowledgehub/fundamentals/what-is-sparql/

PostgreSQL.org. (2021). *PostgreSQL 9.1.24 Documentation.* PostgreSQL.org. From https://www.postgresql.org/docs/9.1/sql-explain.html

Simons, P. M. (2015). Ontology. *Encyclopedia Britannica*.

Stackoverflow. (2018, 2 22). *What is the difference between RDF and OWL?* From stackoverflow: https://stackoverflow.com/questions/1740341/what-is-the-difference-between-rdf-and-owl

Sugumaran, V. (2016). *Chapter 14 - Semantic technologies for enhancing knowledge management systems.* Morgan Kaufmann.

Surani, I. (2020, 2 17). *Challenges of Integrating Heterogeneous Data Sources.* From DATAVERSITY: https://www.dataversity.net/challenges-of-integrating-heterogeneous-data-sources/#:~:text=Challenge%3A%20Data%20heterogeneity%20leads%20to,exponential%20growth%20in%20data%20volume.&text=The%20data%20within%20each%20system,customer%20information%2C%20and%2

The OWL API. (2016). From owlapi: http://owlapi.sourceforge.net/

W3 Schools. (2015). *XML RDF.* From W3 Schools: https://www.w3schools.com/xml/xml_rdf.asp#:~:text=RDF%20stands%20for%20Resource%20Description,RDF%20is%20written%20in%20XML

W3C. (2008, 1 15). *SPARQL Query Language for RDF.* From w3.org: https://www.w3.org/TR/rdf-sparql-query/

W3C.org. (2013, 12 11). *OWL.* From W3C.org: https://www.w3.org/OWL/#:~:text=The%20W3C%20Web%20Ontology%20Language,things%2C%20and%20relations%20between%20things.

W3C.org. (2014, 3 15). *RDF.* From W3C.org: https://www.w3.org/RDF/

W3C.org. (2015). *Semantic Web.* From w3.org: https://www.w3.org/standards/semanticweb/

Zhang, J. (2007). Ontology and the Semantic Web. *Proceedings of the North American Symposium on Knowledge Organization, 1*. From file:///C:/Users/sslee/Downloads/12830-13653-1-PB.pdf