PLUG-AND-PLAY WEB SERVICES

by

Arihant Jain

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Curtis Dyreson, Ph.D. Major Professor	Vladimir Kulyukin, Ph.D. Committee Member
Nicholas Flann, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2019

ABSTRACT

Plug-and-Play Web Services

by

Arihant Jain, Master of Science

Utah State University, 2019

Major Professor: Curtis Dyreson, Ph.D.
Department: Computer Science

The goal of this research is to make it easier to design and create web services for relational databases. A web service is a software service for providing data over HTTP. Web services provide data endpoints for many web applications. We adopt a plug-and-play approach for web service creation whereby a designer constructs a "plug," which is a simple, declarative specification of the hierarchical output produced by the service. If the plug can be "played" on the database then the web service is generated. The designer creates the hierarchy in the plug by nesting attributes of relations (an *attribute* is the name of a column in a database table). Our plug-and-play approach has three advantages. First, a plug is *portable*. You can take the plug to any data source and generate a web service. Second, a plug-and-play service is more *reliable*. The web service generation checks the database to determine if the service can be safely and correctly generated. Third, plug-and-play web services are *easier to code* for complex data since a service designer can write a simple plug, abstracting away the data's real complexity. Building a web service for a plug however is a complex process. First, the relationships among the attributes in a pattern must be generated. To determine how data is related, we build a *foreign key graph*. Each node in the graph is a relation (which includes all of the attributes in the relation) and each (undirected) edge is a foreign key. Each attribute in a plug maps to a node in the graph. We determine data-relatedness by using *closeness* for each parent/child association in a plug. Closeness specifies that data is related by the shortest path(s) between the parent and child in the foreign key graph. The output specified by the plug is constructed by joining the relations along the shortest path using the foreign keys. Our Java implementation extracts the database metadata to create the foreign key graph. We find the shortest paths between pairs of nodes and transpile each path to an SQL query. If there are several paths, the paths can be ranked by estimating or evaluating the number of tuples produced since more tuples is deemed to be more complete, with respect to all possible pairings of attribute values in a plug. A designer can then choose which path, or set of paths, they deem best. After making a selection, the web service is generated. When invoked, the service will execute the generated SQL query to produce data, which is then formatted according to the hierarchy specified by the plug.

(54 pages)

PUBLIC ABSTRACT

Plug-and-Play Web Services

Arihant Jain

The goal of this research is to make it easier to design and create web services for relational databases. A web service is a software service for providing data over computer networks. Web services provide data endpoints for many web applications. We adopt a plug-and-play approach for web service creation whereby a designer constructs a “plug,” which is a simple specification of the output produced by the service. If the plug can be “played” on the database then the web service is generated. Our plug-and-play approach has three advantages. First, a plug is *portable.* You can take the plug to any data source and generate a web service. Second, a plug-and-play service is more *reliable.* The web service generation checks the database to determine if the service can be safely and correctly generated. Third, plug-and-play web services are *easier to code* for complex data since a service designer can write a simple plug, abstracting away the data’s real complexity. We describe how to build a system for plug-and-play web services, and experimentally evaluate the system. The software produced by this research will make life easier for web service designers.

To my parents and Jagrati, for their everlasting love and support

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to Dr. Curtis Dyreson for his immense help and patience. I am extremely thankful to him for the detailed explanations and answers to all my questions, even the most trivial ones. This thesis truly would have not been possible without his unswerving support and advice.

I would also like to thank Dr. Vladimir Kulyukin and Dr. Nicholas Flann for their continuous help and for being a part of my graduate committee. I also want to extend my thanks to Dr. Haitao Wang for his insights and suggestions with the research.

Arihant Jain

CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Modern web applications often use a three-tier architecture consisting of clients (web browsers), a web server, and a database management system (DBMS) as shown in Figure 1.1. The web server is the intermediary between a client and the DBMS, a client communicates with the web server and the web server interacts with the DBMS. Data for the application, including the pages in the application, is managed by the DBMS.

A web service is used to move data between the client and the DBMS. GET services read data while POST services post application data to the DBMS. The transported data is typically formatted as JSON, which is a simple, text-based representation of an object. A web service runs on a web server and in addition to communicating with a client also interacts with the DBMS. Data is read from a DBMS, usually by executing a query, and a service can also call on the DBMS to update data.

There exist many tools and technologies to assist in the design and construction of web services. These tools fall into two broad categories. First there are tools that aid in construction of the *API-side of a web service.* An API is a a web service application program interface. An API-side tool focuses on creating the client's view of a web service, that is, a program interface and documentation. A user provides or can construct the API specification using a tool's user interface, *e.g.,* the Swagger User Interface editor [1]. The resulting API specification, usually a Yet Another Markup Language (YAML) file, describes the structure of the inputs to and outputs from the web service and both serves as documentation for the service and a means to automatically construct the code for the service, *e.g.,* as a PHP or Python script. The second category of tools creates the *DBMS-side of the web service.* A canonical tool in this category is Doctrine [2]. Doctrine is an object relational mapper (ORM) platform. A web service designer adorns a class with Doctrine annotations. The annotations describe how to map objects to relational tables.

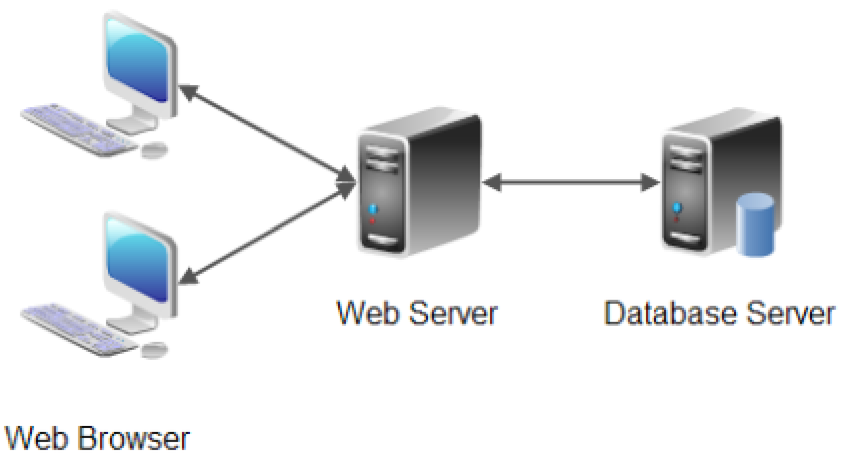


Fig. 1.1: Three-tier architecture

The annotations can be processed to construct create, read, update, delete (CRUD) web services for each annotated class.

1.1 Combining API-side and DBMS-side web service construction

We propose a combined tool, called AutoREST, to create both the API-side and DBMS-side of a web service. AutoREST adopts a *plug-and-play* approach to web service creation. A plug-and-play web service is like a plug-and-play device that can be plugged into any kind of socket and used. A plug-and-play device is *self-contained* in the sense that it has a specification that facilitates the discovery of a hardware component in a system without the need for physical device configuration or user intervention. Similarly a plug-and-play web service is self-contained. AutoREST takes as input a high-level specification of what kind of data the service provides. Given the specification, the service is constructed from the DBMS. In a plug-and-play approach when a web service is constructed for a data source, it either generates a service, a warning that that the data is insufficient for producing a reliable service, or an error that the service cannot be constructed for a data source.

As an example, suppose that a user wants data about books, grouping titles and ISBNs of books below the book authors. Additionally the author would like to translate from English to Spanish the names of key/value pairs in the data (this translation is optional).

```
[{
    "nombre" : "name",
    "libres" : [{"titulo": "title", "ISBN": "isbn"}]
}]
```

Fig. 1.2: A plug-and-play web service specification

```
[{
  "nombre" : "Jane Austen",
  "libres" : [
    {"titulo" : "Pride and Prejudice", "ISBN" : "948829294"},
    {"titulo" : "Emma", "ISBN" : "408493331"},
    ...
  ]
},
{
  "nombre": : "Edward Abbey",
  ...
}
...]
```

Fig. 1.3: Data returned by the constructed service

A user would give the specification shown in Figure 1.2 for growing a new web service to provide the data. The GET service endpoint for providing the data would (if possible to construct) provide data formatted as shown in Figure 1.3.

1.2 Desirable properties of a plug-and-play specification

A plug-and-play web service is dynamically built from a high-level specification known as the *plug*. The plug should have the following properties.

- DBMS (data model) independence - A plug should not depend upon the specifics of a particular DBMS nor even be tied to a particular data model. A hierarchy can be built from a relational, semi-structured, or graph model. In this thesis we focus only on the relational data model.

- Able to construct data in the required shape - The resulting web service should produce or accept data exactly how it is specified by the plug. The plug specifies a hierarchical shape, which may not exist in the data source, so the shape must be constructed. In this thesis we describe how to construct a hierarchy from a relational database.

- Able to determine whether the shape can be constructed - As the web service is generated, it should be possible to also identify issues such as semantic mismatches between the plug and the data source and potential information loss in construction of the shape. We do not address semantic matching in the thesis since the problem is already addressed by the data integration and Semantic web communities, but we do cover potential information loss in the data transformation from flat relational tables to a hierarchy.

1.3 Advantages

Plug-and-play web services have four primary advantages. First, they are *portable*. You can take a plug-and-play web service to any data source and evaluate it. Second, they are more *reliable*. The web service checks the data environment to determine if the service can be safely and correctly evaluated. Third, plug-and-play web services are *easier to code* for complex data since a service designer can specify the service with respect to a simple view of the data, abstracting away the data's real complexity. Fourth, plug-and-play web services promote code reuse and code portability because they can be plugged into any data socket. That is, the client specifies what kind of data they want and server constructs a web service that provides the desired data.

1.4 Technical challenges

There are three primary technical challenges to creating AutoREST. The first challenge is *semantic mismatch*. It is important to determine whether the specification and the database are semantically compatible. For instance, a name used in the specification, *e.g.,* `title`, may be absent in the schema, may have a different meaning than that intended in the plug, or may be ambiguous, *e.g.,* `title` could be ambiguous since it is a common description of an attribute. We do not address this challenge in this thesis since the data integration [3–5] and Semantic web [6] communities address semantic matching. The second challenge is *hierarchy construction.* Even if the data can be semantically matched, data in a relational database is in flat tables, but the web service constructs a hierarchy of data. Some method of constructing the hierarchy from the tables is needed. The third challenge is *data relatedness.* A plug-and-play web service specification specifies a hierarchy by relating attributes from possibly disparate relations. For instance, in the specification given in Figure 1.2 `name`s have to be related to `title`s. The data relatedness challenge is first to figure out how a `name` is related to a `title` and second to determine the *best* relationship since many relations have a `title` or `name` attribute.

1.5 Outline

This thesis is organized as follows. The next thesis gives a model of a plug-and-play web service, describing how data is connected. After that, the architecture of the implementation of AutoREST is presented. We then describe some experiments that explore two variants of AutoREST. The paper concludes with a short discussion of future work.

CHAPTER 2
AutoREST Model

In this section we describe, at an abstract level, how AutoREST addresses the hierarchy construction and data-relatedness challenges outlined in Chapter 1. The key ideas are to model data-relatedness using a graph of foreign key constraints. A spanning tree in the graph determines how to best relate names in a plug-and-play pattern. The tree can be used to construct a query to extract data in the shape specified by the pattern.

2.1 Foreign Key Graph

For our purposes a relational database, D, is a set of relations, $\{R_1, \ldots, R_n\}$, and a set of foreign key constraints, K. Each relation in D has some number of attributes, that is, the schema for relation R_i is $(A_1, \ldots, A_k)$ and each foreign key in K is of the form $R_j \rightarrow R_m$, that is, relation R_j borrows a key from R_m (the attributes in the key are not relevant to our purposes).

Definition 2.1.1 (Foreign Key Graph) *The foreign key graph, $G = (V, E)$, for D is an undirected graph where $V = \{R_1, \ldots, R_n\}$ is the set of vertices and $E = \{(R_j, R_m) \mid R_j \rightarrow R_m \in K\}$ is the set of edges.* ∎

As an example consider the relational schema depicted in Figure 2.1. The foreign key graph is given below and depicted in Figure 2.2.

- $V = \{$`products`, `orders`, `categories`, `suppliers`, `order_details`, `shippers`, `customers`, `employees`$\}$

- $E = \{($`products`, `categories`$)$, $($`products`, `suppliers`$)$, $($`products`, `order_details`$)$, $($`orders`, `order_details`$)$, $($`orders`, `shippers`$)$, $($`orders`, `customers`$)$, $($`orders`, `employees`$)$, $($`employees`, `employees`$)\}$

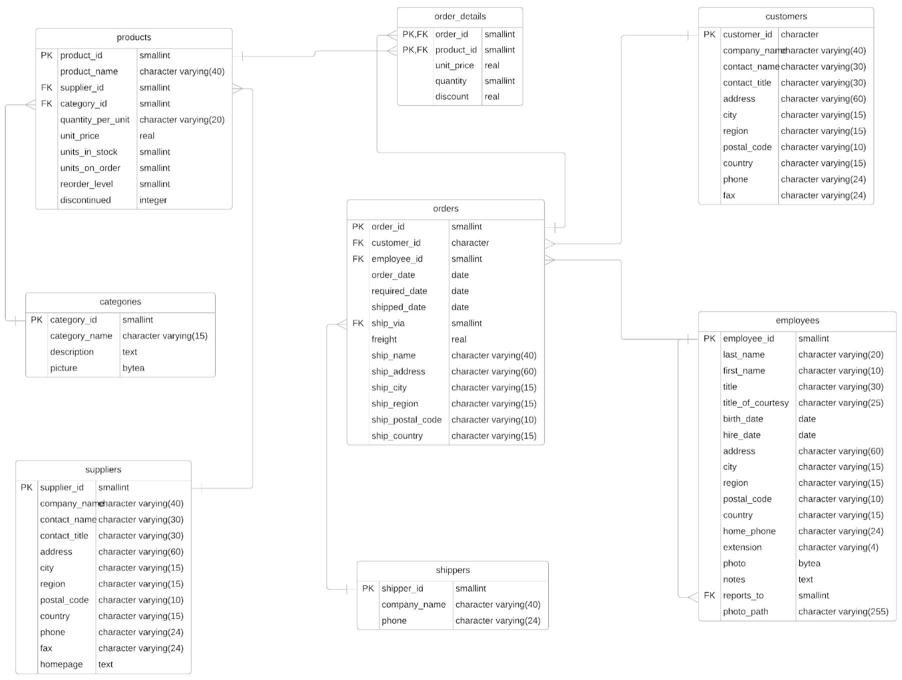

products
PK product_id smallint
product_name character varying(40)
FK supplier_id smallint
FK category_id smallint
quantity_per_unit character varying(20)
unit_price real
units_in_stock smallint
units_on_order smallint
reorder_level smallint
discontinued integer
order_details
PK,FK order_id smallint
PK,FK product_id smallint
unit_price real
quantity smallint
discount real
customers
PK customer_id character
company_name character varying(40)
contact_name character varying(30)
contact_title character varying(30)
address character varying(60)
city character varying(15)
region character varying(15)
postal_code character varying(10)
country character varying(15)
phone character varying(24)
fax character varying(24)
orders
PK order_id smallint
FK customer_id character
FK employee_id smallint
order_date date
required_date date
shipped_date date
FK ship_via smallint
freight real
ship_name character varying(40)
ship_address character varying(60)
ship_city character varying(15)
ship_region character varying(15)
ship_postal_code character varying(10)
ship_country character varying(15)
categories
PK category_id smallint
category_name character varying(15)
description text
picture bytea
employees
PK employee_id smallint
last_name character varying(20)
first_name character varying(10)
title character varying(30)
title_of_courtesy character varying(25)
birth_date date
hire_date date
address character varying(60)
city character varying(15)
region character varying(15)
postal_code character varying(10)
country character varying(15)
home_phone character varying(24)
extension character varying(4)
photo bytea
notes text
FK reports_to smallint
photo_path character varying(255)
suppliers
PK supplier_id smallint
company_name character varying(40)
contact_name character varying(30)
contact_title character varying(30)
address character varying(60)
city character varying(15)
region character varying(15)
postal_code character varying(10)
country character varying(15)
phone character varying(24)
fax character varying(24)
homepage text
shippers
PK shipper_id smallint
company_name character varying(40)
phone character varying(24)

Fig. 2.1: Schema of the Northwind database

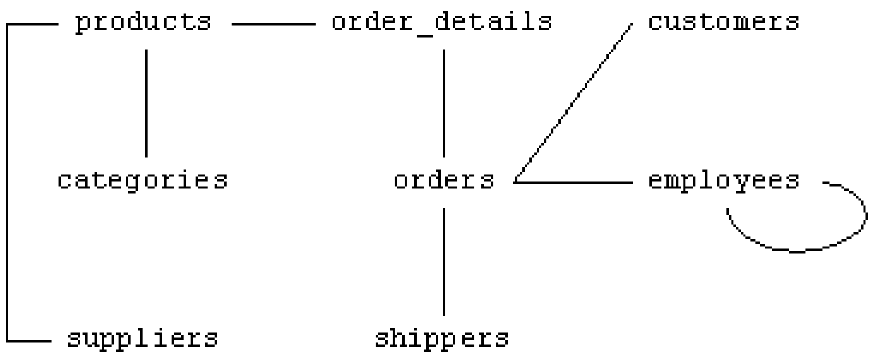


Fig. 2.2: Foreign key graph for the Northwind database

2.2 Relating Data through Names

The foreign key graph can be used to relate names in a pattern based on the notion of *closeness* [7–15]. Closeness can be described as the property that two data items are *related* if they are connected (by a path) and that no shorter paths that connect items of the same *type* exists. In the context of relational databases the *type* of a datum is the domain (an attribute in a relation) to which it belongs.

Suppose that a plug specifies that `notes` should be related to `height`. The `notes` type exists in the `employees` relation, while `height` is part of `orders`. There is a path of length one that connects `employees` to `orders` as well as paths of length greater than one (by traversing the link from `employees` to itself). Closeness stipulates that the shortest path is preferred.

Definition 2.2.1 (Parent/Child Closeness) *Let plug P have parent p with child c where p is an attribute of relation R_p and c is an attribute of relation R_c. Closeness stipulates that a path from R_p to R_c makes p closest to c if and only if there is no shorter path between R_p and R_c in a foreign key graph, F, that is,*

$$\otimes(F, P, x, y) = \{R_x, \ldots, R_y\}$$

where $\otimes$ represent the closest operator and $R_x, \ldots, R_y$ is a shortest path.

As an example assume the pattern contains names `notes` in relation `employees` and `homepage` in relation `suppliers`. Then the shortest path is as follows.

```
[(products, suppliers),
(products, order_details),
(orders, order_details),
(orders, employees)]
```

Parent/child closeness relates a pair of names in a plug, but a plug could contain many names. Closeness for the plug is built from parent/child closeness

Definition 2.2.2 (Plug Closeness) *Let P be the set of parent child relationships, (p, c), in a plug. Then for foreign key graph F the data relationship operator, $\bigotimes$, is defined as follows.*

$$\bigotimes(F, P) = \bigcup_{\forall(p,c)\in P} \otimes(F, P, p, c)$$

To relate data in a plug, P, the paths on the plug are joined using an inorder walk of the spanning tree for $\bigotimes(F, P)$.

Definition 2.2.3 (Relating Data) *Given a plug, P, and a foreign key graph, F, with closeness spanning tree, C, for $\bigotimes(F, P)$ that relates names $x_1, \ldots, x_k$ in P. Let an inorder walk of the spanning tree yield the list of relations $[R_1, \ldots, R_n]$. Then the data relationship operator, $\bowtie_P$, is defined as follows.*

$$\bowtie_P (C, [x_1, \ldots, x_k]) = \pi_{x_1,\ldots,x_k}(\bowtie [R_1, \ldots, R_n])$$

where $\bowtie$ is the left outer join (using foreign keys) of each relation in the inorder walk of C.

For example, to relate `notes` to `height`, the inorder walk for the closeness spanning tree is $[\texttt{employees}, \texttt{orders}]$. The data relationship operator applied to this list is given below.

$$\bigotimes((\{\texttt{employees}, \texttt{orders}\}, \{(\texttt{employees}, \texttt{orders})\}), \texttt{notes}, \texttt{height}) =$$
$$\pi_{\texttt{notes},\texttt{height}}(\texttt{employees} \bowtie \texttt{orders})$$

2.3 An Alternative Definition of Plug Closeness

Plug closeness as defined above is based on the closeness of parent/child relationships in a plug rather than the minimal amount of relationships overall in a plug. An alternative is to use the Steiner tree for the plug. The Steiner tree for a graph is a minimal spanning tree among a subset of nodes in the graph. Computing the Steiner tree is NP-complete [16], even for an unweighted graph (or a graph where all the weights are the same). Though many approximation techniques exist [17], it is unclear whether the Steiner tree gives a better intuitive solution to the data-relatedness problem since a plug designer may construct a plug by reasoning about parent/child relationships in a hierarchy rather than overall minimality of the edges in a plug.

2.4 Potential Information Loss

There may be more than one closeness spanning tree that connects pairs of names. For instance suppose that there is a foreign key that relates `categories` to `orders`. Then there are two paths of length two from `products` to `orders`, one through `categories` and the other through `order_details`. To determine which spanning tree to use, we rank the spanning trees by their *potential information loss.*

Definition 2.4.1 (Loss Ranking) *Let closeness spanning trees* T_1 *and* T_2 *connect names* $x_1, \ldots, x_k$*. Then* T_1 *is less complete (potentially loses more information) with respect to* T_2 *if* $\bigotimes(T_1, [x_1, \ldots, x_k]) \subset \bigotimes(T_2, [x_1, \ldots, x_k])$.

2.5 Name Ambiguity

Names in a pattern can be *ambiguous.* For example, the `city` type exists in the `suppliers`, `customers`, and `employees` relations. Names can be disambiguate by prefixing the name with the name of the relation, *e.g.,* `suppliers.city`, or, since this is a common problem in data integration, by other techniques. We assume in this thesis that names are unambiguous.

CHAPTER 3
AutoREST Architecture

In this chapter we describe the architecture of AutoREST. An overall view of the architecture is given in Figure 3.1. Each step in the process is described below.

3.1 Connecting to the Database

The AutoREST tool uses the metadata from the the database to generate queries and rank each of these web services. The tool provides an interface to connect to a database and create web services instantaneously. This also enables to check and change the web service. The user provides the database access as shown in Figure 3.2, our application extracts the database metadata using the JDBC driver. The metadata includes the table names, column names, foreign key, and primary key collections.

3.2 Parsing and Constructing the Abstract Syntax Tree

The input JSON schema is parsed using ANTLR. ANTLR is a parser generator used for language recognition and code generation [18]. In our case, the language is simply a JSON schema. The parser automatically generates an abstract syntax tree.

For the plug shown in Figure 3.3, the abstract syntax tree shown in Figure 3.4 is generated. Initially, we formed a new syntax, which the user could use to design the web service, but we quickly realized there is a considerable learning curve to learn the multiple rules of our syntax. This also made designing web services counter-intuitive. To make it as simple and intuitive as possible we then used a simple JSON schema object as an input by the user. This also enables the user to visualize the output structure and change the structure as needed.

The input data to generate a web service from the AutoREST app is a simple JSON schema object. The values in the schema represent the name of the columns in the database.

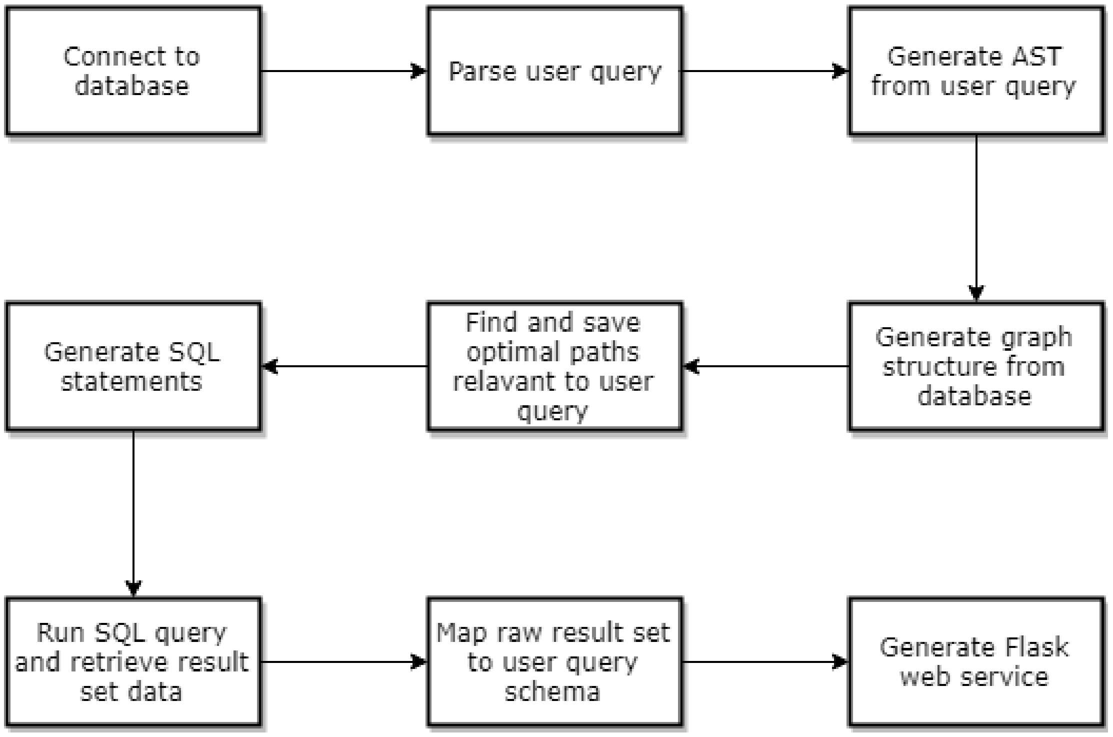
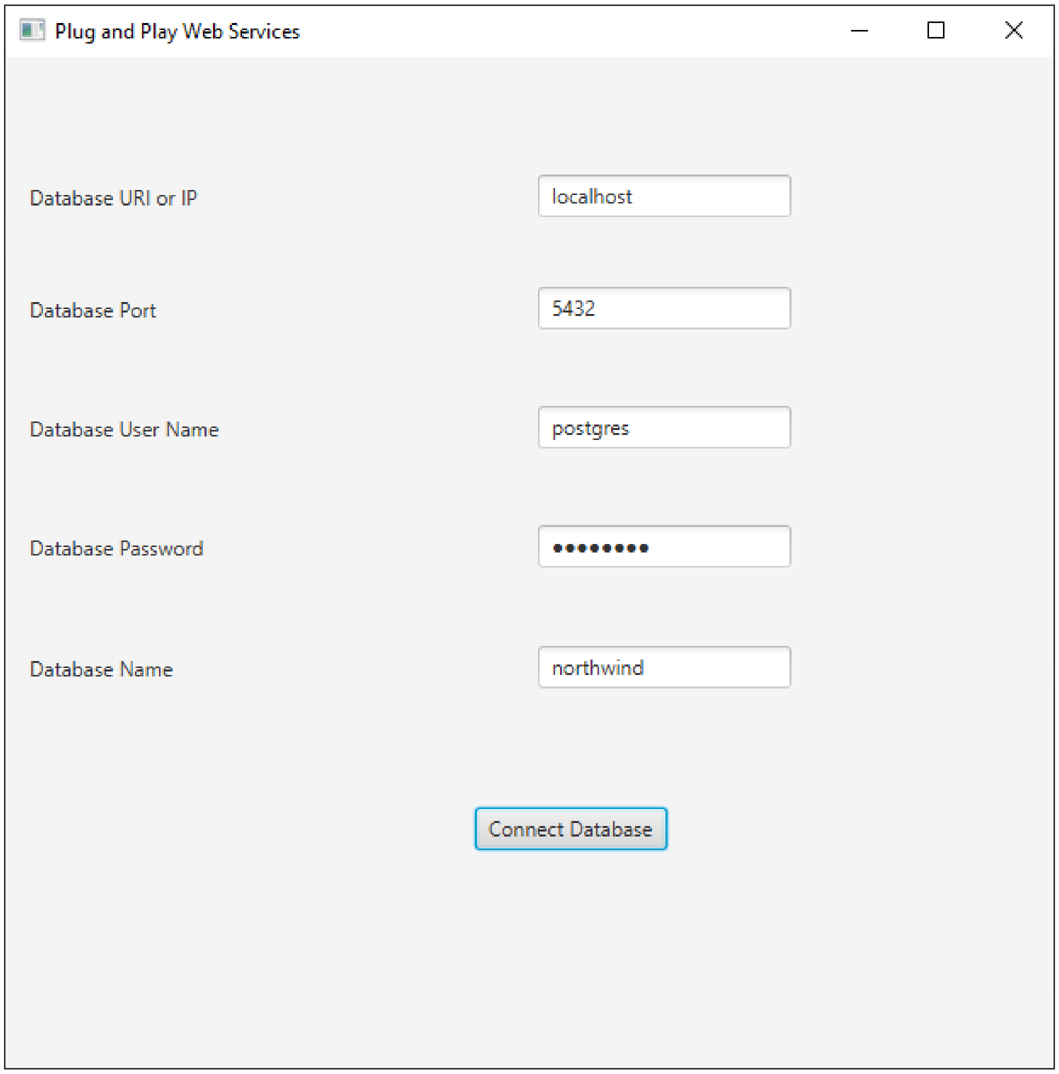

Connect to database
Parse user query
Generate AST from user query
Generate graph structure from database
Find and save optimal paths relavant to user query
Generate SQL statements
Run SQL query and retrieve result set data
Map raw result set to user query schema
Generate Flask web service

Fig. 3.1: Steps in the AutoREST process

The keys in the schema, are used to denote the name of the valuable data that is the user can rename. The input schema can also include any depth of the hierarchy is required as the structure of the web service. The input can be as dynamic as possible and there is no limitation to creating and changing the structure for the web service. Also, the user does not need to know about the underlying structure of the database.

In our implementation, we create a custom pattern tree to retain data about the hierarchy, keys, and values from the query input. To create a path between the column values in the query we build potential table names by a column lookup from all tables in the database. We then match any one of the tables containing the column information. Matching columns to their tables help us to find a relationship between the column values in the query.

The importance of generating these relations can be explained with an example. Suppose, we want to create a web service that contains the information of the customer and the orders that they have placed. The information in the database being used to generate


Plug and Play Web Services

Database URI or IP	localhost
Database Port	5432
Database User Name	postgres
Database Password	●●●●●●●●●
Database Name	northwind

Fig. 3.2: Connect to database

```
{
    "film": "title",
    "category": {
        "category_name": "name"
    },
    "actor": {
        "first_name": "first_name",
        "last_name": "last_name"
    }
}
```

Fig. 3.3: Sample web service plug

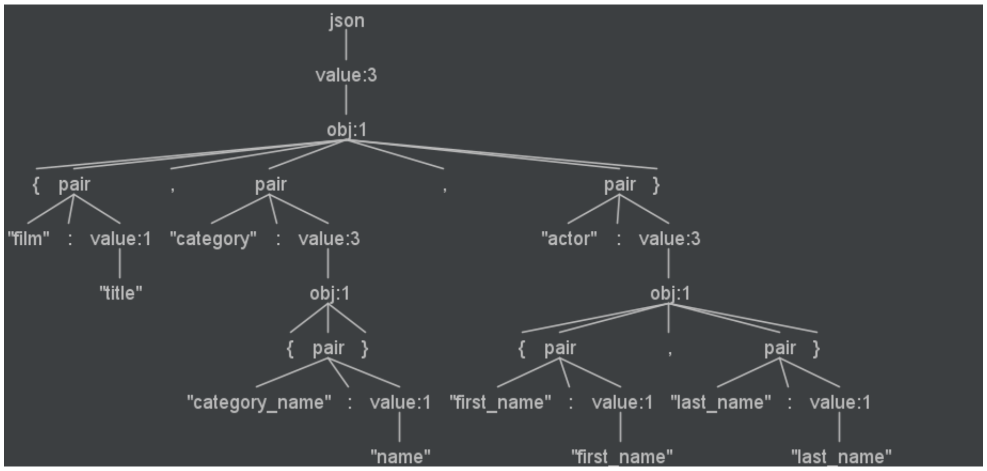


Fig. 3.4: Example of an abstract syntax tree generated

```
SELECT DISTINCT film."title", category."name",
        customer."first_name", customer."last_name"
FROM film_category
LEFT JOIN film ON film."film_id" = film_category."film_id"
LEFT JOIN inventory ON film."film_id" = inventory."film_id"
LEFT JOIN category ON category."category_id" = film_category."category_id"
LEFT JOIN rental ON inventory."inventory_id" = rental."inventory_id"
LEFT JOIN customer ON rental."customer_id" = customer."customer_id"
ORDER BY film."title", category."name",
        customer."first_name", customer."last_name"
```

Fig. 3.5: SQL query that is generated

the web service is present in two different tables namely `customers` and `orders` as shown in Figure 2.1. These tables are related to each other using a foreign key and so to get the required data we need to perform a JOIN operation on these tables. Similarly, if we wanted to generate a web service that contains the information for customers and the products that they have purchased, we would need to join the tables `customers`, `orders` and `order_details` since the product information is present only in the `order_details` table which is related to the table `orders`. The SQL query shown in Figure 3.5 is generated.

Here we can observe that relations between two tables can be direct or indirect. These relations can then be represented in terms of paths. In our implementation, we automatically generate these using the graph data structure.

3.3 Generating the Foreign Key Graph and Finding Optimal Paths

Converting the database to an undirected graph enables us to find meaningful relations or paths amongst tables. There can be several possible paths between two tables in a database. We want to form our query that is efficient and retains the most information possible. In other words, we want to find all the shortest paths between any two tables and choose the best from those paths.

We use a modified breadth-first search algorithm to find all the shortest paths between two tables. As a preprocessing step, we find all the shortest paths between all tables in the database. We do this preprocessing since there are a limited number of tables in a database and it is very unlikely to have a very large number (thousands) of tables in a database so in usage the processing time is negligible.

In our implementation, we first begin with storing paths of all the directly connected tables in the database. Then we simply use these paths to generate the shortest paths between tables that are connected through an intermediary table and so on. We continue this until all shortest paths have been traversed and stored. After the preprocessing we now we have a collection of shortest paths between any two tables in the database which will help us build SQL queries for our input query.

Our specification states that the input query can have any sort of hierarchy and is also free of the structure of data in the database. In our implementation, we dynamically generate paths based on each column value from the query.

Algorithm 1: Algorithm to generate tree paths

```
Input: Graph $G$
$G$ is built after preprocessing
Output: Map $M$
$M$ contains all the paths required to build the query
Procedure savePaths(Graph $G$, PatternTree $T$, String $p$, Map $M$)
$c$ are the children
$p$ is the parent of the current PatternTree object
$l$ is the list of paths betweeen two nodes
$prev$ is the previous child object
begin
    // iterate through the children of the root
    forall $c \in T$ do
        // Traverse down the tree
        if $c.table.value \neq null$ then
            if isNotRoot($T$) then
                // Get all paths between two tables
                $l \leftarrow$ getPaths($table, c.table$)
                // Append previous paths to add new connections
                $M \leftarrow$ mergePaths($G, l, M$)
            else
                // Recursively call savePaths
                $M \leftarrow$ savePaths($G, c, c.table, M$)
        else
            $M \leftarrow$ savePaths($G, c, p.table, M$)
    // Navigate between children
    if $T.c.size() > 1$ then
        forall $c \in T$ do
            if $prev \neq null$ then
                $l \leftarrow$ getPaths($table, c.table$)
                $M \leftarrow$ mergePaths($G, l, M$)
            else
                $prev \leftarrow c$
return $M$
```

The algorithm is shown in Algorithm 1. There are several cases of how queries are processed as in the remainder of this section.

```
{
    "order_key": "o_orderkey",
    "order_status": "o_orderstatus"
}
```

Fig. 3.6: Simple, single table plug

3.3.1 Case: Single table plug

Suppose we want to create a simple web service that returns an orders key and status information from the Northwind database using the plug shown in Figure 3.6. We process the plug as described previously using the following steps.

1. Parsing the JSON string to an abstract syntax tree
2. Creating a pattern tree data structure storing the hierarchy and values
3. Finding potential tables for input columns
4. Generating shortest paths amongst all tables in the database

After the above processing, we have data for the `SELECT` and `ORDER BY` keywords. To create a meaningful query we need to find the join conditions between the columns in the database. In this section, we will discuss the algorithm we use to process queries to generate paths or joins conditions for the query.

For a given search query we first begin with the first column value and then we find the relation to the next column value in the same hierarchy. For the above query, it is `o_orderkey` and `o_orderstatus`, respectively.

Looking at the schema in Figure 3.20 we see that both the column values are in the same table, hence we have a same table relation. To get the data for the `FROM` keyword, all we need is the name of the table and since it is a same table relation there is a need for a join condition. Therefore, for the above search query, our algorithm generates the SQL query shown in Figure 3.7. Similarly, if the query had more columns from the same table then we would only need to add the column name data in the `SELECT` and `ORDER BY` keywords.

```
SELECT DISTINCT orders."o_orderkey", orders."o_orderstatus"
FROM orders
ORDER BY orders."o_orderkey", orders."o_orderstatus
```

Fig. 3.7: SQL for the single table plug shown in Figure 3.6

```
{
    "customer_name": "c_name",
    "order_key": "o_orderkey",
    "order_totalprice": "o_totalprice",
    "order_status": "o_orderstatus"
}
```

Fig. 3.8: Multi-table plug

3.3.2 Case: Multi-table plug

Suppose we want to create a web service to find the names of the customer and the order information that the customer has placed, then we would use the plug given in Figure 3.8. Again we get the data for `SELECT` and `ORDER BY` from the initial processing stages of the search query.

Now we need to find the relation amongst the different columns in the query. As described earlier we start with finding the relation `c_name` and the next column value `o_orderkey`. We find that this is a multi-table relation which means that the value in the columns is from different tables. So now we need to find a join condition to make a meaningful query.

Since we have already calculated all the possible join condition between all the tables we only need to retrieve meaningful that result. To find a relation between two tables we simply retrieve the join condition between tables - customers and orders and create a query-specific path. Now we move to the next column and generate a relationship with the first column value in the root hierarchy, `c_name`. Since we already have the join condition for these tables, we move to the next column value and so on. Finally, the SQL query generated is given in Figure 3.9.

As an alternative we could have also added the join condition on the query using the `WHERE` keyword as shown in Figure 3.10. The reason we are using left joins is if there were

```
SELECT DISTINCT customer."c_name", orders."o_orderkey",
                orders."o_totalprice", orders."o_orderstatus"
FROM orders
LEFT JOIN customer ON customer."c_custkey" = orders."o_custkey"
ORDER BY customer."c_name", orders."o_orderkey",
                orders."o_totalprice", orders."o_orderstatus"
```

Fig. 3.9: Generated SQL query for plug in Figure 3.8

```
SELECT DISTINCT customer."c_name", orders."o_orderkey",
                orders."o_totalprice", orders."o_orderstatus"
FROM orders, customer
WHERE customer."c_custkey" = orders."o_custkey"
ORDER BY customer."c_name", orders."o_orderkey",
                orders."o_totalprice", orders."o_orderstatus"
```

Fig. 3.10: Alternative generated SQL query

customers that did not place any orders then we would not get their names in the result set. Since we want to get all the entries for customer names we use left joins. Using left joins we get null values for the column that do not have data.

3.3.3 Case: Multi-hierarchy plug

Suppose that the plug is as given in Figure 3.11. Finding relations in a hierarchical query is different from a flat query. Rather than finding a relation of a column value with the root column value, we find the relations between the column value in the hierarchy

```
{
    "customer_name": "c_name",
    "order_details": {
        "order_key": "o_orderkey",
        "order_totalprice": "o_totalprice",
        "order_status": "o_orderstatus",
        "tax": {
            "line_tax": "l_tax"
        }
    }
}
```

Fig. 3.11: A multi-hierarchy plug

```
SELECT DISTINCT customer."c_name", orders."o_orderkey",
        orders."o_totalprice", orders."o_orderstatus", lineitem."l_tax"
FROM lineitem
LEFT JOIN orders ON orders."o_orderkey" = lineitem."l_orderkey"
LEFT JOIN customer ON customer."c_custkey" = orders."o_custkey"
ORDER BY customer."c_name", orders."o_orderkey",
        orders."o_totalprice", orders."o_orderstatus", lineitem."l_tax"
```

Fig. 3.12: SQL for the multi-hierarchy plug of Figure 3.11

```
{
    "customer": "c_name",
    "phone": "c_phone",
    "parts": {
        "part": "p_name",
        "part_brand": "p_brand"
    }
}
```

Fig. 3.13: A multiple path plug

with the first column value in the parent object. For the above query, we find the relation between `o_orderkey`, `o_totalprice`, and `o_orderstatus` with `c_name` and similarly after that we find the relation between `l_tax` and `o_orderkey`.

The query generated for this plug is shown in Figure 3.12. We create the hierarchical structure from the result set after executing the query.

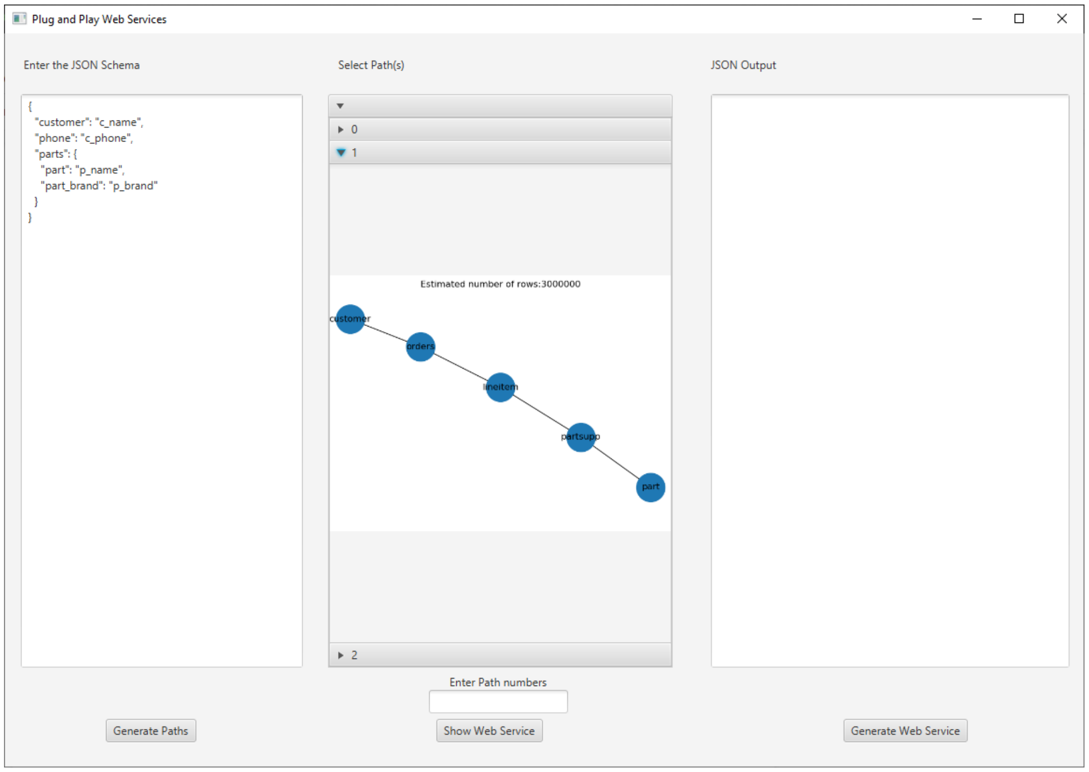
3.3.4 Case: Multiple path plug

There could be multiple shortest paths connecting two relations. An example of a multiple path plug is given in Figure 3.13. The algorithm processes similarly as explained before for hierarchical queries, but here we see that there is more than one shortest path between the tables customer and part. One path is

```
Customer - orders - lineitem - partsupp - part
```

while the other is given below.

```
Customer - nation - supplier - partsupp - part
```



```
{
 "customer": "c_name",
 "phone": "c_phone",
 "parts": {
  "part": "p_name",
  "part_brand": "p_brand"
 }
}
```

Fig. 3.14: Path visualization in AutoREST

Such a situation is quite likely to occur in a database with several relationship types between a pair of entity types. Our aim is to provide the ability to create the most optimal web service and it is quite likely that one of these paths is better than the other. The query for the first path is shown in Figure 3.15 while for the second path the query is given in Figure 3.16.

To enable the user to choose a desirable path, AutoREST visualizes the paths as shown in Figure 3.14. The left pane in the figure shows a pattern. The middle pane is the alternative paths, shown one at a time. The right pane is the generated query.

We measure the desirability of the web service on the basis of rows returned, which is an indicator of the completeness of the plug computation. The more rows being returned from a query implies that less data is being lost with the join condition. The rows can be either counted by executing the query and then counting the rows or using some kind of estimation. By now we have shown examples of complex queries that have multiple join

```
SELECT DISTINCT customer."c_name", customer."c_phone",
            part."p_name", part."p_brand"
FROM lineitem
LEFT JOIN orders ON orders."o_orderkey" = lineitem."l_orderkey"
LEFT JOIN partsupp ON lineitem."l_suppkey" = partsupp."ps_suppkey"
LEFT JOIN customer ON customer."c_custkey" = orders."o_custkey"
LEFT JOIN part ON partsupp."ps_partkey" = part."p_partkey"
ORDER BY customer."c_name", customer."c_phone",
            part."p_name", part."p_brand"
```

Fig. 3.15: First SQL for the multi-path plug of Figure 3.13

```
SELECT DISTINCT customer."c_name", customer."c_phone",
            part."p_name", part."p_brand"
FROM partsupp
LEFT JOIN supplier ON supplier."s_suppkey" = partsupp."ps_suppkey"
LEFT JOIN nation ON nation."n_nationkey" = supplier."s_nationkey"
LEFT JOIN customer ON customer."c_nationkey" = nation."n_nationkey"
LEFT JOIN part ON partsupp."ps_partkey" = part."p_partkey"
ORDER BY customer."c_name", customer."c_phone",
            part."p_name", part."p_brand"
```

Fig. 3.16: Second SQL for the multi-path plug of Figure 3.13

```
SELECT DISTINCT customer."c_name", customer."c_phone",
            part."p_name", part."p_brand"
FROM lineitem
LEFT JOIN orders ON orders."o_orderkey" = lineitem."l_orderkey"
LEFT JOIN partsupp ON lineitem."l_suppkey" = partsupp."ps_suppkey"
LEFT JOIN supplier ON supplier."s_suppkey" = partsupp."ps_suppkey"
LEFT JOIN nation ON nation."n_nationkey" = supplier."s_nationkey"
LEFT JOIN customer ON customer."c_nationkey" = nation."n_nationkey"
LEFT JOIN part ON partsupp."ps_partkey" = part."p_partkey"
 UNION
SELECT DISTINCT customer."c_name", customer."c_phone",
            part."p_name", part."p_brand"
FROM lineitem
LEFT JOIN orders ON orders."o_orderkey" = lineitem."l_orderkey"
LEFT JOIN partsupp ON lineitem."l_suppkey" = partsupp."ps_suppkey"
LEFT JOIN customer ON customer."c_custkey" = orders."o_custkey"
LEFT JOIN part ON partsupp."ps_partkey" = part."p_partkey"
ORDER BY "c_name", "c_phone", "p_name", "p_brand"
```

Fig. 3.17: UNION of two paths in SQL for the multi-path plug of Figure 3.13

conditions, and these joins are very costly in terms of time and memory usage, so executing the query and then counting the rows is a very costly process. The `EXPLAIN ANALYZE` keyword in PostgreSQL does this exactly. That is the reason it is more efficient to generate some sort of estimation.

In our implementation, we use the `EXPLAIN` keyword for the PostgreSQL database to find the estimated number of rows that will be output for the given query. PostgreSQL makes this estimation on the basis of the metadata of the database.

This enables the user to determine the optimal web service. Our application shows the estimated number of rows and also present the user with a graphical representation of the path of the join condition in the database.

It is also possible that two queries generate a result set exclusive to the query and the user might want to use all the unique results from both queries, for that we use the `UNION` keyword to combine the unique results from multiple paths. Combining both queries for the above example will be as follows -

It is also possible for two queries to have the same paths but different join conditions

```
{
  "supplier1": "s_name",
  "related_to": {
    "supplier2": "s_name"
  }
}
```

Fig. 3.18: A same table plug

```
SELECT DISTINCT t1."s_name", t2."s_name"
FROM (SELECT DISTINCT supplier."s_name", partsupp."ps_partkey"
FROM partsupp
LEFT JOIN supplier ON supplier."s_suppkey" = partsupp."ps_suppkey"
ORDER BY supplier."s_name", partsupp."ps_partkey") t1
LEFT JOIN (SELECT DISTINCT supplier."s_name", partsupp."ps_partkey"
FROM partsupp
LEFT JOIN supplier ON supplier."s_suppkey" = partsupp."ps_suppkey"
ORDER BY supplier."s_name", partsupp."ps_partkey") t2
ON t1.ps_partkey = t2.ps_partkey
ORDER BY t1."s_name", t2."s_name"
```

Fig. 3.19: SQL for the same table plug of Figure 3.18

in the case of a table having more than one foreign key and our implementation finds both paths.

3.3.5 Case: Recursive plug

Suppose we want to generate a web service that a supplier that is related to itself. An example plug is shown in Figure 3.18. We find the relation between the two same columns by joining the table of the column to a table that is directly connected to the column table. We then create two aliases of the result set and do a left join between them, with the join condition being the primary key of the directly connected table. For the above query, we first join the supplier table to `partsupp`, a table that is directly connected to the supplier table and add the primary key of the table `partsupp` in the `SELECT` keyword. Then two aliases of the are created and both are left joined on the primary key of `partsupp` table. The result set of the query (the query for the plug is given in Figure 3.19) shows which suppliers are related.

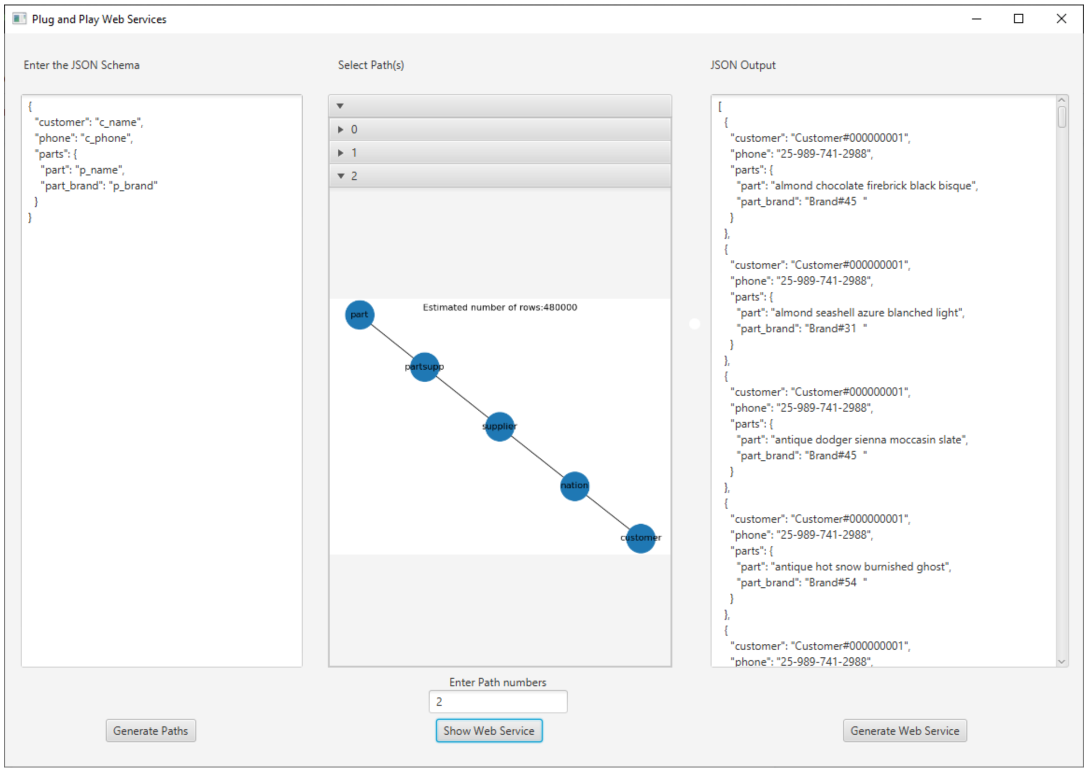
```
SELECT DISTINCT t1.employee_id, t2.employee_id
FROM employees t1
LEFT JOIN employees t2
ON t2.employee_id = t1.reports_to
ORDER BY t1.employee_id, t2.employee_id
```

Fig. 3.20: SQL for same table reference in northwind database

It is also possible that the table has a foreign key that references itself. For example, the `employees` table in the Northwind database has a foreign key `reports_to` that references `employees`. Our algorithm first checks if such a foreign key exists, if it does then it does not process the self join as explained above. The query then generated is shown in Figure 3.20.

3.4 Formatting the Result

After executing the query, the result set is saved as a flat JSON object. This object is then converted back to the input query structure using string manipulation. The user is able to see and interactively change the web service output in the application as shown in Figure 3.21.



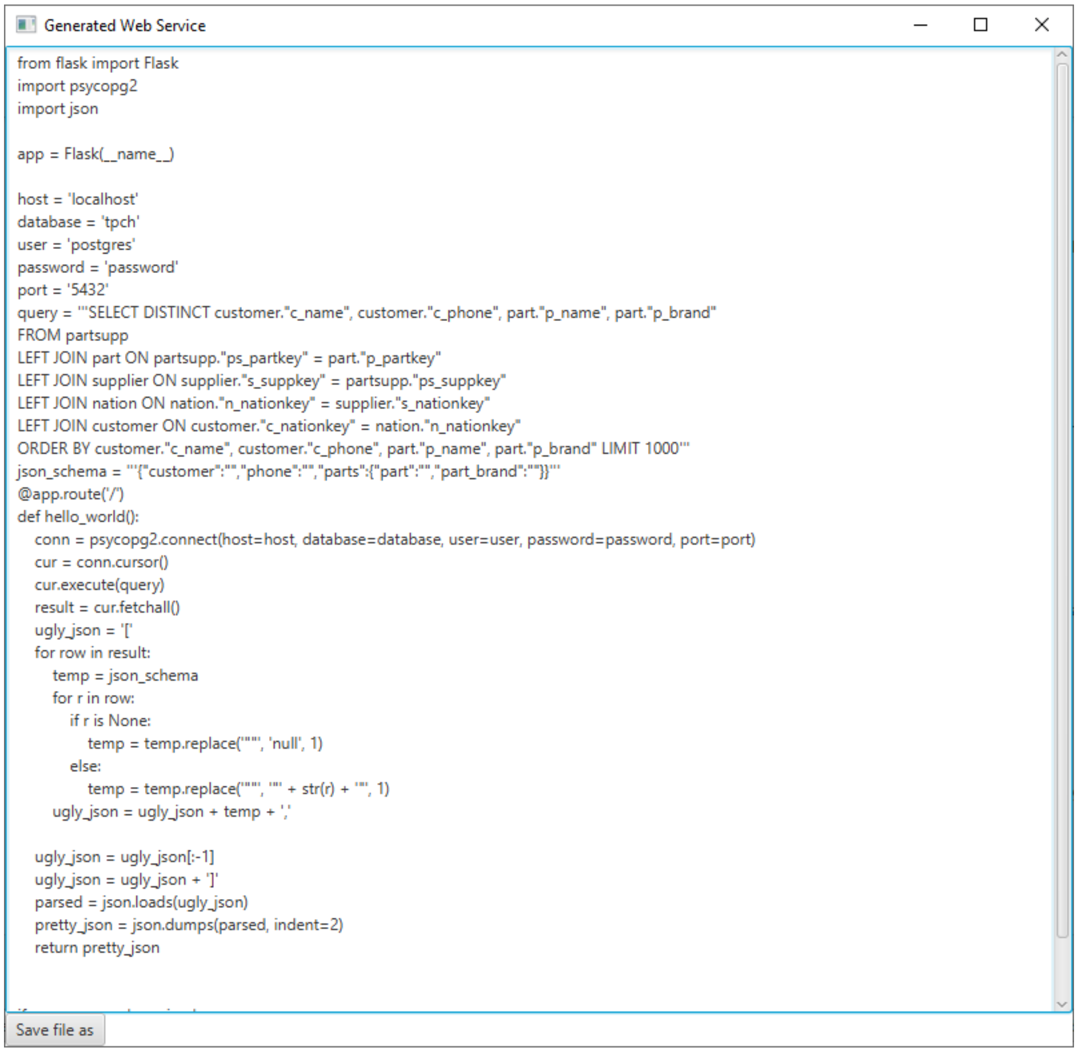
```
{
  "customer": "c_name",
  "phone": "c_phone",
  "parts": {
    "part": "p_name",
    "part_brand": "p_brand"
  }
}
```

```
[
 {
  "customer": "Customer#000000001",
  "phone": "25-989-741-2988",
  "parts": {
   "part": "almond chocolate firebrick black bisque",
   "part_brand": "Brand#45  "
  }
 },
 {
  "customer": "Customer#000000001",
  "phone": "25-989-741-2988",
  "parts": {
   "part": "almond seashell azure blanched light",
   "part_brand": "Brand#31  "
  }
 },
 {
  "customer": "Customer#000000001",
  "phone": "25-989-741-2988",
  "parts": {
   "part": "antique dodger sienna moccasin slate",
   "part_brand": "Brand#45  "
  }
 },
 {
  "customer": "Customer#000000001",
  "phone": "25-989-741-2988",
  "parts": {
   "part": "antique hot snow burnished ghost",
   "part_brand": "Brand#54  "
  }
 },
 {
  "customer": "Customer#000000001",
  "phone": "25-989-741-2988",
```

Fig. 3.21: Repopulated data in AutoREST

3.5 Creating the Service

In our implementation, we auto generate a Python script using the Flask framework to create the web service. An example of the Flask script is shown in Figure 3.22.



```
from flask import Flask
import psycopg2
import json

app = Flask(__name__)

host = 'localhost'
database = 'tpch'
user = 'postgres'
password = 'password'
port = '5432'
query = '''SELECT DISTINCT customer."c_name", customer."c_phone", part."p_name", part."p_brand"
FROM partsupp
LEFT JOIN part ON partsupp."ps_partkey" = part."p_partkey"
LEFT JOIN supplier ON supplier."s_suppkey" = partsupp."ps_suppkey"
LEFT JOIN nation ON nation."n_nationkey" = supplier."s_nationkey"
LEFT JOIN customer ON customer."c_nationkey" = nation."n_nationkey"
ORDER BY customer."c_name", customer."c_phone", part."p_name", part."p_brand" LIMIT 1000'''
json_schema = '''{"customer":"","phone":"","parts":{"part":"","part_brand":""}}'''
@app.route('/')
def hello_world():
    conn = psycopg2.connect(host=host, database=database, user=user, password=password, port=port)
    cur = conn.cursor()
    cur.execute(query)
    result = cur.fetchall()
    ugly_json = '['
    for row in result:
        temp = json_schema
        for r in row:
            if r is None:
                temp = temp.replace('""', 'null', 1)
            else:
                temp = temp.replace('""', '"' + str(r) + '"', 1)
        ugly_json = ugly_json + temp + ','

    ugly_json = ugly_json[:-1]
    ugly_json = ugly_json + ']'
    parsed = json.loads(ugly_json)
    pretty_json = json.dumps(parsed, indent=2)
    return pretty_json
```

Fig. 3.22: Flask web service code

CHAPTER 4
Evaluation

In this section we describe the results of several experiments to evaluate AutoREST. The evaluation measures *feasibility* of plug-and-play web services. We measure the time taken to generate a web service. Since the service is typically generated once and then reused, the time taken is not critical. However, we explore two alternatives in cost estimation in an SQL query compiler while creating a web service using AutoREST.

4.1 Experiments Environment

We performed our experiments on a desktop machine with an i7-4770 CPU with a clock speed of 3.40 GHz and 16 GB of DDR3 memory. The OS used is Ubuntu 18 LTS, 64-bit and the Java version used in both developing and testing is version 8. We performed the experiments using the Postgres DBMS version 10. We used an out-of-the box version of both Postgres and Java, with no adjustments made for performance tuning, such as increasing cache memory size.

4.2 EXPLAIN vs. EXPLAIN ANALYZE Experiment

In this experiment we used a standard relational benchmark database, TPC-H [19]. The TPC-H database generator can be used to generate a database of a given size. For this experiment we used a database of size 10MB. We generated seven plugs, which are given in the Appendix **??**. We measured the total time to create the web service, from input of the plug to completion of code creation. We tested using `EXPLAIN` vs. `EXPLAIN ANALYZE` to resolve shortest paths queries. The difference between `EXPLAIN` vs `EXPLAIN ANALYZE` is that the former estimates the cost of a query from database statistics, while the latter runs the query capturing the actual cost. Estimating query cost is much faster than running a query and measuring the cost.

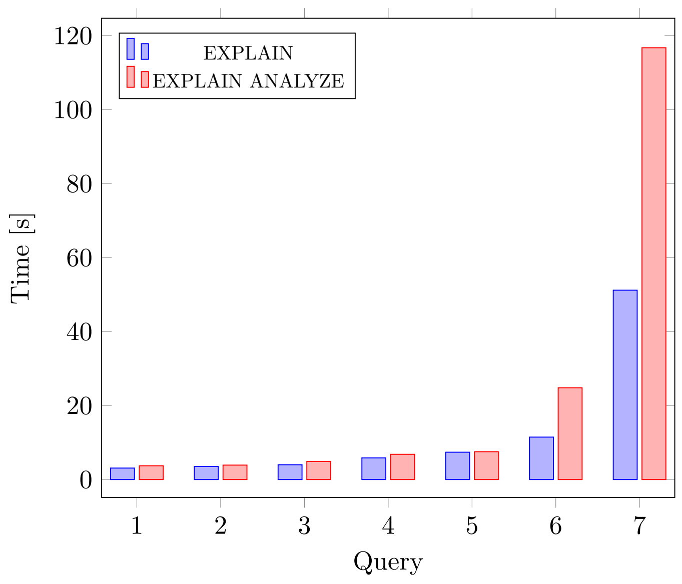


Fig. 4.1: Timing `EXPLAIN` vs. `EXPLAIN ANALYZE` on seven plugs of increasing complexity

Figure 4.1 plots the cost of generating the web service for each plug. The plugs increase in complexity from plug one to plug seven, and therefore in cost. The experiment also shows that using `EXPLAIN ANALYZE` is more expensive for complex plugs, for plugs six and seven it is more than double the cost.

`EXPLAIN ANALYZE` takes more time, but it is unclear if it is producing a "better" result. The quality differences between `EXPLAIN` and `EXPLAIN ANALYZE` can be measured by examining how close the former comes to estimating the number of rows in the query result, which is what we use for gauging completeness and ranking paths. Figure 4.2 shows the percent difference in the queries corresponding to the seven plugs. The query size estimator in Postgres accurately predicts the size of the result for most of the queries, only query 2 shows significant differences. We observed that sometimes the query estimator overestimates the number of output rows for queries that involve `DISTINCT`, which eliminates duplicate rows from the query result. In any case, for practical purposes, `EXPLAIN` seems the better alternative.

4.3 Pagination Experiment

We also measured the time taken to produce the first result. Pagination is typically

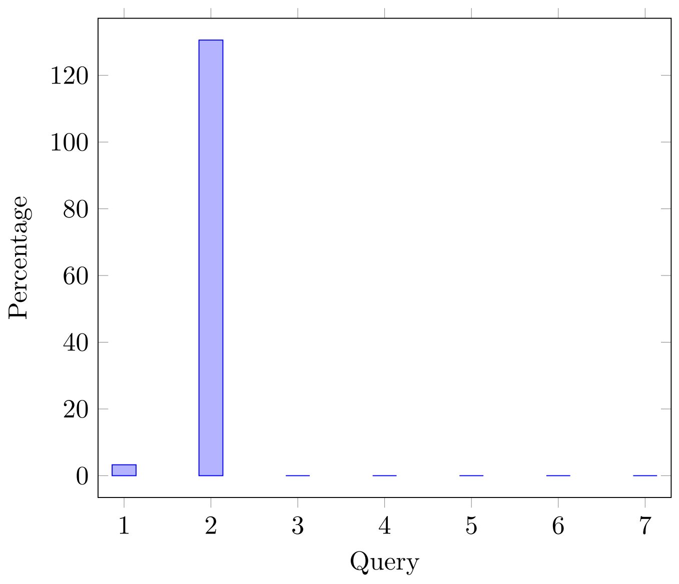


Fig. 4.2: Percentage difference of number of rows in EXPLAIN vs EXPLAIN ANALYZE

used for web services, so the time to the first result is essentially the time to produce the first page. Figure 4.3 shows the difference in the cost of computing the first result vs. the complete result using `EXPLAIN`.

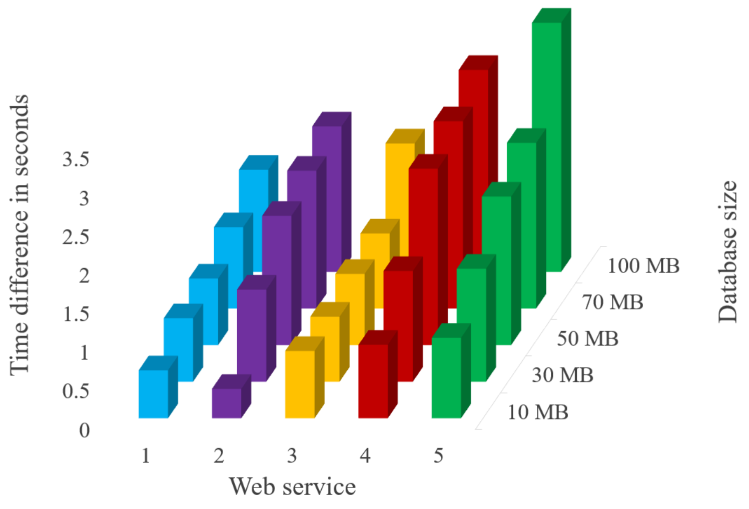


Fig. 4.4: Time difference between EXPLAIN and EXPLAIN ANALYZE

4.4 Database Size Experiment

The previous experiments used a relatively small database, so we were interested in

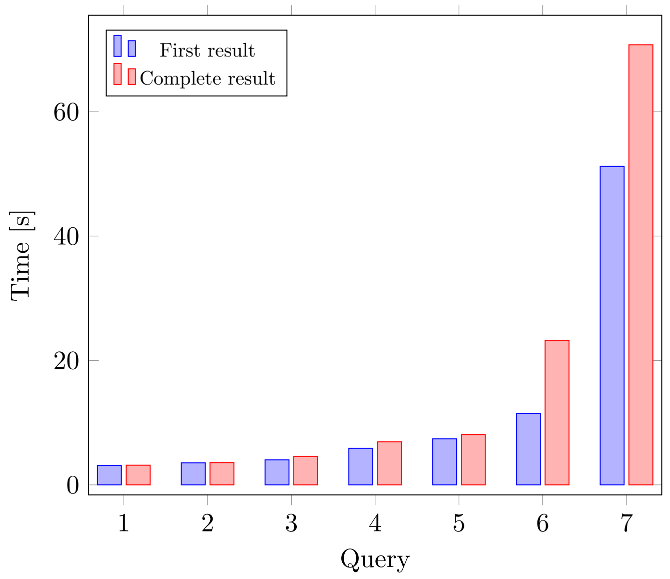


Fig. 4.3: Timing first result vs complete result set

determining how the size of the database impacted the time taken. Figure 4.4 plots the time difference between `EXPLAIN` and `EXPLAIN ANALYZE` for the seven plugs on databases of increasing size. The results show that as the database size increases, the time difference also increases, which means that `EXPLAIN ANALYZE` takes longer with larger databases. We have only included results from the initial five queries as the time difference in the last two queries is extremely large and makes it hard to notice the subtler difference among the less complicated queries.

CHAPTER 5
Related Work

Previous hierarchy-related research in transforming the shape of data can be broadly classified into several categories.

Query relaxation/approximation - One way to loosen the tight coupling of path expressions to the hierarchy of data is to relax the path expressions or approximately match them to the data by exploring a space of hierarchies that are within a given edit distance [20–22]. Though such techniques work well for small variations in hierarchy or values, there can be a *very large* edit distance among a pair of instances which we would like to consider as the same data. Relaxing a query to explore all hierarchies within a large edit distance is overly permissive, and includes many hierarchies which do not have the same data. Query correction [23] and refinement [24] approaches are also best at exploring only small changes to the hierarchy.

Hierarchical search engines - Hierarchical search engines (*c.f.,* [25,26]) de-couple queries from specific hierarchies, similar to our aims, and can find data in differently-shaped hierarchies. But like query relaxation, search engines are overly permissive, and once data is found, a search engine does not transform that data to a hierarchy needed by a query.

Structure-independent querying - The idea of using a least common ancestor (LCA) has been explored for querying data independent of its hierarchy. Schema-free XQuery uses the meaningful LCA [27], XSeek exploits node interconnections [26], and others use the smallest LCA [28]. Similarly, we proposed a closest XPath axis [7] based on the LCA. In contrast to all of the above research, none of these approaches are *data transformations*, that is they do not transform *values* (subtrees in the hierarchy), rather they can only utilize values from the source hierarchy; data transformations need to produce values in the shape of the target hierarchy.

Declarative transformations - There are declarative languages for specifying transformations of hierarchical data [29, 30]. These languages hide from users many specification details that would be needed in a language such as XSLT. However, each transformation depends on the hierarchy of the input and would have to be re-programmed for a different hierarchy. It would be more desirable if a programmer could simply declare the desired hierarchy in a single plug-and-play specification.

Data integration - Data can be integrated from one or more source schemas to a target schema by specifying a mapping to carry out a specific, fixed transformation of the data [4]. Once the data is in the target schema, there is still the problem of queries that need data in a hierarchy other than the target schema. In some sense schema mediators integrate data to a fixed schema, which is the starting point for what plug-and-play web services aims to do. The different problem leads to a difference in techniques used to map or transform the data. For instance tuple-generating dependencies (TGDs) are a popular technique for integrating schemas [3, 5]. Part of a TGD is a specification of the source hierarchy from which to extract the data. Specifying the source hierarchy will not work for plug-and-play web services, which must be agnostic about the source. A second concern for is that the web service construction must be fully automatic. A third difference is the need to determine potential information loss, which is (largely) absent from such mappings for data integration. For schema mediation if a programmer programs a data transformation that loses information, that information is gone and subsequent path expressions on the transformed data will never know about the information loss. Fan and Bohannon explored preserving information in data integration, namely by describing schema embeddings that ensure invertible mappings that are query preserving [31]. We focus on an important special case of the mappings they investigated. Query preservation concerns all possible queries, while plug-and-play specifications are designed to check a single pattern. Our approach for quickly determining whether a mapping is invertible (or in our terminology reversible) is based on the concept of *closeness*, and in those cases where a mapping is not reversible we can identify weaker, but still useful classes of mappings that permit some information loss.

We note that Codd's solution [32] to the problem of querying different hierarchies, which is to replace the hierarchical model with the relational model, is orthogonal to plug-and-play web services, which are also desirable for relational languages where differently-shaped data is less severe but still present.

We also note that our research will focus only on the *structure, not the semantics,* of the data because Semantic Web technologies, *i.e.,* ontologies, already address the orthogonal semantic matching problem. Hence, solutions developed by the semantic Web community can be used to semantically match in plug-and-play queries.

Plug-and-play web services are similar to GraphQL [33], but GraphQL does not address any of the technical challenges in Section 1. GraphQL is a API-side tool. That is, in GraphQL a user can "query" a web service, by choosing a subset of the information available from the service. A designer must supply the schema on the server-side and specify precisely how to construct the data, that is, the designer must construct the web service to use GraphQL.

CHAPTER 6
Conclusion and Future Work

This thesis presents a system called AutoREST that leverages the relational database to generate a web service, while also preserving the underlying relationships in the database. AutoREST is a plug-and-play web service that generates a web service from a simple JSON specification of the output of the service. We described how the specification is used to compute the hierarchical output from relational tables and how different attributes are related in a foreign key graph. The graph is also used to determine completeness of the generated service. AutoREST essentially eliminates the need for any prior coding knowledge to create a web service. The tool also enables fast web service creation and interactivity, to quickly modify the web service. This thesis describes the model for AutoREST, how AutoREST is implemented, and an experimental evaluation.

In future, it is important to extend AutoREST with a semantic matching technique whereby the semantics of a plug can be matched with that of a database. In this thesis we relied on syntactic matching. AutoREST currently only supports PostgreSQL database, we propose to make AutoREST generic across all relational databases using similar features that we used to implement from the PostgreSQL database. AutoREST only generates GET services, we would also like to extend AutoREST to support PUT, POST and DELETE services. Finally, user evaluation of AutoREST is critical. We would like to compare the time taken and usability of AutoREST with other approaches to web service creation. We anticipate that AutoREST will lower the time and effort, but our hypothesis has yet to be tested.

REFERENCES

[1] Swagger.io, "Swagger ui — api development tools — swagger [online]," https://swagger.io/tools/swagger-ui/, 2019, (Accessed on 07/23/2019).

[2] Doctrine, "Object relational mapper - doctrine : Php open source project [online]," https://www.doctrine-project.org/projects/orm.html, 2019, (Accessed on 07/23/2019).

[3] R. Fagin, L. Haas, M. Hernandez, R. Miller, L. Popa, and Y. Velegrakis, "Clio: Schema Mapping Creation and Data Exchange," in *LNCS 5600*, 2009, pp. 198–236.

[4] M. Bhide, M. Agarwal, A. Bar-Or, S. Padmanabhan, S. Mittapalli, and G. Venkatachaliah, "XPEDIA: XML ProcEssing for Data Integration," *PVLDB*, vol. 2, no. 2, pp. 1330–1341, 2009.

[5] H. Jiang, H. Ho, L. Popa, and W.-S. Han, "Mapping-driven XML Transformation," in *WWW*, 2007, pp. 1063–1072.

[6] R. Unadkat, "Survey paper on semantic web," *Int. J. Adv. Pervasive Ubiquitous Comput.*, vol. 7, no. 4, pp. 13–17, Oct. 2015. [Online]. Available: http://dx.doi.org/10.4018/IJAPUC.2015100102

[7] S. Zhang and C. E. Dyreson, "Symmetrically Exploiting XML," in *WWW*, 2006, pp. 103–111.

[8] C. Dyreson and S. Zhang, "The Benefits of Utilizing Closeness in XML," in *DEXA Work.*, 2008, pp. 269–273.

[9] C. Dyreson, S. Bhowmick, A. Jannu, K. Mallampalli, and S. Zhang, "XMorph: A Shape-polymorphic, Domain-specific XML Data Transformation Language," in *ICDE*, 2010, pp. 844–847.

[10] C. E. Dyreson, S. S. Bhowmick, and K. Mallampalli, “Using XMorph to Transform XML Data,” *PVLDB*, vol. 3, no. 2, pp. 1541–1544, 2010.

[11] C. E. Dyreson and S. S. Bhowmick, “Querying XML Data: As You Shape It,” in *ICDE*, 2012, pp. 642–653.

[12] B. Q. Truong, S. S. Bhowmick, and C. E. Dyreson, “SINBAD: Towards Structure-Independent Querying of Common Neighbors in XML Databases,” in *DASFAA (1)*, 2012, pp. 156–171.

[13] C. E. Dyreson, S. S. Bhowmick, and R. Grapp, “Querying virtual hierarchies using virtual prefix-based numbers,” in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 791–802. [Online]. Available: http://doi.acm.org/10.1145/2588555.2610506

[14] ——, “Virtual exist-db: Liberating hierarchical queries from the shackles of access path dependence,” *PVLDB*, vol. 8, no. 12, pp. 1932–1943, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1932-dyreson.pdf

[15] C. E. Dyreson and S. S. Bhowmick, “Plug-and-play queries for temporal data sockets,” in *Flexible Query Answering Systems - 12th International Conference, FQAS 2017, London, UK, June 21-22, 2017, Proceedings*, 2017, pp. 124–136. [Online]. Available: https://doi.org/10.1007/978-3-319-59692-1_11

[16] R. M. Karp, “Reducibility among combinatorial problems,” in *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, 1972, pp. 85–103. [Online]. Available: https://people.eecs.berkeley.edu/~luca/cs172/karp.pdf

[17] S. Beyer and M. Chimani, “Strong steiner tree approximations in practice,” *J. Exp. Algorithmics*, vol. 24, no. 1, pp. 1.7:1–1.7:33, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3299903

[18] T. Parr, “Antlr [online],” https://www.antlr.org/, (Accessed on 08/01/2019).

[19] TPC.org, “Tpc-h homepage [online],” https://tpc.org/tpch/, 2019, (Accessed on 07/22/2019).

[20] S. Amer-Yahia, S. Cho, and D. Srivastava, “Tree Pattern Relaxation,” in *EDBT*, 2002, pp. 496–513.

[21] N. Augsten, M. H. Böhlen, and J. Gamper, “The *pq*-gram distance between ordered labeled trees,” *ACM Trans. Database Syst.*, vol. 35, no. 1, pp. 4:1–4:36, 2010. [Online]. Available: https://doi.org/10.1145/1670243.1670247

[22] Y. Kanza and Y. Sagiv, “Flexible Queries over Semistructured Data,” in *PODS*, 2001.

[23] S. Cohen and T. Brodianskiy, “Correcting Queries for XML,” *Inf. Syst.*, vol. 34, no. 8, pp. 690–710, 2009.

[24] A. Balmin, L. S. Colby, E. Curtmola, Q. Li, and F. Ozcan, “Search Driven Analysis of Heterogenous XML Data,” in *CIDR*, 2009.

[25] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, “XSEarch: A Semantic Search Engine for XML,” in *VLDB*, 2003, pp. 45–56.

[26] Z. Liu, J. Walker, and Y. Chen, “XSeek: A Semantic XML Search Engine Using Keywords,” in *VLDB*, 2007, pp. 1330–1333.

[27] Y. Li, C. Yu, and H. V. Jagadish, “Schema-Free XQuery,” in *VLDB*, 2004, pp. 72–83.

[28] Y. Xu and Y. Papakonstantinou, “Efficient Keyword Search for Smallest LCAs in XML Databases,” in *SIGMOD Conference*, 2005, pp. 537–538.

[29] S. Krishnamurthi, K. E. Gray, and P. T. Graunke, “Transformation-by-Example for XML,” in *PADL*, 2000, pp. 249–262.

[30] T. Pankowski, “A High-Level Language for Specifying XML Data Transformations,” in *ADBIS*, 2004, pp. 159–172.

[31] W. Fan and P. Bohannon, “Information preserving xml schema embedding,” *ACM Trans. Database Syst.*, vol. 33, no. 1, 2008.

[32] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks,” *CACM*, vol. 13, no. 6, pp. 377–387, 1970.

[33] Facebook Inc., “Working draft graphql [online],” https://graphql.github.io/graphql-spec/draft/, 2019, (Accessed on 07/22/2019).

APPENDIX

A.1 SQL Keywords

We access data from a relational using SQL syntax. We use the following keywords for to create our web service

1. **SELECT**

   The SELECT term in an SQL query is used to retrieve data from the tables in a database. The data used with the SELECT keywords contain the column names present in the tables in the database. The SELECT keyword must contain column names only from the tables specified in the FROM clause The column names are separated with commas and can be renamed with using AS keyword SELECT * retrieves all the columns present in the columns specified in the FROM keyword The SELECT clause can be used with the DISTINCT keyword to remove duplicate results and return unique results.

   Examples of SELECT statements -

   ```
   SELECT customer_name, customer_phone
   SELECT customer_name AS name, customer_phone AS number
   SELECT *
   SELECT DISTINCT customer_name AS name
   ```

2. **FROM**

   The FROM term in an SQL query is used to specify the tables in the database from which the data needs to be retrieved. The FROM keyword along with SELECT keyword are the two mandatory keywords in every SQL query The FROM keyword can also use a table that is in itself a query whose results are referenced as a table The FROM keyword is always followed by the SELECT keyword and the FROM keyword can be followed by other optional keywords.

Examples of FROM statements -

```
FROM customer, orders

FROM (SELECT * FROM customer, orders) AS table1
```

3. **WHERE**

   The WHERE term in an SQL query is used to specify the join conditions based on a relation between two tables. The WHERE keyword can also be used to add conditions to filter out the data with expressions such as less than, greater than, and equal to Several conditions can be added in the where clause using the AND, OR and NOT keywords.

   Examples of WHERE statements -

   ```
   WHERE table1.userID = table2.userID

   WHERE table1.userID = table2.userID AND table1.userID < 100

   WHERE table1.userID = table2.userID AND NOT table1.userID < 100
   ```

4. **ORDER BY**

   The ORDER BY keyword is basically used to sort the result set that is generated by the rest of the query. The result set can be sorted in either ascending order or descending order We can sort multiple columns and their sorting can be prioritized by the order of the column names specified. Multiple column names can be specified with a comma delimiter The result set is sorted first on the column names that are specified first and it is next sorted on the column name that is specified second and so on The column names in the ORDER BY keyword can only be the ones that are mentioned in the SELECT clause.

Examples of ORDER BY statements -

```
ORDER BY userID
ORDER BY userID, number
```

In the second statement the results would first be sorted on the basis of userID first and among the same userID, and then the number would be sorted

A.2 JDBC Drivers

JDBC is short for Java Database Connectivity driver which acts as a bridge between a Java application and a database. JDBC can be used to get metadata about the database, run queries and retrieve the query result to the application. It requires credentials to access the database which is a necessity to use our application.