

Generating Plugs and Data Sockets for Plug-and-Play Database Web Services

Arihant Jain¹, Curtis Dyreson¹^[0000–0003–0236–1515], and Sourav S Bhowmick²^[0000–0003–1957–8016]

¹ Department of Computer Science, Utah State University, Logan UT, USA
arihant25jain@gmail.com, curtis.dyreson@usu.edu
<https://www.usu.edu/cs/people/CurtisDyreson/>

² Nanyang Technological University, Singapore
assourav@ntu.edu.sg
<https://personal.ntu.edu.sg/assourav/>

Abstract. We propose a novel system for creating data plugs and sockets for plug-and-play database web services. We adopt a plug-and-play approach to couple an application to a database. In our approach 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 “socket” then the web service is generated. Our plug-and-play approach has three advantages. First, a plug is *portable*. A plug can be played on any data source to generate a web service. Second, a plug is *reliable*. The database is checked to ensure that 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 a system for plug-and-play web services and experimentally evaluate the system.

Keywords: Web services · Databases · Plug-and-play

1 Introduction

Web services are a common technology for transferring data between a web server and an application. As shown in Figure 1 a web service is a bridge between an application and a database with a coupling on both ends. The *application coupling* is between the code in the application and the web service. A web service provides (or accepts) data formatted in a specific *shape*, which is usually a *hierarchy* since the data is typically formatted in JSON or XML, and could be further transformed using GraphQL. The second coupling is between the web service and the back-end database, which we will call the *database coupling*. The database coupling maps flat, relational data in the database to the shape (hierarchy) used by the service; it is typically an object-relational mapping (ORM).

We observed that the design and construction of the application and database couplings could be improved in (at least) three ways. First, the application coupling is *rigid*. The web service constructs data to a specific shape. We will call

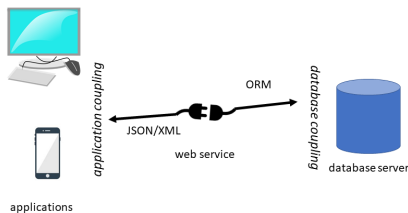


Fig. 1. Couplings in a web service

this shape a *data socket*. An application also needs data in a specific shape, which we will call a *data plug*. If the data plug does not fit the data socket then the application is unable to couple to the web service. Rigidity reduces application portability. Second, both couplings are *brittle*, changes to a data plug or data socket, *i.e.*, changes to the database, application, or web service, may break an existing coupling. Brittleness “pins” both the database *schema* (how the data is organized) and the application (which uses the schema to construct queries), limiting their evolution. Third, the application and database couplings are *static*. That is, an application can only choose among the data sockets pre-defined by the database coupling. Complicating the problem is that the application coupling and database coupling are specified using separate technologies.

This paper proposes a plug-and-play approach to web service construction to make the couplings more flexible, resilient to change, and dynamic. We call our plug-and-play web service creator AUTOREST. Suppose that a biodiversity application wants visualize a taxonomic hierarchy from data stored in a biodiversity database. There are many such databases [1] hosted by various biodiversity platforms such as Symbiota2, Specify, or Arctos, to which the application could couple. The databases have the same kinds of data but different schemas. The designers of the taxonomic hierarchy viewer application describe its data needs as a set of data plugs. AUTOREST either constructs data sockets from the database to fit each of the data plugs or describes how the construction will fail (lose information).

As an example, suppose that as part of the visualization of the taxonomic hierarchy the application consumes data about journal articles related to scientific names, grouping titles and DOIs of articles below the names. Additionally the application would like to translate the keys in the key/value pairs in the data (this translation is optional) from English to Spanish. The application designer would give the data plug specification shown in Figure 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 3.

This paper is organized as follows. The next section describes the architecture of AUTOREST. We then describe some experiments that explore two variants of AUTOREST. Related work is described in Section 4. The paper concludes in Section 5.

```
[{ "el nombre científico" : "scientific name",
  "los articulos" : [{ "titulo": "title",
                       "DOI": "DOI"}] }]
```

Fig. 2. A plug-and-play web service specification

```
[ { "el nombre científico" : "Canis lupus",
  "los articulos" : [ { "titulo" : "From the Past to the Present: Wolf Phylogeography",
                       "DOI" : "10.3389/fevo.2016.00134" },
    { "titulo" : "...",
      "DOI" : "..." },
    ... ] },
{ "el nombre científico" : "...", ... }, ... ]
```

Fig. 3. Data returned by the constructed service

2 AUTOREST Architecture

In this section we describe the architecture of AUTOREST. This section describes each step in the generating a data plug and socket.

2.1 Getting Started

AUTOREST has a GUI written in Python, though AUTOREST is primarily written in Java. The code is publicly available from github: <https://github.com/cdyreson/autorest>. AUTOREST provides an interface to connect to a database and harvest the metadata, *e.g.*, the schema, from the database.

2.2 Association Multigraph Construction

AUTOREST next creates an *association multigraph*. An edge in the graph is a foreign key relationship (it is undirected since the edge can be traversed in either direction) and a node is a table. Attributes for a table are associated with the node. We use foreign keys because they are available in the schema.

2.3 Parsing the Plug

The plug is parsed, creating an abstract syntax tree (AST). We use ANTLR to parse the plug and walk the AST to perform other actions. AUTOREST matches the plug to the association multigraph by first matching names in the plug to attributes associated to nodes in the graph. A name may match multiple nodes. For each match, AUTOREST builds the spanning tree from the leaves of the plug (the plug specifies a hierarchy) to the root using the principle of *closeness* to associate parents with children. Closeness can be described as the property that two data items are *related* if they are connected (by a path) and

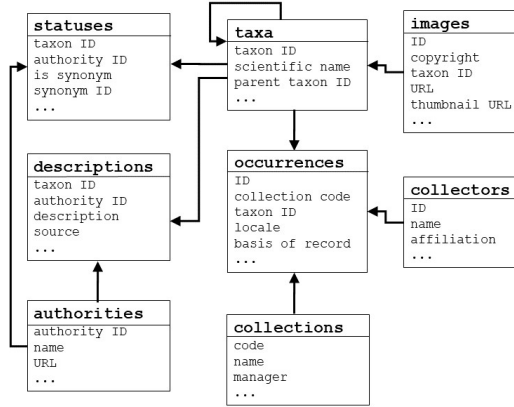


Fig. 4. Reduced schema of the Symbiota2 database

that no shorter paths that connect items of the same *type* exists [6, 16]. In the context of relational databases the *type* of a datum is the domain (an attribute in a relation) to which it belongs. This matching creates a closest spanning tree, which is traversed using an inorder walk to generate a *path* that connects the data. Since a name may match multiple nodes in the graph and multiple edges could connect a pair of nodes, there could be several paths for a plug. The next step determines the *best* path.

2.4 Finding Closest Paths

There can be several possible paths between two tables in a database. To find the closest path we use a modified breadth-first search algorithm to find all the shortest paths between the two tables. There are several cases of how queries are processed as described in the remainder of this section.

Case: Single table plug Suppose we want to create a simple web service that returns an orders key and status information from the Symbiota2 database using the plug shown in Figure 5. We process the plug as described previously. After processing, we have data for the **SELECT** and **ORDER BY** keywords. To create a 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 join conditions for the query.

For a given search query we first begin with the first column and then we find the relation to the next column. For the example query, it is **locale** and **basis of record**, respectively. Looking at the schema in Figure 4 we see that both columns are in the same table. To get the data for the **FROM** keyword, all we need is the name of the table. Our algorithm generates the SQL query shown in Figure 6. Similarly, if the query had more columns from the same table then we would only need to add the column names to the **SELECT** and **ORDER BY** clauses.

```
[ { "where": "locale", "basis": "basis of record" } ]
```

Fig. 5. Simple, single table plug

```
SELECT DISTINCT locale, 'basis of record'  
FROM occurrences  
ORDER BY locale, 'basis of record'
```

Fig. 6. SQL for the single table plug shown in Figure 5

Case: Multi-table plug Suppose we want to create a web service to find the **scientific names** of **taxa** and information about the **images** for each taxon, then we would use the plug given in Figure 7. Again we get the data for **SELECT** and **ORDER BY** from the initial processing stages of the search query. Next we need to find the relation among the different columns in the query. As described earlier we start with finding the relation for **scientific name** and the next column **URL**. These columns are from different tables. So we build the join condition between tables, **taxa** and **images** and create a query-specific path resulting in the SQL query shown in Figure 8. The reason we use left joins is if there were **taxa** that do not have any **images** then we would not get their **scientific names** in the result set.

Case: Multi hierarchy plug Suppose that the plug is as given in Figure 9. Finding paths in a hierarchical query differs from a flat query since we need to find paths between parents and children in the plug. For the example query, we find the relation between **scientific name** and **locale**, and similarly after than between **scientific name** and **URL**. The query generated for this plug is shown in Figure 10. We create the hierarchical structure from the result set after executing the query. Since the result is ordered by nodes higher in the hierarchy, the hierarchy can be constructed in a streaming fashion from the result.

Case: Multiple-path plug There could be multiple shortest paths connecting two relations, for example for the plug given in Figure 11 one path is

taxa - descriptions - authorities

while another is given below.

taxa - statuses - authorities

Such a situation is quite likely to occur in a database with several relationship types between a pair of entity types.

To enable the user to choose the best path, AUTOREST visualizes the paths. In this visualization on the full Symbiota2 schema (rather than the reduced and simplified schema used previously) there are seven paths that connect the **authorities** and **taxa** tables. The visualization enables a developer to choose

```
[ { "taxon": "scientific name", "image": "URL",
    "copyright": "copyright", "thumbnail": "thumbnail URL" } ]
```

Fig. 7. Multi-table plug

```
SELECT DISTINCT taxa.'scientific name', images.URL,
                images.copyright, images.'thumbnail URL'
FROM taxa LEFT JOIN images USING ('taxon ID')
ORDER BY taxa.'scientific name', images.URL,
         images.copyright, images.'thumbnail URL'
```

Fig. 8. Generated SQL query for plug in Figure 7

a path other than the one AUTOREST deems as best (lowest cost/most information retained).

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 (*e.g.*, using `EXPLAIN ANALYZE`) or by estimating the number of rows (*e.g.*, using `EXPLAIN`). AUTOREST shows the estimated number of rows and also presents the user with a graphical representation of the path of the join condition in the database.

Finally, to maximize completeness a user can choose to perform the *union* of alternative paths. We do not automatically detect when a union will improve the completeness since the query subsumption problem (figuring out if a query produces a subset of another query) is also NP-complete. Rather we leave it to the designer to choose to union alternatives.

2.5 Creating the Service

In our implementation, we auto generate a Python script using the Flask framework to create the web service.

3 Evaluation

In this section we describe the results of several experiments to evaluate AUTOREST. The evaluation measures the *feasibility* of plug-and-play web services. We explore two alternatives in cost estimation in an SQL query compiler while creating a web service using AUTOREST.

We performed our experiments on a desktop machine with an i7-4770 CPU with a clock speed of 3.40GHz and 16GB of DDR3 memory. The OS used is Ubuntu 18 LTS, 64-bit and the Java version used is version 11. We performed the experiments using the Postgres DBMS version 12. We used an out-of-the box

```
[ { "taxon": "scientific name", "occurrences": [ { "locale" : "locale" } ],
  "images": [ { "URL" : "URL"} ] } ]
```

Fig. 9. A multi-hierarchy plug

```
SELECT DISTINCT taxa.'scientific name', occurrences.locale, images.URL
FROM taxa LEFT JOIN occurrences USING ('taxon ID')
      LEFT JOIN images USING ('taxon ID')
ORDER BY taxa.'scientific name', occurrences.locale, images.URL
```

Fig. 10. SQL for the multi-hierarchy plug of Figure 9

version of both Postgres and Java, with no adjustments made for performance tuning, such as increasing cache memory size. The experiments used a standard relational benchmark database, TPC-H [15]. We used TPC-H rather than Symbiota2 since we wanted to experiment with different database sizes (in a later experiment).

For the first experiment the TPC-H database generator was used to generate a database 10MB in size. We manually created seven plugs based on TPC-H queries, which are in the test section of the implementation package. The experiment measures 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. Figure 12 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 13 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.

We also measured the time to produce the first result. Pagination is typically used for web services, so the time to the first result is essentially the time to produce the first page. Figure 14 shows the difference in the cost of computing the first result vs. the complete result using `EXPLAIN`.

```
[ { "scientific name": "scientific name", "editors": "editors" } ]
```

Fig. 11. A multiple path plug

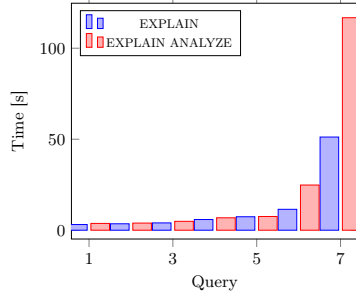


Fig. 12. Timing **EXPLAIN** vs. **EXPLAIN ANALYZE** on seven plugs of increasing complexity

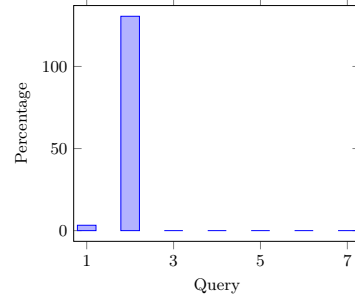


Fig. 13. Percentage difference of number of rows in **EXPLAIN** vs **EXPLAIN ANALYZE**

The previous experiments used a relatively small database, so we were interested in determining how the size of the database impacted the time taken. Figure 15 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.

4 Related Work

Related work falls into two categories, existing tools for web services creation and peer-reviewed research. We cover the tools first.

There are API-side tools to create or document a client's view of a web service, that is, a program interface and documentation *e.g.*, the Swagger User Interface editor [13]. API-side tools like GraphQL can be further applied to process the data returned by a web service, but lack the database construction of the service as described in this paper. There are also tools to create the *DBMS-side of the web service*. A canonical tool in this category is Doctrine [5]. AUTOREST combines the API-side and DBMS-side construction.

Plug-and-play web services are a technique for easing the burden of constructing a hierarchy from a database, which has been investigated previously in various ways. The problem of constructing a hierarchy from relational data is simplified by storing the hierarchical data in a relational database [11, 14] or key/value store, such as MongoDB. The main challenge addressed in this paper is how to transform (flat) relational tables to hierarchical data (JSON), which is a

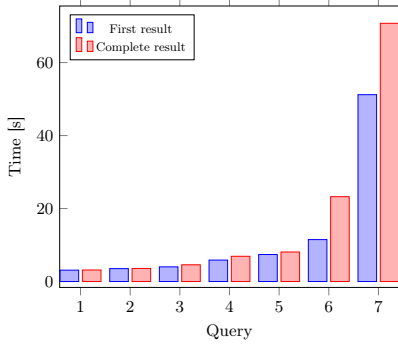


Fig. 14. Timing first result vs complete result set

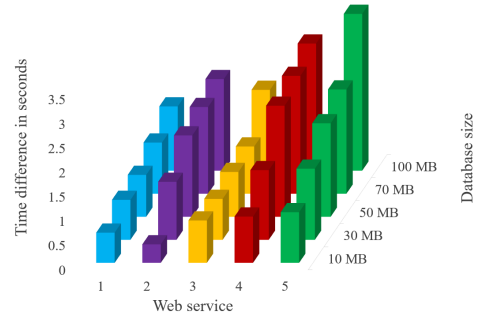


Fig. 15. Difference between EXPLAIN and EXPLAIN ANALYZE

different problem than how to transform hierarchical data to hierarchical data [6, 10, 12, 16]. Codd famously proposed transforming hierarchical data to relational data [4], which is the paradigm that has dominated much of the database literature. Web service composition [3] is another way of changing the shape, through the process of composing existing web services. Such compositions are prone to brittleness [8]. AUTOREST grows a web service rather than composing existing services; composition does not address how to create a hierarchy from tables.

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 [2]. 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 [7, 9]. Part of a TGD is a specification of the source hierarchy from which to extract the data. Specifying the source will not work for plug-and-play web services, which must be agnostic about the source.

5 Conclusion

We built a system called AUTOREST to provide plug-and-play web services. AUTOREST 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 attributes are related in an association multigraph. AUTOREST essentially eliminates the need for any prior coding knowledge to create a web service, and also enables fast web service creation. This paper describes how AUTOREST is implemented and gives an experimental evaluation.

Acknowledgements

This work was supported in part by the National Science Foundation under Award No. DBI-1759965, *Collaborative Research: ABI Development: Symbiota2: Enabling greater collaboration and flexibility for mobilizing biodiversity data*. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of NSF.

References

1. Ball-Damerow, J., Brenskelle, L., Barve, N., Soltis, P., Sierwald, P., Bieler, R., Lafrance, R., Ariño, A., Robert, G.: Research applications of primary biodiversity databases in the digital age. *PLOS ONE* **14**, e0215794 (09 2019)
2. Bhide, M., Agarwal, M., Bar-Or, A., Padmanabhan, S., Mittapalli, S., Venkatachaliah, G.: XPEDIA: XML ProcEssing for Data Integration. *PVLDB* **2**(2), 1330–1341 (2009)
3. Chan, P.P.W., Lyu, M.R.: Dynamic web service composition: A new approach in building reliable web service. In: *AINA*. pp. 20–25 (2008)
4. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. *CACM* **13**(6), 377–387 (1970)
5. Doctrine: Object relational mapper [online]. <https://www.doctrine-project.org/projects/orm.html> (2019), (Accessed on 07/23/2019)
6. Dyreson, C.E., Bhowmick, S.S.: Querying XML Data: As You Shape It. In: *ICDE*. pp. 642–653 (2012)
7. Fagin, R., Haas, L., Hernandez, M., Miller, R., Popa, L., Velegrakis, Y.: Clío: Schema Mapping Creation and Data Exchange. In: *LNCS 5600*. pp. 198–236 (2009)
8. Hu, T., Guo, M., Guo, S., Ozaki, H., Zheng, L., Ota, K., Dong, M.: Mttf of composite web services. In: *ISPA*. pp. 130–137 (2010)
9. Jiang, H., Ho, H., Popa, L., Han, W.S.: Mapping-driven XML Transformation. In: *WWW*. pp. 1063–1072 (2007)
10. Krishnamurthi, S., Gray, K.E., Graunke, P.T.: Transformation-by-Example for XML. In: *PADL*. pp. 249–262 (2000)
11. Liu, Z.H., Hammerschmidt, B., McMahon, D.: Json data management: Supporting schema-less development in rdbms. In: *SIGMOD*. pp. 1247–1258. *ACM* (2014)
12. Pankowski, T.: A High-Level Language for Specifying XML Data Transformations. In: *ADBS*. pp. 159–172 (2004)
13. Swagger.io: Swagger ui [online]. <https://swagger.io/tools/swagger-ui/> (2019), (Accessed on 07/23/2019)
14. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and Querying Ordered XML using a Relational Database System. In: *SIGMOD Conference*. pp. 204–215 (2002)
15. TPC.org: Tpc-h homepage [online]. <https://tpc.org/tpch/> (2019), (Accessed on 07/22/2019)
16. Zhang, S., Dyreson, C.E.: Symmetrically Exploiting XML. In: *WWW*. pp. 103–111 (2006)