Solution design and methodology

The solution described in this post provides a set of optimizations that solve the aforementioned challenges while reducing the amount of work that has to be performed by an LLM for generating accurate output. This work extends upon the post Generating value from enterprise data: Best practices for Text2SQL and generative AI. That post has many useful recommendations for generating high-quality SQL, and the guidelines outlined might be sufficient for your needs, depending on the inherent complexity of the database schemas.

To achieve generative accuracy for complex scenarios, the solution breaks down NL2SQL generation into a sequence of focused steps and sub-problems, narrowing the generative focus to the appropriate data domain. Using data abstractions for complex joins and data structure, this approach enables the use of smaller and more affordable LLMs for the task. This approach results in reduced prompt size and complexity for inference, reduced response latency, and improved accuracy, while enabling the use of off-the-shelf pre-trained models.

Narrowing scope to specific data domains

The solution workflow narrows down the overall schema space into the data domain targeted by the user's query. Each data domain corresponds to the set of database data structures (tables, views, and so on) that are commonly used together to answer a set of related user queries, for an application or business domain. The solution uses the data domain to construct prompt inputs for the generative LLM.

This pattern consists of the following elements:

- Mapping input queries to domains – This involves mapping each user query to the data domain that is appropriate for generating the response for NL2SQL at runtime. This mapping is similar in nature to intent classification, and enables the construction of an LLM prompt that is scoped for each input query (described next).
- Scoping data domain for focused prompt construction – This is a divide-and-conquer pattern. By focusing on the data domain of the input query, redundant information, such as schemas for other data domains in the enterprise data store, can be excluded. This might be considered as a form of prompt pruning; however, it offers more than prompt reduction alone. Reducing the prompt context to the in-focus data domain enables greater scope for few-shot learning examples, declaration of specific business rules, and more.
- Augmenting SQL DDL definitions with metadata to enhance LLM inference – This involves enhancing the LLM prompt context by augmenting the SQL DDL for the data domain with descriptions of tables, columns, and rules to be used by the LLM as guidance on its generation. This is described in more detail later in this post.
- Determine query dialect and connection information – For each data domain, the database server metadata (such as the SQL dialect and connection URI) is captured during use case onboarding and made available at runtime to be automatically included in the prompt for SQL generation and subsequent query execution. This enables scalability through decoupling the natural language query from the specific queried data source. Together, the SQL dialect and connectivity abstractions allow for the solution to be data source agnostic; data sources might be distributed within or across different clouds, or provided by different vendors. This modularity enables scalable addition of new data sources and data domains, because each is independent.

Managing identifiers for SQL generation (resource IDs)

Resolving identifiers involves extracting the named resources, as named entities, from the user's query and mapping the values to unique IDs appropriate for the target data source prior to NL2SQL generation. This can be implemented using natural language processing (NLP) or LLMs to apply named entity recognition (NER) capabilities to drive the resolution process. This optional step has the most value when there are many named resources and the lookup process is complex. For instance, in a user query such as "In what games did Isabelle Werth, Nedo Nadi, and Allyson Felix compete?" there are named resources: 'allyson felix', 'isabelle werth', and 'nedo nadi'. This step allows for rapid and precise feedback to the user when a resource can't be resolved to an identifier (for example, due to ambiguity).

This optional process of handling many or paired identifiers is included to offload the burden on LLMs for user queries with challenging sets of identifiers to be incorporated, such as those that might come in pairs (such as ID-type, ID-value), or where there are many identifiers. Rather than having the generative LLM insert each unique ID into the SQL directly, the identifiers are made available by defining a temporary data structure (such as a temporary table) and a set of corresponding insert statements. The LLM is prompted with few-shot learning examples to generate SQL for the user query by joining with the temporary data structure, rather than attempt identity injection. This results in a simpler and more consistent query pattern for cases when there are one, many, or pairs of identifiers.

Handling complex data structures: Abstracting domain data structures

This step is aimed at simplifying complex data structures into a form that can be understood by the language model without having to decipher complex inter-data relationships. Complex data structures might appear as nested tables or lists within a table column, for instance.

We can define temporary data structures (such as views and tables) that abstract complex multi-table joins, nested structures, and more. These higher-level abstractions provide simplified data structures for query generation and execution. The top-level definitions of these abstractions are included as part of the prompt context for query generation, and the full definitions are provided to the SQL execution engine, along with the generated query. The resulting queries from this process can use simple set operations (such as IN, as opposed to complex joins) that LLMs are well trained on, thereby alleviating the need for nested joins and filters over complex data structures.

Augmenting data with data definitions for prompt construction

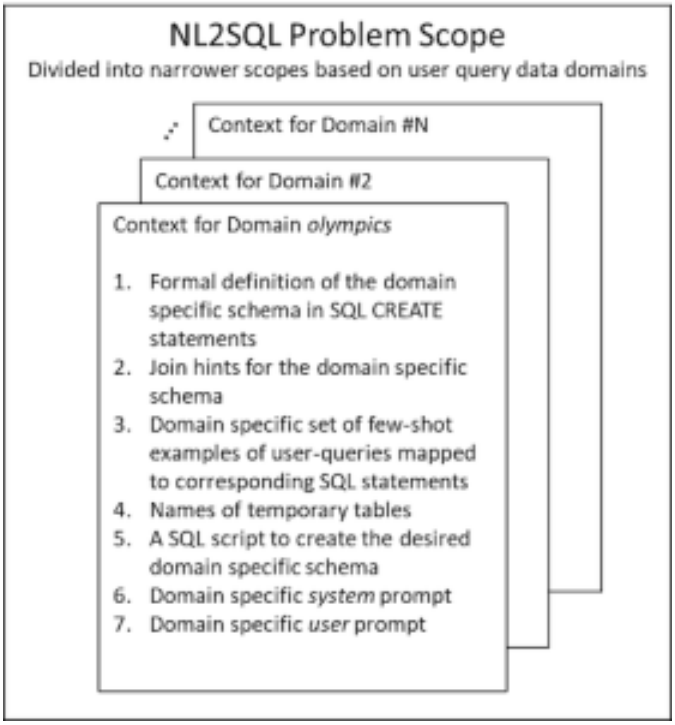
Several of the optimizations noted earlier require making some of the specifics of the data domain explicit. Fortunately, this only has to be done when schemas and use cases are onboarded or updated. The benefit is higher generative accuracy, reduced generative latency and cost, and the ability to support arbitrarily complex query requirements.

To capture the semantics of a data domain, the following elements are defined:

- The standard tables and views in data schema, along with comments to describe the tables and columns.
- Join hints for the tables and views, such as when to use outer joins.
- Data domain-specific rules, such as which columns might not appear in a final select statement.

- The set of few-shot examples of user queries and corresponding SQL statements. A good set of examples would include a wide variety of user queries for that domain.
- Definitions of the data schemas for any temporary tables and views used in the solution.
- A domain-specific system prompt that specifies the role and expertise that the LLM has, the SQL dialect, and the scope of its operation.
- A domain-specific user prompt.
- Additionally, if temporary tables or views are used for the data domain, a SQL script is required that, when executed, creates the desired temporary data structures needs to be defined. Depending on the use case, this can be a static or dynamically generated script.

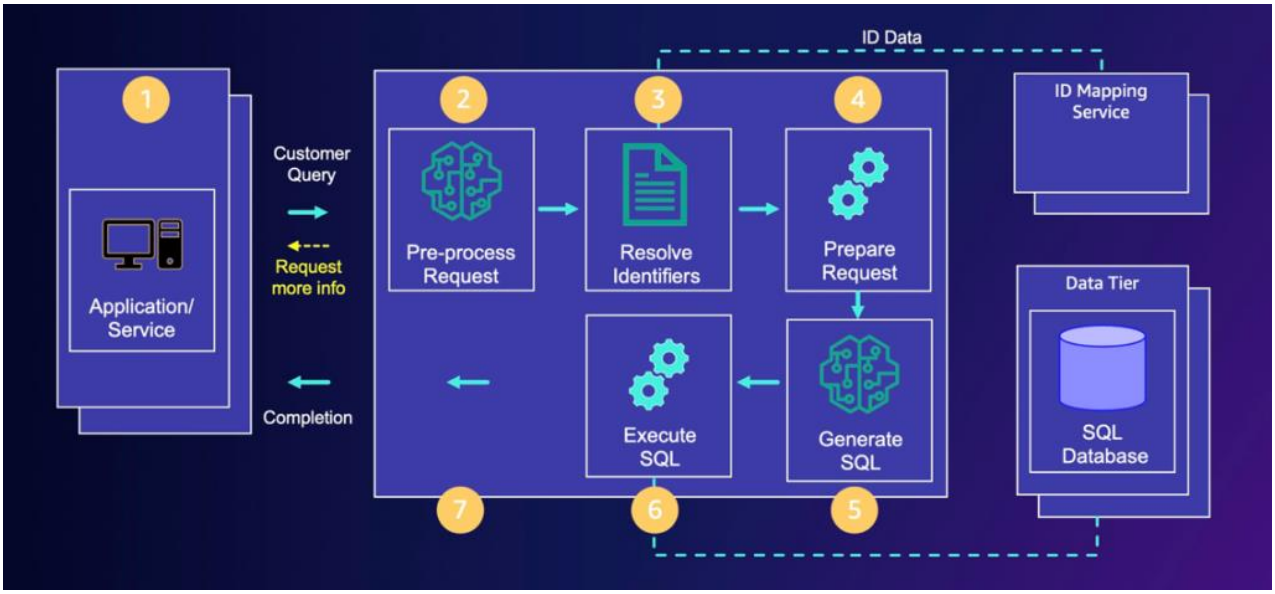
Accordingly, the prompt for generating the SQL is dynamic and constructed based on the data domain of the input question, with a set of specific definitions of data structure and rules appropriate for the input query. We refer to this set of elements as the *data domain context*. The purpose of the data domain context is to provide the necessary prompt metadata for the generative LLM. Examples of this, and the methods described in the previous sections, are included in the GitHub repository. There is one context for each data domain, as illustrated in the following figure.



Bringing it all together: The execution flow

This section describes the execution flow of the solution. An example implementation of this pattern is available in the GitHub repository. Access the repository to follow along with the code.

To illustrate the execution flow, we use an example database with data about Olympics statistics and another with the company's employee vacation schedule. We follow the execution flow for the domain regarding Olympics statistics using the user query "In what games did Isabelle Werth, Nedo Nadi, and Allyson Felix compete?" to show the inputs and outputs of the steps in the execution flow, as illustrated in the following figure.



Preprocess the request

The first step of the NL2SQL flow is to preprocess the request. The main objective of this step is to classify the user query into a domain. As explained earlier, this narrows down the scope of the problem to the appropriate data domain for SQL generation. Additionally, this step identifies and extracts the referenced named resources in the user query. These are then used to call the identity service in the next step to get the database identifiers for these named resources.

Using the earlier mentioned example, the inputs and outputs of this step are as follows:

```
user_query = "In what games did Isabelle Werth, Nedo Nadi and Allyson Felix
compete?"


pre_processed_request = request_pre_processor.run(user_query)


domain = pre_processed_request[app_consts.DOMAIN]




# Output pre_processed_request:


  {'user_query': 'In what games did Isabelle Werth, Nedo Nadi and Allyson Felix
compete?',


   'domain': 'olympics',
```

```
   'named_resources': {'allyson felix', 'isabelle werth', 'nedo nadi'} }
```

Resolve identifiers (to database IDs)

This step processes the named resources' strings extracted in the previous step and resolves them to be identifiers that can be used in database queries. As mentioned earlier, the named resources (for example, "group22", "user123", and "I") are looked up using solution-specific means, such through database lookups or an ID service.

The following code shows the execution of this step in our running example:

```
named_resources = pre_processed_request[app_consts.NAMED_RESOURCES]

if len(named_resources) > 0:

  identifiers = id_service_facade.resolve(named_resources)

  # add identifiers to the pre_processed_request object

  pre_processed_request[app_consts.IDENTIFIERS] = identifiers

else:

  pre_processed_request[app_consts.IDENTIFIERS] = []



# Output pre_processed_request:

  {'user_query': 'In what games did Isabelle Werth, Nedo Nadi and Allyson Felix
compete?',

   'domain': 'olympics',

   'named_resources': {'allyson felix', 'isabelle werth', 'nedo nadi'},

   'identifiers': [ {'id': 34551, 'role': 32, 'name': 'allyson felix'},
```

```
    {'id': 129726, 'role': 32, 'name': 'isabelle werth'},
```

```
    {'id': 84026, 'role': 32, 'name': 'nedo nadi'} ] }
```

Prepare the request

This step is pivotal in this pattern. Having obtained the domain and the named resources along with their looked-up IDs, we use the corresponding context for that domain to generate the following:

- A prompt for the LLM to generate a SQL query corresponding to the user query
- A SQL script to create the domain-specific schema

To create the prompt for the LLM, this step assembles the system prompt, the user prompt, and the received user query from the input, along with the domain-specific schema definition, including new temporary tables created as well as any join hints, and finally the few-shot examples for the domain. Other than the user query that is received as in input, other components are based on the values provided in the context for that domain.

A SQL script for creating required domain-specific temporary structures (such as views and tables) is constructed from the information in the context. The domain-specific schema in the LLM prompt, join hints, and the few-shot examples are aligned with the schema that gets generated by running this script. In our example, this step is shown in the following code. The output is a dictionary with two keys, llm_prompt and sql_preamble. The value strings for these have been clipped here; the full output can be seen in the Jupyter notebook.

```
prepared_request = request_preparer.run(pre_processed_request)



# Output prepared_request:

{'llm_prompt': 'You are a SQL expert. Given the following SQL tables definitions,
...

CREATE TABLE games (id INTEGER PRIMARY KEY, games_year INTEGER, ...);

...

<example>

question: How many gold medals has Yukio Endo won? answer: ```{"sql":
```

```
"SELECT a.id, count(m.medal_name) as "count"

FROM athletes_in_focus a INNER JOIN games_competitor gc ...

WHERE m.medal_name = 'Gold' GROUP BY a.id;" }```

</example>

...

'sql_preamble': [ 'CREATE temp TABLE athletes_in_focus (row_id INTEGER

PRIMARY KEY, id INTEGER, full_name TEXT DEFAULT NULL);',

'INSERT INTO athletes_in_focus VALUES

(1,84026,'nedo nadi'), (2,34551,'allyson felix'), (3,129726,'isabelle werth');"]}
```

Generate SQL

Now that the prompt has been prepared along with any information necessary to provide the proper context to the LLM, we provide that information to the SQL-generating LLM in this step. The goal is to have the LLM output SQL with the correct join structure, filters, and columns. See the following code:

```
llm_response = llm_service_facade.invoke(prepared_request[ 'llm_prompt' ])

generated_sql = llm_response[ 'llm_output' ]



# Output generated_sql:

{'sql': 'SELECT g.games_name, g.games_year FROM athletes_in_focus a

JOIN games_competitor gc ON gc.person_id = a.id
```

```
JOIN games g ON gc.games_id = g.id;'}
```

Execute the SQL

After the SQL query is generated by the LLM, we can send it off to the next step. At this step, the SQL preamble and the generated SQL are merged to create a complete SQL script for execution. The complete SQL script is then executed against the data store, a response is fetched, and then the response is passed back to the client or end-user. See the following code:

```
sql_script = prepared_request[ 'sql_preamble' ] + [ generated_sql[ 'sql' ] ]

database = app_consts.get_database_for_domain(domain)

results = rdbms_service_facade.execute_sql(database, sql_script)



# Output results:

{'rdbms_output': [

('games_name', 'games_year'),

('2004 Summer', 2004),

...

('2016 Summer', 2016)],
```

```
'processing_status': 'success'}
```

Solution benefits

Overall, our tests have shown several benefits, such as:

- High accuracy – This is measured by a string matching of the generated query with the target SQL query for each test case. In our tests, we observed over 95% accuracy for 100 queries, spanning three data domains.

- High consistency – This is measured in terms of the same SQL generated being generated across multiple runs. We observed over 95% consistency for 100 queries, spanning three data domains. With the test configuration, the queries were accurate most of the time; a small number occasionally produced inconsistent results.
- Low cost and latency – The approach supports the use of small, low-cost, low-latency LLMs. We observed SQL generation in the 1–3 second range using models Meta's Code Llama 13B and Anthropic's Claude Haiku 3.
- Scalability – The methods that we employed in terms of data abstractions facilitate scaling independent of the number of entities or identifiers in the data for a given use case. For instance, in our tests consisting of a list of 200 different named resources per row of a table, and over 10,000 such rows, we measured a latency range of 2–5 seconds for SQL generation and 3.5–4.0 seconds for SQL execution.
- Solving complexity – Using the data abstractions for simplifying complexity enabled the accurate generation of arbitrarily complex enterprise queries, which almost certainly would not be possible otherwise.

We attribute the success of the solution with these excellent but lightweight models (compared to a Meta Llama 70B variant or Anthropic's Claude Sonnet) to the points noted earlier, with the reduced LLM task complexity being the driving force. The implementation code demonstrates how this is achieved. Overall, by using the optimizations outlined in this post, natural language SQL generation for enterprise data is much more feasible than would be otherwise.

AWS solution architecture

In this section, we illustrate how you might implement the architecture on AWS. The end-user sends their natural language queries to the NL2SQL solution using a REST API. Amazon API Gateway is used to provision the REST API, which can be secured by Amazon Cognito. The API is linked to an AWS Lambda function, which implements and orchestrates the processing steps described earlier using a programming language of the user's choice (such as Python) in a serverless manner. In this example implementation, where Amazon Bedrock is noted, the solution uses Anthropic's Claude Haiku 3.

Briefly, the processing steps are as follows:

1. Determine the domain by invoking an LLM on Amazon Bedrock for classification.
2. Invoke Amazon Bedrock to extract relevant named resources from the request.
3. After the named resources are determined, this step calls a service (the Identity Service) that returns identifier specifics relevant to the named resources for the task at hand. The Identity Service is logically a key/value lookup service, which might support for multiple domains.
4. This step runs on Lambda to create the LLM prompt to generate the SQL, and to define temporary SQL structures that will be executed by the SQL engine along with the SQL generated by the LLM (in the next step).
5. Given the prepared prompt, this step invokes an LLM running on Amazon Bedrock to generate the SQL statements that correspond to the input natural language query.
6. This step executes the generated SQL query against the target database. In our example implementation, we used an SQLite database for illustration purposes, but you could use another database server.

The final result is obtained by running the preceding pipeline on Lambda. When the workflow is complete, the result is provided as a response to the REST API request.

The following diagram illustrates the solution architecture.