DS4300

Eddy Liu, Qixiang Jiang

# Exploring Query Languages Through Translation

**Team Members:** Eddy Liu, Qixiang Jiang

**Abstract:**

Machine translation of query languages is powerful. Translations can enhance comprehensibility and enable collaboration by eliminating experts’ need to learn the syntactical nuances of each language. Because query languages are rigorously defined, pir team hypothesised that a “rules-based” algorithm can replicate the outputs of two sufficiently similar query languages. Specifically, our database application seeks to implement a subsect of the translation from SQL to Mongo Query Language (MQL). In doing so, we sought to explore the syntactical nuances of both query languages and the inherent limitations of query-language-to-query-language translation.

For the sake of future extensibility, our team decided to take a two-step approach to the SQL-to-MQL translation.

1. The first step was to parse a given SQL query and convert the string into a Python dictionary object. This dictionary would follow prefix notation such that the keys in the dict are operators and the values in the dict are the operands.
   1. The dictionary object provides functionality and readability as the dictionary is able to effectively encode hierarchical and nested queries.
   2. The dictionary also allows recursive function calls—enabling abstraction and code reuse.
   3. The prefix notation enhances translation efforts in that translation to Mongo would merely constitute re-mapping the key to another value.
2. The second step was to extract aggregations and reconstitute the dictionary object into a string using syntax rules.
   1. For this step, we defined inner classes like “avg” and “sum” to explicitly help re-aggregate the dictionary query into a MQL representation of the query. We specifically chose to parse aggregations by creating a Mongo pipeline to help consolidate code.

Our two-step approach to translation treats the Python dictionary as an “interlingua” for translation. The interlingua encodes relevant information about the query such that sub-attributes of the query are readily accessible.

The paper then discussed the potential advantages and drawbacks to prioritising accuracy in the rules-based approach translation. Specifically, our team highlights the benefits of using such a translation software for both employees and organisations. The paper also discusses potential scalability issues with our current rules-based approach.

In the conclusion section, the paper emphasises that translation is not a substitute for API Monoliths or Microservices at an enterprise level—but rather a tool to enhance existing infrastructure. We hypothesise that if translation to an interlingua is co-developed with APIs, that procedure may reduce the strain of creating and maintaining a translation algorithm. We also assert that ChatGPT and its derivatives may be a useful alternative to translation if strict functionality is not as high of a priority. We also discuss implications for overhauling legacy systems—such as systematically replacing SQL.

**Introduction**

Language is important. Language shapes human communication and understanding of abstract concepts—which makes tools like Google Translate all the more useful. In general, Machine Translation (MT), like Google Translate, is classified as an “AI-hard” problem ([Koehn](https://people.csail.mit.edu/koehn/publications/challenges2004.pdf) 2004). This is because MT is forced to encode the quirks of natural language—including ambiguity, nuance, and sublime context of human language.

However, query languages are notably distinct from natural languages in that there is no ambiguity. Each query is formulaic, deterministic. Each term conveys a specific meaning to the underlying database. Further, for querying against a dataset, there is an objectively correct “answer” to each query. This formulaic determinism implies translations of query languages can be algorithmically generated using a rule-based approach ([machinetranslate](https://machinetranslate.org/rule-based-machine-translation).org).

As of February 2022, Google had implemented a feature to its cloud services that enables interactive SQL translation between different SQL dialects ([Google BigQuery](https://cloud.google.com/bigquery/docs/interactive-sql-translator)). While this initiative—to the best of my knowledge—is paywalled behind Google Cloud and we were unable to use the service, documentation about query-mapping serves as a foundational proof-of-concept that a rule-based approach to query translation is viable.

That being said, instead of recreating the Google Cloud translation software by re-mapping keywords for similarly structured SQL dialects, we instead opted to tackle translation from SQL to MongoDB’s Query Language, MQL. We specifically chose SQL and Mongo due to the prevalence of both languages in the industry. Professor Rachlin always says “Everyone knows SQL”—in the context of interpersonal differentiation. This implies a wide range of specialists would be familiar with SQL—or similar relational databases. MongoDB, a document store database, is the most popular NoSQL database.

Because the two databases figuratively and associatively represent two fundamentally distinct database archetypes, our team sought to build on—instead of replicate—Google’s SQL translation and explore the functional gap between the two query languages.

**Methodology**

Motivated by the versatile application of SQL queries on PySpark DataFrames and encouraged by Professor Rachlin, our project aims to delve into cross-query language translation. We focus specifically on understanding the relationship between SQL and MongoDB querying languages by developing a translator capable of converting SQL queries into MongoDB queries. This exploration is underpinned by the notion that despite the distinct paradigms underlying relational and NoSQL databases, a systematic translation between their query languages can facilitate seamless data operations across both environments.

**Specific Steps**

* Operator Syntax Mapping:

Initiate by creating a Pandas DataFrame to store the syntax mapping of operators between SQL and MongoDB. This DataFrame acts as a reference guide to convert SQL operators to their MongoDB counterparts.

For example:

"\_p1 OR \_p2" in SQL would map to "{$or: {\_p1, \_p2}}" in MQL

* SQL String-Parser Function:

Develop a custom SQL string-parser function that translates an SQL query into a dictionary representation. This parser will:

1. Prioritise parsing of parentheses and curly brackets to maintain the logical grouping of conditions.
2. Raise an exception in cases of mismatched brackets to ensure query integrity.
3. Store each query element in a potentially recursive dictionary format, facilitating nested queries. An example structure could be:

{"SELECT":"SUM(\_p1)",

"FROM":"db",

"WHERE":{ # conditions

"OR": ["x > 0", "y <= 1"]}

* Create a Mongo\_Reconstructor:

Design a Mongo\_Reconstructor that takes the dictionary representation from the SQL parser and reconstructs it into a MongoDB query format. This includes converting conditions and aggregations into their respective MongoDB syntax, such as:

{"$or": [{"x": {"$gt": 0}},

{"y": {"$lte": 1}}]}

* Testing for Correctness:

Verify the correctness of the translation tool by testing it with real data scenarios. This involves:

1. Downloading a CSV file and importing the data into both SQL and MongoDB databases.
2. Performing aggregation functions and queries on both platforms to compare outcomes, ensuring that translated MongoDB queries yield equivalent results to their SQL counterparts.

**Analysis / Discussion**

In the limited scope of our focus on aggregation, the algorithmic translation functions as intended. In terms of implementation, a rigorous conversion from SQL query to Python dictionary would be considerably more difficult to implement.

Looking forward, because we used human-defined algorithms, our project did not attempt to cover, aliasing or joins. According to [Josep Ferrer](https://www.kdnuggets.com/the-essential-guide-to-sql-execution-order), a writer for KDnuggets, the execution order for aliases directly affects query output—which makes prioritising consistent results across databases considerably more difficult.

While our project focused on an algorithmic SQL-to-MQL approach to machine translation, from our experience we believe the inverse—a MQL-to-SQL algorithm—would be harder to implement. For one, MQL is dynamic in its lack of schema. This means, even if a translator was able to redefine the essence of the MQL query in SQL, this translation may not necessarily be runnable without significant alterations to the database.

Even if we would overcome the fixed schema through on-the-go table creation and/or creating limited-purpose columns that store a significant number of null values, Mongo also provides distinct query functionality that is not native to SQL. MQL aggregation offers enhanced flexibility through aggregation and unwinding schemes. MQL also offers unique geospatial functionality like {$nearSphere}, for which there is no comparable built-in SQL query. While there is a work around for {$nearSphere} specifically—StackOverflow members offered solutions by calculating Euclidean and/or Polar Distance ([StackOverflow 2234204](https://stackoverflow.com/questions/2234204/find-nearest-latitude-longitude-with-an-sql-query))—this solution raises another issue of scalability. How should a corporation commit to encoding distinct functionality into a translation software?

One more aspect to consider when translating between similar algorithms? One of the key aspects is lexical density—amount of information sorted in the query. This is pertinent to encoding the information into the dictionary object as well as how to decode it. Google’s BigQuery does not seem that involved with this question in that the syntaxes and underlying assumptions build on SQL.

That being said, we would also like to bring to light the potential upsides of a query translation software. For individuals, a translation software enhances the scope of ad hoc queries as they would no longer be limited by the confines / framework of a specific database. It would also help eliminate the inherent limitations to only knowing a few query languages.

We hypothesise that, for corporations, translation functions as a valuable tool to assist onboarding by decreasing requisite training for proprietary/novel query languages, to improve inter-team collaboration and communication by demystifying “foreign” query languages, and to desegregate thought-leaders by quickly and conveniently sharing knowledge.

For database adoption, the ease of the query languages is an important aspect of success. For example, CouchBase functions as a document store, like MongoDB, but includes built-in SQL query options. However, translations offer lower barriers of entry for trying less mainstream query languages in pilot programs.

Our algorithm implementation also requires two classes per query language: one to encode the query (along with its syntactical nuances) as the “interlingua” and a second to reconstruct the query from the “interlingua”. The algorithm also runs into issues when one language needs/provides ‘niche’ query infrastructure that the target language does not natively support.

While it may be possible to replicate the essence query in a non-native environment, an organisation would be difficult to maintain the translation software. We hypothesise three potential worse-case scenarios of an corporate-run *functionality-first algorithmic* approach to database translation. First, the interlingua either becomes a bottleneck due to insufficient parsing of lexical density or it becomes indecipherable to the average user. Second, we hypothesise that the translation software itself becomes an unwieldy monolith with rules that make it very difficult to maintain. Alternatively, an enterprise may choose to limit functionality of the translator to only what can be reasonably recreated on every language node in the translation software. This poses the risk that as more languages are added, the functionality to the end user diminishes considerably.

In terms of functionality of such an algorithm there are quite a few drawbacks for use in a production environment. First, human readability suffers for long queries and stored procedures. Additionally, because queries in a language are extracted via interlingua, if the algorithm is not comprehensive enough, the resultant query will not be optimal runtime wise—meaning experts in the target language will still be needed. Thirdly, there are inherent limitations translating from a high-specification language to a low-specification one.

Why translate when companies could build API infrastructure like monoliths or microservices?

**Conclusion**

*Translation does not necessarily seek to replace* API Monoliths or Microservices at an enterprise level; corporations will still need API infrastructure to function effectively. We hypothesise that if translation to an interlingua is co-developed with APIs, that procedure may reduce the strain of creating and maintaining a translation algorithm. We also assert that ChatGPT and its derivatives may be a useful alternative to translation if strict functionality is not as high of a priority.

A functionality-first approach to translation may be especially useful as an early stage in overhauling legacy SQL systems to novel databases. While the direct translation may not be optimised for the target language in question, the translation serves as an framework to build off of. From there, the organisation could utilise ChatGPT-esque tools and database-specific specialists to parse and redefine queries/stored procedures—in effect replacing the functionality-first approach.

**Works Cited:**

**Reference:**

* [SQL to Aggregation Mapping Chart](https://www.mongodb.com/docs/manual/reference/sql-aggregation-comparison/) (Click the title)
* [What is SQL Aggregation?](https://www.integrate.io/glossary/what-is-sql-aggregation/)