Mallet: SQL Dialect Translation with LLM Rule Generation

Amadou Latyr Ngom
MIT CSAIL
United States of America
ngom@mit.edu

Tim Kraska
MIT CSAIL, AWS
United States of America
kraska@mit.edu

ABSTRACT

Translating between the SQL dialects of different systems is important for migration and federated query processing. Existing approaches rely on hand-crafted translation rules, which tend to be incomplete and hard to maintain, especially as the number of dialects to translate increases. Thus, dialect translation remains a largely unsolved problem.

To address this issue, we introduce Mallet, a system that leverages Large Language Models (LLMs) to automate the generation of SQL-to-SQL translation rules, namely schema conversion, automated UDF generation, extension selection, and expression composition. Once the rules are generated, they are infinitely reusable on new workloads without putting the LLM on the critical path of query execution. Mallet enhances the accuracy of the LLMs by (1) performing retrieval augmented generation (RAG) over system documentation and human expertise, (2) subjecting the rules to empirical validation using the actual SQL systems to detect hallucinations, and (3) automatically creating accurate few-shot learning instances. Contributors, without knowing the system's code, can improve Mallet by providing natural-language expertise for RAG.

ACM Reference Format:
Amadou Latyr Ngom and Tim Kraska. 2024. Mallet: SQL Dialect Translation with LLM Rule Generation. In *Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '24), June 14, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663742.3663973

1 INTRODUCTION

The database management landscape has grown increasingly complex over the last few decades. This complexity arises not just from the number of Database Management Systems (DBMSs), each specialized for various kinds of workloads, but also from the profusion of incompatible SQL dialects. Such incompatibility severely hampers the ease of migration and federation, locking users with their initial DMBS, no matter how suboptimal in cost and performance and lacking in a particular feature. Translating between arbitrary dialects would not only facilitate migrations but also enable intelligent federated systems that can route portions of the workload to the systems that can handle them at better cost or performance [6]. Thus, it would constitute a significant boon to the DBMS landscape.

Dialect translation is a difficult problem. Consider, for example, the translation process shown in Fig. 1. We are trying to migrate a MySQL workload (Fig. 1a) to PostgreSQL (Fig. 1b). First, we must detect that certain functionalities (e.g., spatial functionalities) are not supported by PostgreSQL without extensions (e.g., PostGIS). Second, the MySQL types must be matched to PostgreSQL types, which may have different names or representations (e.g., `mediumblob` to `bytea`). Third, we must find a common intermediate representation for each type to dump and load the data (e.g., well-known text (WKT) for spatial types). Fourth, some MySQL functions may require generating PostgreSQL UDFs that mimic them. Fifth, other MySQL functions may require combining PostgreSQL functions (e.g., some date/time functions). Finally (not shown), user-defined logic (e.g., UDFs or stored procedures) written in MySQL must be translated to PostgreSQL.

```
--- MySQL: Schema
CREATE TABLE user (
    uid BIGINT PRIMARY KEY,
    ...
    preview_img MEDIUMBLOB,
    last_login_fail DATETIME,
    settings JSON,
    location POINT
);
--- MySQL: get some settings.
SELECT JSON_EXTRACT(settings, <path>) FROM user;
--- MySQL: check security issues.
SELECT TIMESTAMPDIFF(MINUTE, last_login_fail, NOW()) FROM user;
--- MySQL: get location info.
SELECT ST_ASTEXT(location) FROM user;
```

(a) MySQL Schema and Queries.

```
-- Postgres: Install Extensions
CREATE EXTENSION postgis;
-- Postgres: Schema
CREATE TABLE user (
    uid BIGINT PRIMARY KEY,
    ...
    preview_img BYTEA,
    last_login_fail TIMESTAMP,
    settings JSON,
    location GEOMETRY(POINT, 4326)
);
--- MySQL: Dumping to CSV
SELECT HEX(preview_img), ST_ASTEXT(location), ...;
--- Postgres: Loading from CSV
INSERT INTO user SELECT DECODE(preview_img, 'HEX'), ST_GEOMFROMTEXT(location) ...;

--- Postgres: Generate UDF
--- NOTE: This is necessary for a general handling of JSONPaths.
CREATE FUNCTION EXTRACT_UDF(doc JSON, path TEXT) RETURNS JSON AS $$
import json
from jsonpath_ng import jsonpath, parse
<equivalent python code>
$$ LANGUAGE plpython3u;

--- Postgres: get some settings.
SELECT EXTRACT_UDF(settings, <path>) FROM user;
--- Postgres: check security issues.
SELECT FLOOR((EXTRACT EPOCH FROM AGE(NOW(), last_login_fail))/60) FROM user;
--- Postgres: get location info.
SELECT ST_ASTEXT(location) FROM user;
```

(b) Translation Process.

Figure 1: Example translation from MySQL to PostgreSQL.

The problem worsens when we consider that the steps above must be repeated for several pairs of systems for which to perform translation (i.e., directly between two dialects or through an intermediate dialect). Though several tools exist to make migrations easier, they all use carefully hand-written rules, which is a tedious approach. They may work well for a specific set of systems [1], but they do not scale well with the number of systems, which explains why all the tools that aim to support a larger number of systems fail on basic queries for some pairs of systems [4, 5, 8, 12].

aiDM '24, June 14, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0680-6/24/06.
https://doi.org/10.1145/3663742.3663973

As an alternative to hand-written rules, we consider using LLMs, which have shown promise in data management tasks ([11, 13, 14]). While one could directly attempt per-query translation using LLMs (e.g., prompt it with "Rewrite <query> from MySQL to PostgreSQL"), such an approach would be fraught with limitations. First, speed and cost-wise, this approach is needlessly slow and expansive, as it requires an LLM invocation for every single new query. Second, an individual query may contain an arbitrarily large number of system-specific functionalities; an LLM attempting to translate all of them simultaneously is bound to hallucinate or terminate before completion due to output limits. Finally, for correctness, an expert has to verify every translation, making this approach unscalable.

In this paper, we propose a more scalable solution than hand-written systems and more accurate, fast, verifiable, and cheap than per-query translation. We note that in Fig. 1, each fine-grained *rule* (e.g., translating a single type or a single function) is relatively simple; common sense and expertise in both systems are enough for a human to hand-write a given rule. Our core idea is to automatically generate – rather than hand-write – these translation rules.

We introduce our experimental prototype called Mallet, a system that uses LLMs to generate a comprehensive set of fine-grained transformation rules for each functionality in a DBMS. As each rule is simpler than a whole query, this approach is more accurate than per-query LLM translation. Further, because these rules can then be infinitely reused across queries once generated, tested, and verified, Mallet is also much faster, cost-effective, and verifiable than coarse-grained translation without suffering from the lack of scalability of hand-written systems.

For greater accuracy, Mallet further uses RAG [7] not only over system documentation but also over natural-language human expertise that allows contributors to improve the system without code. In addition, since the LLM can automatically generate sample data, Mallet can use the SQL systems to automatically test rules, catch hallucinations, and generate accurate few-shot learning examples [2] of any given functionality. Our preliminary evaluation substantiates the practical utility of Mallet for dialect translation.

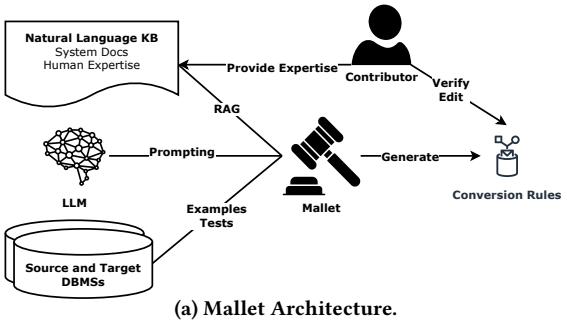
In summary, we make the following contributions:

- We introduce rule generation, a technique that uses LLMs to generate fine-grained dialect translation rules. It is more scalable than hand-written rules and more accurate, fast, and cheap than per-query LLM translation.
- We perform RAG over system documentation and over expertise from contributors who can improve the system without code.
- We further enhance the rule generation process by using the LLMs in conjunction with the DBMSs to automatically test rules and catch hallucinations and to enable few-shot learning.
- We develop an initial rule generation prototype called Mallet and show it to be superior to existing hand-written systems and per-query LLM translation.

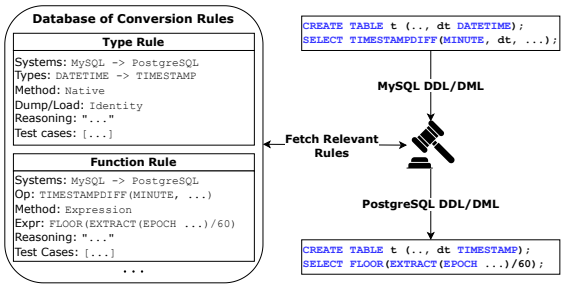
2 MALLET

2.1 Overview

Fig. 2 provides an overview of Mallet. Fig. 2a shows the system's architecture. The goal of Mallet is to use an LLM enhanced through RAG, few-shot learning, and automated testing for hallucination



(a) Mallet Architecture.



(b) Mallet at Runtime. Rules are retrieved without involving the LLM.

Figure 2: Mallet Overview

detection to generate a database of fine-grained translation rules that determine how to translate a workload from one DBMS to another (i.e., the *source* and *target*). Contributors to Mallet can improve the system by either providing English expertise or editing the rule database when automated approaches fail. Once generated, the rules are applicable without involving the LLM again. Fig. 2b shows example MySQL queries being translated to PostgreSQL using only the database of translation rules, making the process significantly faster than placing the LLM on the critical path. We now delve into the components shown in Fig. 2a.

2.2 RAG

RAG I: System Documentation Even though the LLM was trained with the documentation of a DBMS, some functionalities have subtleties that the LLM will only pay attention to if included in its context. Therefore, we include relevant documentation snippets to avoid excessively relying on expertise for these subtleties. Since documentation is highly structured, we primarily rely on keyword search, which often returns the desired snippet. As a fallback when such a search fails, we use vector embedding similarity search.

RAG II: Contributor Expertise When documentation and common sense alone are insufficient, or when the LLM makes systematic errors, it is necessary to provide expertise to help the translation process. This allows contributors to improve the system without knowing any of the system's code, which is one of the essential benefits of LLM-based programming. Most often, the expertise is general and applicable across many systems (e.g., "Use WKT to dump spatial data types"), but sometimes, it has to be specific (e.g., "Use <specific functions> to convert <specific oracle feature>").

Because expertise snippets are crucial for correctness, we rely on exact searches to include them within prompts, or else they might be

wasted by approximate searches. Currently, a contributor has to provide a regex to match their targeted rule (e.g., `.*mysql_postgres.*` for conversions between MySQL and PostgreSQL).

2.3 Using the DBMSs

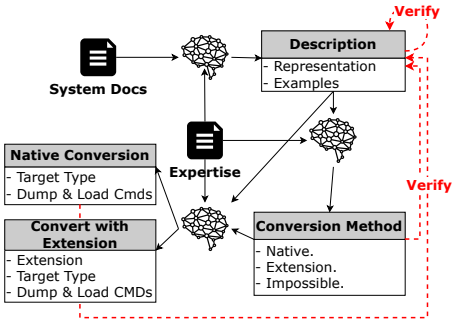
Hallucination Detection Since we have the correct implementation of any given functionality in the source system, and since an LLM can generate sample data (alternatively, application data can be sampled), it is possible to produce correct input/output values for any functionality in the source system. Then, by applying the rules, we can obtain the resulting values in the target system. Whenever the values are not equivalent, the rule is incorrect; it means that the LLM produced a hallucination. Whenever such a hallucination is detected, we iteratively re-prompt the LLM in two ways: (a) Include previous errors in the prompt to allow it to fix them or (b) re-prompt from scratch. (b) is necessary because there are cases where the LLM is stuck making small tweaks to a fundamentally incorrect rule. In such cases, restarting from scratch is better.

Few-shot learning Previous work has shown that LLM performance can be increased by providing example solutions to a task before generalizing them to a new setting. This is called *few-shot learning*. The same method used for testing can generate example input/output pairs for the prompts. The one additional complication is that prompts are textual, whereas some types have an opaque (e.g., binary) representation. In such cases, we derive standard string representations by asking the LLM to generate a rule for representing each type as a string (e.g., hex for blobs, WKT for spatial types).

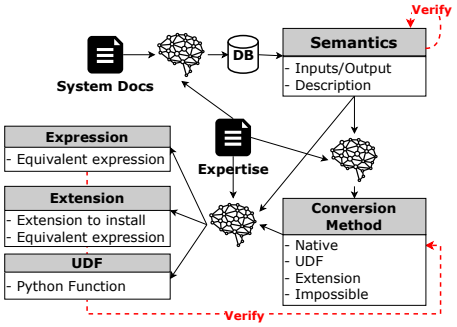
2.4 Conversion Rules

Schema Rules Fig. 3a shows the type conversion rule generation process. The process starts with RAG over documentation and expertise to fetch relevant information. The first prompt asks the LLM to generate a description and representation of the type, and varied example values. These values are checked against the system for formatting and typing correctness. This output (together with the RAG) is passed onto another prompt to ask the LLM for the best conversion method for the given type. We ask the LLM to select one of three methods. (1) `Native`: a conversion using a native type, (2) `Extension(name)`: a conversion with an extension which the LLM also specifies, and (3) `Impossible`: it is impossible to convert the type. This last option is only relevant for very unusual types when the target system cannot be extended to support them. For example, converting the `HLLSKETCH` type from Redshift to MySQL is currently impossible. When a conversion is impossible, we can still use federation (see Section 1) to adaptively move the translatable portions of the workload.

Functionality Rules. Fig. 3b shows the functionality conversion rule generation process. Mallet starts by using RAG, the LLM, and the underlying DBMS to produce input/output pairs, along with documentation emphasizing corner cases and subtleties that must be explicitly handled. Then, it asks the LLM for a conversion method, as follows. (1) `Native`: compose existing expressions, (2) `UDF`: generate a new UDF, (3) `Extension(name)`: install an extension, and (4) `Impossible`: the conversion is impossible. Based on the method, we give specific instructions to the LLM to generate the rule, then



(a) Type Conversion.



(b) Functionality Conversion.

Figure 3: Conversion Processes. The arrows from documentation and expertise indicate RAG. Outputs of previous LLM or DBMS calls are fed as context or as examples for the next steps in the process. Finally, verification steps (red arrows) use the LLM or the DBMS to test intermediate results or final rules and retry if necessary.

run our hallucination detection mechanism and retry if one is detected. If, after enough retries, a valid rule is not produced, we return `Impossible`, which either indicates that expertise is lacking or that the translation is infeasible by the LLM.

Note that we currently only generate Python UDFs since we focus on feasibility rather than performance. See Section 4 for a discussion on alternative approaches and performance in particular.

3 PRELIMINARY RESULTS

We present preliminary results showing the effectiveness of Mallet. Our prototype takes two DBMSs, a schema, and a set of queries and performs the migration. It is built on top of SQLglot [8], an SQL parser and translator, with an incomplete set of hand-written rules we aim to complete. To generate rules, we use GPT-4 [10] with temperature set to 0 for repeatability. For RAG, we use ChromaDB [3] with OpenAI's `text-embedding-ada-002` for embeddings [9].

We compare four approaches. The first is naively prompting GPT-4 with a whole query, given the schema as context. The second is SQLGlot without Mallet's additional transformations. The third and fourth are JOOQ and SQLines, commercial migration tools advertised as able to translate between any SQL dialects. For the latter two, since we rely on their online interfaces to manually translate queries, we cannot measure accurate translation times.

System	Overhead	Failures	Summary of Reasons
Mallet	**0.026s**	**0**	-
Naive GPT	16.335s	5	Syntax, output limits, divisions, hallucinations
SQLGlot	**0.004s**	5	Minor AST bugs, type casts, rollup.
JOOQ	-	13	Date operations
SQLines	-	14	Date operations, rollup

Table 1: Translating TPC-DS from MySQL to PostgreSQL.

System	Overhead	Failures	Summary of Reasons
Mallet	**0.023s**	**0**	-
Naive GPT	7.982s	Varies	JSonPaths (stochasitc)
SQLGlot	-	5	Unhandled ops
JOOQ	-	5	Unhandled ops
SQLines	-	5	Unhandled ops

Table 2: Translating STPCH from MySQL to PostgreSQL

We evaluate with two workloads, translating them from MySQL to PostgreSQL. The first is TPC-DS. It contains few special functions but helps us see the significant performance benefits of rule generation over per-query translation. The second workload is a modified version of the first five queries of TPC-H, which we call STPC-H (specialized TPC-H). STPC-H adds more types (e.g., JSON, binary, spatial) and uses more special operations in predicates and selections. Thus, each system should have its own STPC-H. The benchmark is still under construction; we seek to expand it with a comprehensive coverage of the special features of each system.

TPC-DS. The results for TPC-DS are shown in Table 1. Only Mallet can fully translate the queries. Applying rules takes 26ms; despite being unoptimized Python code, it is three orders of magnitude faster than Naive GPT4 (16s). Further, Naive GPT results in failures related to syntax errors, output limits, division semantics (floating point vs integer division), and hallucination that inserted a spurious group-by in a query. SQLglot fails due mainly to minor syntactic bugs unrelated to special functionalities. As TPC-DS contains very few special functionalities, SQLglot can translate them. However, at the time of this writing, JOOQ incorrectly translates some date operations to PostgreSQL. SQLines does not translate them at all.

STPC-H. The results of STPC-H are shown in Section 3. Hand-written approaches fail on all queries because they do not handle the special operations. Naive GPT randomly gets JSONPath operations wrong, meaning the same operations are correctly handled for some queries but not others, so we report varying failures. Again, we observed significant differences in translation speed (23ms vs 7.98s).

4 FUTURE DIRECTIONS

Intelligent Federation. Intelligent federation is the most important next step, as it allows practical deployment with cost and performance benefits even when we cannot translate every feature. Mallet can be used with BRAD [6] to optimize data placement and query routing across many systems with incompatible dialects.

Performance. So far, we have focused on the feasibility and correctness of LLM-generated rules, but their performance is also important. We intend to generate multiple alternative ways of translating the same functionality, using either static microbenchmarks or multi-armed bandit (MAB), to pick the best-performing ones.

Simulation. Hyper-Q [1] showed that even when a functionality cannot be supported in the target system, it is still possible to *simulate* it in an external proxy that interacts with the system. We will investigate LLM-generated simulations to extend our capabilities.

Procedure Translation. While we focused on migrating data and queries, many applications use custom procedures. To avoid relying on federation, we must be able to translate them. This problem is more complex than query translation as it involves arbitrary imperative code. However, it may be solvable for a broad class of applications that only use simple procedures.

LLM Fine-Tuning. We have focused on using general-purpose LLMs enhanced with RAG and few-shot learning. LLM fine-tuning is the other alternative to solving complex tasks. Two questions will need to be investigated. (1) Do LLMs fine-tuned for coding in general, and SQL generation in particular perform better than general purpose ones. (2) Using existing migrations as training data, can we effectively fine-tune an LLM specifically for SQL conversion?

Confidentiality. Our approach is more privacy-compliant than per-query LLM translation, as it can avoid sending confidential user queries to the cloud. However, to improve the system (e.g., fixing buggy or inefficient rules), users currently have to disclose the specific functionalities they use. Sharing improvements to individual functionalities in a confidential manner is an unaddressed problem.

5 CONCLUSION

We presented Mallet, the first automated SQL-to-SQL translation tool using LLMs for rule generation. By carefully dividing functionality translation into simple steps, adaptively augmenting prompts with system documentation and human expertise, automatically generating few-shot learning cases using the SQL systems themselves, and iteratively verifying results on LLM-generated test cases, Mallet aims to produce a complex and comprehensive set of rules to translate workloads between any systems without placing the LLM on the critical path. This makes it much more scalable than hand-written approaches while preserving speed and accuracy.

ACKNOWLEDGEMENTS

This research is supported by Amazon, Google, and Intel as part of the MIT Data Systems and AI Lab (DSAIL) at MIT and NSF IIS 1900933. This research was also sponsored by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] L. Antova, D. Bryant, T. Cao, M. Duller, M. A. Soliman, and F. M. Waas. Rapid adoption of cloud data warehouse technology using datometry hyper-q. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 825–839, New York, NY, USA, 2018. Association for Computing Machinery.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[3] Chroma. Chroma DB. https://www.trychroma.com/.

[4] Datometry. DATOMETRY OPENDB FOR POSTGRESQL. https://airlift.datometry.com/opendb-for-postrgresql/.

[5] JOOQ. SQL Translation. https://www.jooq.org/translate/.

[6] T. Kraska, T. Li, S. Madden, M. Markakis, A. Ngom, Z. Wu, and G. X. Yu. Check out the big brain on brad: Simplifying cloud data processing with learned automated data meshes. *Proc. VLDB Endow.*, 16(11):3293–3301, jul 2023.

[7] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[8] T. Mao. Python SQL Parser and Transpiler. https://github.com/tobymao/sqlglot, 2024.

[9] OpenAI. Embeddings - OpenAI API. https://platform.openai.com/docs/guides/embeddings/.

[10] OpenAI. Gpt-4 technical report, 2024.

[11] M. Pourreza and D. Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction, 2023.

[12] SQLines. Data and Analytics Platform Migration. https://www.sqlines.com/.

[13] R. Sun, S. O. Arik, H. Nakhost, H. Dai, R. Sinha, P. Yin, and T. Pfister. Sql-palm: Improved large language model adaptation for text-to-sql, 2023.

[14] I. Trummer. Codexdb: synthesizing code for query processing from natural language instructions using gpt-3 codex. *Proc. VLDB Endow.*, 15(11):2921–2928, jul 2022.