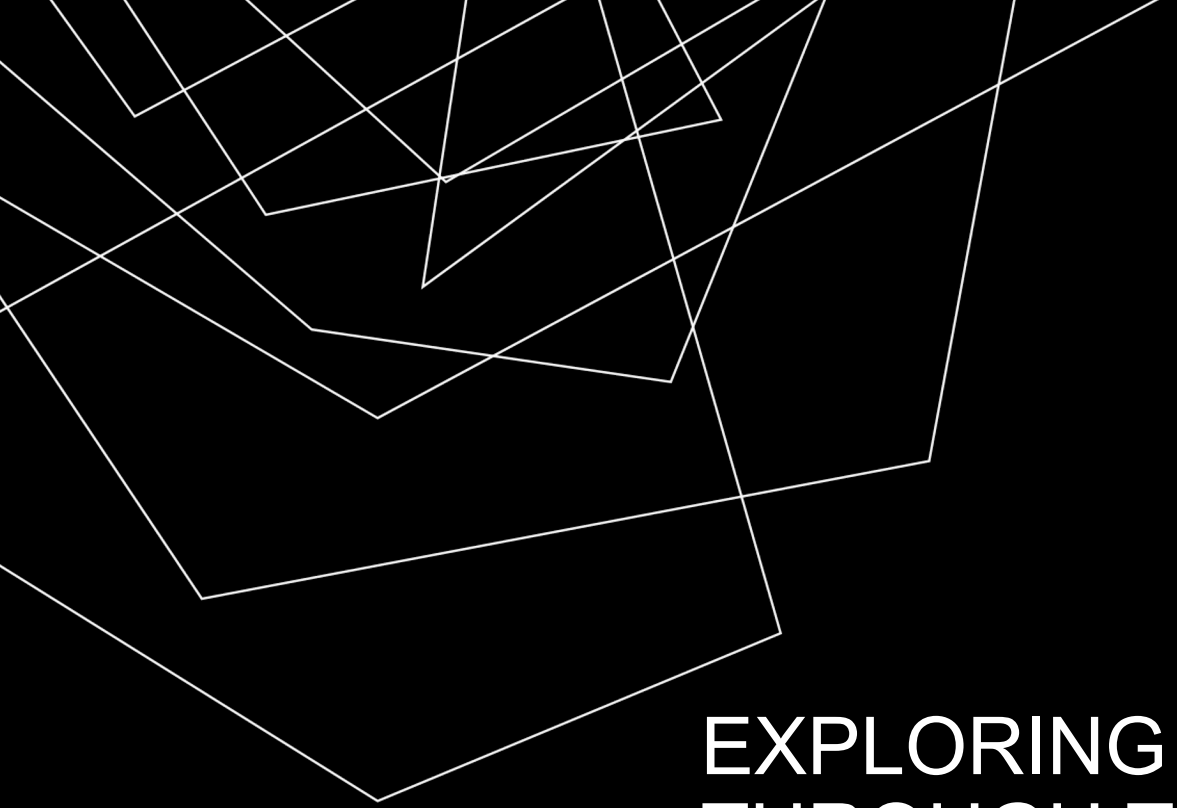


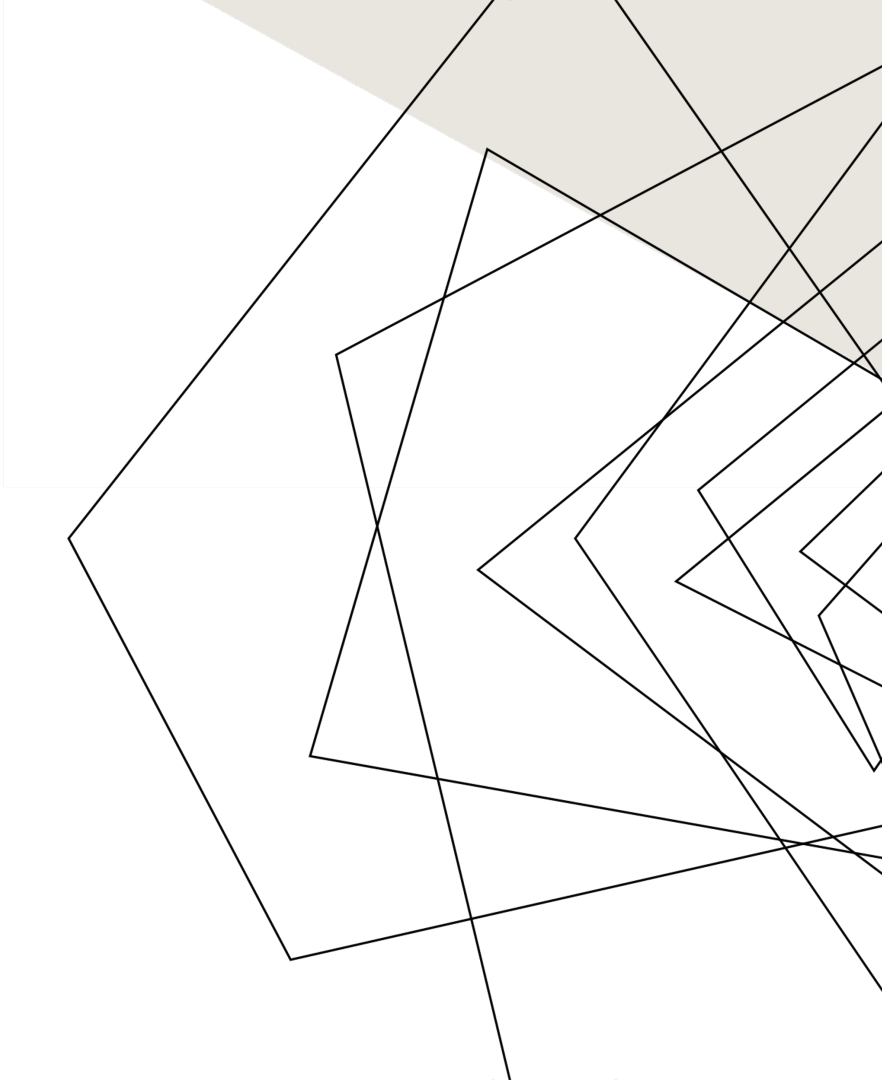
EXPLORING QUERY LANGUAGE THROUGH TRANSLATION

DS4300

EDDY LIU & QIXAING JIANG

CONTENT

- Introduction
- Methodology
- Steps
- Results
- Limitations



INTRODUCTION

Machine Translation (MT), like Google Translate, is forced to encode the quirks of natural language—including ambiguity and nuance.

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.

Strategies

Direct approach - The source language input is translated word by word in the target language.

Transfer approach - The source language and the target language messages are transferred into intermediate representations. The difference with interlingual MT is that the transfer approach depends on the language pair involved.

Interlingual approach - The source language input is transformed into a semantic representation of the text, an interlingua. The interlingua is the basis for generating the target text.

https://machinetranslate.org/rule-based-machine-translation

METHODOLOGY

- Inspired by the SQL queries on PySpark DataFrames and encouraged by Professor Rachlin, we decided to explore cross-query language translation.
- Specifically, we wanted to explore the relationship in querying languages between SQL and MongoDB by writing a SQL-to-Mongo translator.

STEP 1: Preprocessing

Create a Pandas DataFrame storing the syntax of operators

for example:

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

```
keywords = ["select", "from", "where"]
comp_ops = {
    "=": "$eq",
    ">": "$gt",
    "<": "$lt",
    ">=": "$gte",
    "<=": "$lte",
    "or": "$or",
    "and": "$and"
}

# Create DataFrame to store the comparison operators mappings
df_comparison_ops = pd.DataFrame(comp_ops.items(), columns=["SQL", "MQL"])
df_comparison_ops
```

	SQL	MQL
0	=	$eq
1	>	$gt
2	<	$lt
3	>=	$gte
4	<=	$lte
5	or	$or
6	and	$and

```
df_comparison_ops[df_comparison_ops["SQL"]=="or"]["MQL"].item()
```

```
'$or'
```

STEP 2: SQL-to-Dict

Create a custom SQL string-parser function that creates a dictionary representation of the SQL Query.

store each element of the query in a potentially-recursive dictionary, for example:

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

"FROM":"db",

"WHERE":{ # conditions

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

```
query = """SELECT (sum(quantity)) from (Test_data)"""
q_dict
```

```
{'select': 'sum(quantity)', 'from': 'test_data'}
```

STEP 3: Dict-To-MQL

Create a Mongo_Reconstructor

```
1  query = """SELECT (sum(quantity)) from (Test_data)"""
2  q_dict
```

```
{'select': 'sum(quantity)', 'from': 'test_data'}
```

Example:

reconstructs the SQL Query as a Dictionary for a Mongo Query.

```
{"$or": [{"x": {"$gt": 0}},
  {"y": {"$lte": 1}}]}
```

STEP 4: Testing

Test for Correctness:

- download a CSV into both Mongo and SQL
- call aggregation functions on both

```
# # Define a query for summing quantities in the Test_data collection
# query_1 = {
#     "select": "sum(quantity)",
#     "from": "Test_data"
# }

# Create an instance of Dict2Mongo with the updated query_1
dict2mongo = Dict2Mongo(mongo_query)

# Generate and print the MongoDB query for the Test_data collection
mongo_query = dict2mongo.to_mongo_query()
print("MongoDB Query for Sum Aggregation in Test_data:", mongo_query)

```

```
MongoDB Query for Sum Aggregation in Test_data: {'from': 'test_data', 'pipeline': [{'$group': {'_id': None, 'total': {'$sum': '$quantit
y'}}}]}
```

```

```

```
# Connect to the local MongoDB instance
client = MongoClient('mongodb://localhost:27017/')

# Select your database
db = client['Test_data']

# Use the generated aggregation pipeline from dict2mongo_1 or dict2mongo_2
pipeline = mongo_query['pipeline']  # For sum aggregation example

print(pipeline)

# Execute the aggregation query on the Test_data collection
result = db['Test_data'].aggregate(pipeline)

# Print the result
for doc in result:
    print(doc)

```

```
[{'$group': {'_id': None, 'total': {'$sum': '$quantity'}}}]
{'_id': None, 'total': 20}
```

RESULT

```
# # Define a query for summing quantities in the Test_data collection
# query_1 = {
#     "select": "sum(quantity)",
#     "from": "Test_data"
# }

# Create an instance of Dict2Mongo with the updated query_1
dict2mongo = Dict2Mongo(mongo_query)

# Generate and print the MongoDB query for the Test_data collection
mongo_query = dict2mongo.to_mongo_query()
print("MongoDB Query for Sum Aggregation in Test_data:", mongo_query)

```

```
MongoDB Query for Sum Aggregation in Test_data: {'from': 'test_data', 'pipeline': [{'$group': {'_id': None, 'total': {'$sum': '$quantit
y'}}}]}
```

```

```

```
# Connect to the local MongoDB instance
client = MongoClient('mongodb://localhost:27017/')

# Select your database
db = client['Test_data']

# Use the generated aggregation pipeline from dict2mongo_1 or dict2mongo_2
pipeline = mongo_query['pipeline']  # For sum aggregation example

print(pipeline)

# Execute the aggregation query on the Test_data collection
result = db['Test_data'].aggregate(pipeline)

# Print the result
for doc in result:
    print(doc)

```

```
[{'$group': {'_id': None, 'total': {'$sum': '$quantity'}}}]
{'_id': None, 'total': 20}
```

LIMITATIONS

- Joins: MongoDB's lack of native join support can complicate the conversion of SQL queries involving multiple tables.
- Transactions: MongoDB's transaction model may not fully match SQL databases, impacting transactional features and capabilities.
- Tooling and Ecosystem: MongoDB's tooling and ecosystem may not offer the same breadth and depth as SQL databases, impacting developer productivity and integration options.

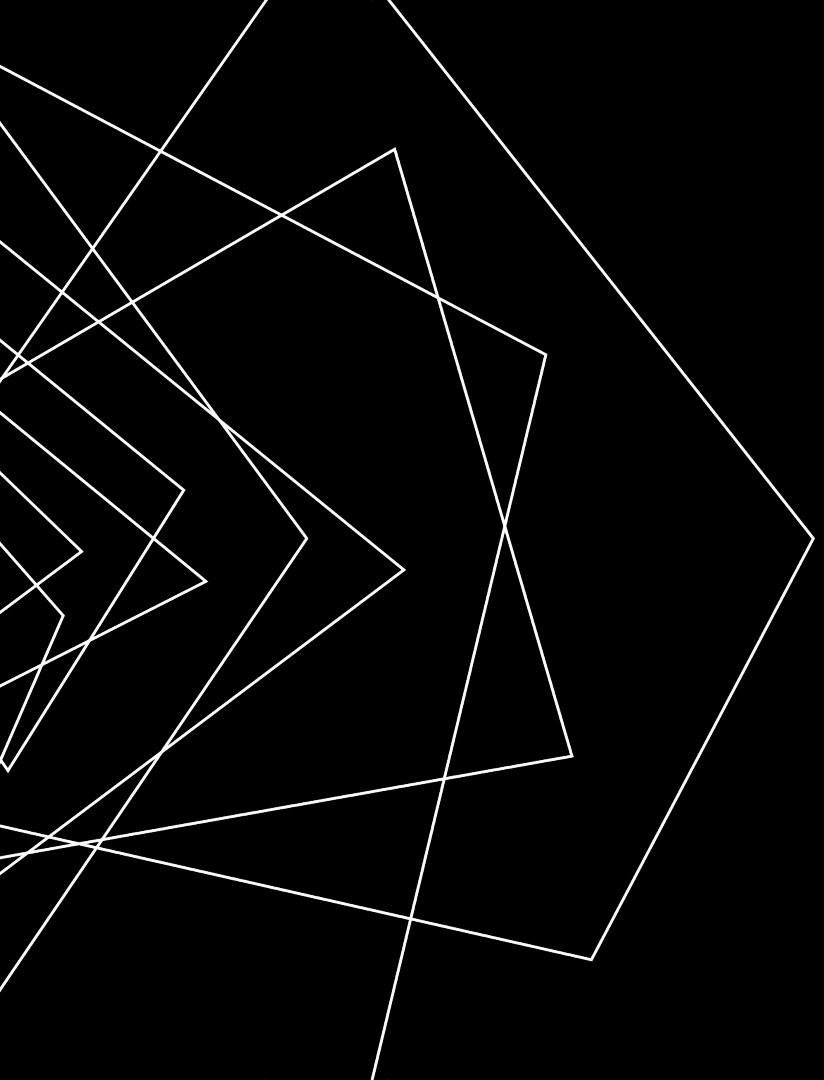
FUTURE LIMITATIONS

- two classes per query language
- needs to encode ‘niche’ query infrastructure
- difficult to maintain a consistency-first algorithm

Potential Solutions

- ML language models?
- Maintained as an extension to API?

Q&A



THANK YOU!