Toward effective text-to-MongoDB query translation

Aicha Aggoune [a,1,*]

[a] Computer Science Department, LabSTIC, University of 8th May 1945, Guelma, Algeria
[1] aggoune;aicha@univ-guelma.dz
* corresponding author

ARTICLE INFO

Article history
Received January 3, 2026
Revised February 20, 2026
Accepted April 13, 2026
Available online May 31, 2026

Keywords
Flexible schema
MongoDB querying
Query translation
RAG-SBERT-T5-base
Semantic ambiguity

ABSTRACT

Translating natural language questions into MongoDB queries is critical for flexible data access in current NoSQL systems. However, semantic ambiguity in user questions and MongoDB's dynamic schema make this work difficult. This study presents QMQL (Question to Mongo Query Language), a hybrid approach meant to address these challenges. QMQL combines a Graph Attention Network (GAT) for refining schema elements with a Retrieval-Augmented Generation (RAG) mechanism that employs BERT embeddings to retrieve relevant schema and resolve semantic ambiguity. A T5-base model is used to generate a MongoDB query corresponding to the user's question. An experimental evaluation on an extended dataset encompassing various real-world domains demonstrates the effectiveness of the proposed approach. QMQL achieves excellent performance with an EMA of 0.89, an EM of 0.91, and a BLEU score of 0.95, exceeding previous approaches, particularly for semantically ambiguous questions and sophisticated queries across flexible MongoDB schemas.

1. Introduction

MongoDB is a NoSQL document-oriented database that stores data as collections of JSON-like documents. Each document comprises a set of field-value pairs, enabling the inclusion of nested fields and embedding documents. MongoDB's query language (MQL) supports complex queries, aggregation pipelines, geospatial queries, and text search via the MongoDB shell. This allows developers to specify various data access patterns [1]. Traditionally, retrieving data from databases requires specific knowledge about the query language and the underlying data structure. A Natural Language Interface (NLI) provides adaptable front-ends to pose queries such as ReactQueryBuilder [2]. This approach is particularly difficult for novice users. Natural language database querying (NLDQ) is a straightforward method that involves querying databases using natural language questions [3]. This approach would seamlessly integrate into human interactions and represents the best user interface for many domains. Existing works mainly focus on translating natural language questions into SQL queries for relational databases, using advanced machine learning and deep learning methods such as the seq2SQL and Rasat models based on seq2seq model [4], [5], the sketch-based methods [6], [7] divide the SQL into sub-modules, each corresponding to a particular prediction slot to be filled, LSTM-based text-to-SQL [8], Bert-SQL [9], etc. However, research on translating natural language questions into NoSQL queries has received relatively little attention, mainly due to the flexibility of schema structures and the diversity of NoSQL query languages, which further increases the complexity of the translation process. These studies used transformer-based models to improve both the encoder and the decoder modules. For example, the Two-Stage Controllable (TSC) framework [10] was proposed to leverage pre-trained

language models (PLMs), such as BERT, to translate questions into corresponding template questions. The latter are then used by the DistilBERT encoder to extract named entities for generating queries for Elasticsearch, a document-oriented database. Similarly, the GPT-3.5 model [11] was applied to question-answering over Elasticsearch. A BERT-based question-answering system [12] was developed to formulate various types of MongoDB queries from natural language questions, including Find, Insert, Update, and Remove clauses. A GPT-4-based chatbot [13] is being developed to manipulate a Neo4j graph database by translating natural-language questions into Cypher queries. Similarly, a natural language interface [14] based on a graph storage technique, namely labelled property graphs, enables explicit modelling of relationships among event data. It takes a natural-language question from the user and automatically constructs a corresponding Cypher query to be executed on the stored event data. SMART [15] is a multi-step approach for Text-to-NoSQL conversion. It combines SLM (Small Language Model) to generate initial schema-aware NoSQL queries and RAG (Retrieval-augmented Generation) to refine the query using semantically similar examples from the TEND dataset, which contains natural language questions paired with NoSQL queries. In [16], the authors introduce MultiTEND, the first and largest multilingual benchmark for natural language-to-NoSQL query generation, covering six languages: English, German, French, Russian, Japanese, and Mandarin Chinese. To improve multilingual performance, they propose the Multilink framework, and the experimental results demonstrate a boost in execution accuracy of about 15% for English and an average improvement of 10% for non-English languages.

While existing text-to-NoSQL translation methods fail to address the semantic ambiguity problem in natural language questions, the study in [17] addresses this issue by leveraging an LSTM-RNN with a knowledge graph to semantically identify and infer relevant answers from Noe4J, a NoSQL graph-oriented database, within the Vietnamese Tourism domain. The results show the effectiveness of the proposed approach to managing semantic ambiguity in Vietnamese tourism questions.

In this paper, we present QMQL (Question to Mongo Query Language), an intelligent semantic model that translates natural-language questions into MongoDB queries, assisting end-users in the data-querying process within question-answering systems, chatbots, or any system that stores data in MongoDB. The user-facing interface of the proposed approach takes a user query in natural language and automatically generates a corresponding MongoDB query to execute against the database.

The main contributions of this work are organized as follows:

- GAT-Based Schema Representation: To model a flexible schema of a document-oriented database, we use a Graph Neural Network (GNN) [18], especially a Graph Attention Network (GAT), to capture both structural and semantic links between schema items.
- Schema-Grounded RAG Framework: We develop a retrieval-augmented generation (RAG) [19] framework for NoSQL document-oriented databases that leverages the refined schema representations produced by the GAT module to guide query generation, ensuring contextually appropriate responses. We use a fine-tuned T5-base [20] (Text-to-Text Transfer Transformer– base) with 220 million parameters for query generation from a question, along with the original textual description of the relevant schema.
- Generalizable Disambiguation approach: We present a disambiguation approach for resolving ambiguity in natural language questions, allowing the model to be applied to new schemas and NoSQL databases. This technique is based on Sentence-BERT (SBERT) [21] for effective word-sense disambiguation.
- Integration with End-to-End Evaluation: We demonstrate that combining these components into a single pipeline enhances accuracy and generalization over previous work, emphasizing the combined advantage while maintaining each component's originality.

The obtained QMQL model was trained on our extended datasets of question-query pairs [22] across three different domains: movies, student training, and finance. This diversity allows us to evaluate the robustness of the model in understanding user questions.

The remainder of this paper is outlined as follows: Section 2 presents the principal components used in this model. Section 3 describes the QMQL model. The experimentation study and discussion are presented in Section 4. Finally, Section 5 concludes the paper and outlines the future directions.

2. Theoretical Foundations

The proposed model for translating natural-language questions into corresponding MongoDB queries addresses two key issues: schema flexibility and semantic understanding. The model is built on a fine-tuned T5-base combined with a GAT and RAG to extract a document-oriented schema under dynamic data structures, addressing schema flexibility. In addition, SentenceBert is used as a retriever in the RAG to derive semantically meaningful sentence embeddings of the user's question, effectively addressing word-sense ambiguity. Altogether, this architecture enables accurate translation of natural language questions into MongoDB queries while preserving schema flexibility and handling semantic ambiguity in both simple and complex queries.

2.1. MongoDB database

MongoDB is a document-oriented, operational database built from the ground up as an alternative to relational databases for handling big data and unstructured information [23]. Unlike relational databases, MongoDB allows developers to store rich JSON-like documents, known as BSON (Binary JSON), that map naturally to the objects they use in their code. MongoDB's architecture enables efficient CRUD (Create, Read, Update, and Delete) operations, which are crucial for data access and processing [24]. A notable feature of MongoDB is its horizontal scalability through sharding, which allows the system to handle a vast volume of queries and data distributed across multiple servers. In addition, its robust security keeps the data safe whether it is stored or being transferred. MongoDB provides different types of indexes, each designed for specific query types. The most common type is the standard B-tree index, which offers a good mix of speed and flexibility by supporting basic tasks such as searching, sorting, and combining data, especially for queries that use multiple fields [25]. Fig. 1 shows an example of a MongoDB document of the movie domain [26] and a natural language question with the corresponding MongoDB query.

Fig. 1. Example of a JSON document, question, and the corresponding MongoDB query.

The architecture of MongoDB includes the Document Model, key-value pairs, relationships, objects, graphs, geospatial, unified interface, transactional, search, mobile, real-time analytics, security, multi-cloud, and distributed architecture [27]. Mapping relational databases to MongoDB remains an active area of research in order to explore and fully leverage the advantages of MongoDB [28], [29], [30]. In recent years, the migration from object-relational databases to NoSQL document-oriented databases has emerged as an important and rapidly growing research domain [31], [32].

The schema-less design of MongoDB allows each document to have different fields and data types, allowing for dynamic data storage. This feature makes MongoDB ideal for rapidly changing applications

and unstructured data. However, this flexibility makes translating natural-language questions into MongoDB queries more challenging. In this work, we use a MongoDB database containing documents from three different domains: movies, student training, and finance, to evaluate the robustness of the proposed model in understanding the user questions. This database was created by combining three publicly available MongoDB sample databases [33].

- The first database, sample_mflix, contains data on movies distributed across six collections: comment, embedded_movies, movies, sessions, theatres, and users.
- The second database, sample_training, contains realistic data from MongoDB Private Training Offerings to help students explore MongoDB's functionality. It includes seven collections: companies, grades, inspections, posts, routes, trips, and zips.
- The third database, sample_analytics, represents a typical financial services application and contains three collections: customers, accounts, and transactions.

Together, these sources from a heterogeneous MongoDB database that serves as the underlining knowledge base for answering user questions expressed in natural language. From this database, we constructed a dataset of natural language questions paired with their corresponding MongoDB queries. To comply with the requirement of double-blind peer review, the dataset is fully anonymized.

2.2. Graph Attention Network Model

The graph attention network (GAT) is a type of graph neural network (GNN) architecture that leverages masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations [34]. The idea is to compute the hidden representations of each node in the graph by attending to its neighbors, following a self-attention strategy that can process efficiently on arbitrarily structured graphs. Using a self-supervised GAT to exploit the flexible MongoDB schema is a suitable technique, as it allows the model to automatically learn the different components of the database without requiring labeled data. The application of GNN models has primarily been studied for translating text into SQL; however, at present, very little work explores their use for translating text into NoSQL [35], [36], [37]. The work introduced in [38] begins with a Relational Graph Convolutional Network (RGCN) for automatically detecting intricate relationships within NoSQL data, building node embeddings, which encode the structure and the semantics of the schema. The automated feature selection across multimodal data is addressed through the Tree-based Pipeline Optimization Tool (TPOT) with transformers-BERT and convolutional neural networks-ResNet. For analyzing nested NoSQL structures, the Hierarchical Attention Networks (HAN) were employed, which improved the classification performance with an F1-score of 0.88. This combination demonstrates significant performance gains for NoSQL-based systems that open the way for efficient treatment of large-scale, heterogeneous datasets and samples.

In contrast, our approach adopts the GAT model to provide a global representation for each node, enabling more efficient extraction heterogeneous and dynamic MongoDB schemas. GATs introduce the attention mechanism a, a single-layer feedforward neural network, parametrized by weight vector $\vec{a} \in \mathbb{R}^{2F'}$. The input to layer is a set of node features, $h = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$, $\vec{h}_i \in \mathbb{R}^F$, where N is the number of nodes, and F is the number of features in each node. The layer produces a new set of output node features $h' = \{\vec{h'}_1, \vec{h'}_2, \ldots, \vec{h'}_N\}$, $\vec{h'}_i \in \mathbb{R}^{F'}$. To obtain sufficient expressive power to transform the input features into higher-level features, a shared linear transformation $\vec{y}_i$ of weight matrix $W \in \mathbb{R}^{F' \times F}$ is required to every node i, where: $\vec{y}_i = \vec{h}_i . W^k$ with K denotes the number of attention heads. For each edge e_{ij}, the attention coefficients and Softmax function are calculated as:

$$e_{ij} = a(W\vec{h}_i, W\vec{h}_j) \tag{1}$$

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in N(i) \cup \{i\}} exp(e_{ik})} \tag{2}$$

The attention coefficients are normalized by applying the LeakyReLU [39] nonlinearity with negative input slope α = 0.2.

$$\alpha_{ij} = \frac{exp\left(LeakyReLU(\vec{a}^T[W\vec{h}_i||W\vec{h}_j])\right)}{\sum_{k\in N_i} exp\left(LeakyReLU(\vec{a}^T[W\vec{h}_i||W\vec{h}_j])\right)} \quad (3)$$

The output node features $\vec{h}'_i$ is finally computed in the final GAT layer as follows:

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in N(i)\cup\{i\}} \alpha_{ij}^k \vec{y}_j\right) \quad (4)$$

2.3. Retrieval Augmented Generation Technique

Retrieval-augmented generation (RAG) is a powerful technique that enhances downstream task execution by retrieving additional information, such as knowledge, skills, and tools from external sources [40]. The RAG system consists of two core modules [41]: the retriever and the generator. The retriever searches for relevant information from the data store, and the generator produces the required contents. The RAG process is as follows: (i) the retriever receives the input query and searches for relevant information; (ii) the original query and retrieval results are then fed into the generator through a specific augmentation methodology; and (iii) finally, the generator achieves the desired results. Widely used retrieval methods are sparse and dense retrieval, where the entire process can be divided into two distinct phases: (i) each object is first encoded into a specific representation; (ii) an index is constructed to organize the data source for efficient search.

To address semantic ambiguity in natural language questions and improve the matching with MongoDB queries, dense retrieval is more suitable. In particular, the Sentence-BERT dense model offers a deeper understanding of textual meaning, outperforming traditional sparse methods, such as BM25 [42].

Sentence-BERT (SBERT) uses Siamese and triplet networks to derive semantically meaningful sentence embeddings [21]. It computes semantic similarity between sentences by encoding each sentence into a shared embedding space. SBERT employs a Siamese architecture, where identical BERT encoders independently generate fixed-sized vectors, avoiding BERT's costly cross-attention operations.

In the proposed approach, SBERT is utilized prior to MongoDB query generation to match parts of the natural language question with the most relevant components of the MongoDB schema to identify which collections, fields, and nested attributes the question refers to.

Given a user question Q, SBERT encodes it into a dense embedding that captures the semantic meaning of the question rather than relying on keyword-level similarity: $\overrightarrow{e_Q}$= SBERT(Q)

From the graph representation of the MongoDB schema $h' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}$, generated by the GAT model, SBERT computes the semantic similarity (using cosine measure) between the dense embeddings of the user question $\overrightarrow{e_Q}$ and the embeddings of the schema node i.

$$sim_{cos}(\overrightarrow{e_Q}, \vec{h}'_i) = \frac{\overrightarrow{e_Q}.\vec{h}'_i}{\|\overrightarrow{e_Q}\|\|\vec{h}'_i\|} \quad (5)$$

By utilizing similarity, contrastive learning allows our model to align the semantic embedding of a user question with the structural embedding of the right schema component. We use InfoNCE [43] as the objective function, where the model brings a question and its matching GAT-encoded schema node together in the latent space while excluding irrelevant nodes. This alignment improves SBERT's ability to obtain the most important schema items during the RAG process, leading to more accurate and semantically grounded MongoDB query generation. InfoNCE minimizes the loss function defined between the embedding question $\overrightarrow{e_Q}$, its corresponding positive schema embedding $\vec{h}^+$, and a set of negative schema embeddings $\{\vec{h}^-_i\}$:

$$\mathcal{L}_{InfoNCE} = -\log \frac{\exp(s(\vec{e_Q}.\vec{h}^+)/\tau)}{\exp\left(\frac{s(\vec{e_Q}.\vec{h}^+)}{\tau}\right)+\sum_{j=1}^{M}\exp\left(s(\vec{e_Q}.\vec{h_j}^-)/\tau\right)} \quad (6)$$

with t is the temperature parameter introduced to avoid the gradient saturation. After retrieving the corresponding schema of the user question, the second step of the RAG pipeline is focused on MongoDB query generation. The input question and retrieved relevant schema are then fed into the generator through a specific augmentation methodology. We used a fine-tuned T5-base to translate the augmented user question input into its corresponding executable MongoDB query.

2.4. T5-base Generator

T5 (Text-to-Text Transfer Transformer) [20] is a pretrained generative language model developed by Google AI in 2020 that supports more languages than other NLP-specific models. It is intended to manage all NLP problems (e.g., summarization, translation, classification, etc.) as a text-to-text transformer, allowing it to perform a variety of tasks within a unified framework. In our work, we use the T5-base variant as a generator component within the RAG pipeline, as it provides stronger accuracy without excessive resource demands [44].

T5 generates target tokens based on encoder-decoder architecture, where the encoder input is the user question augmented with the retrieved schema components from the AT-based graph representation for the translation task. The encoder represents both the semantic intent of the question and the schema's structural limitations at the same time. This creates contextualized representations that help the decoder when it is generating queries. Based on these representations, the decoder uses autoregression to create the target query tokens. This makes sure that the output is both semantically correct and follows the structure of the underlying schema.

Let $\mathrm{q} = (\mathrm{q}_1, \mathrm{q}_2, \ldots, \mathrm{q}_\mathrm{n})$ denote the token sequence of the natural language question and $\mathrm{s} = (\mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_\mathrm{m})$ denote the token sequence representing the relevant schema components. The encoder maps the augmented sequence input $[\mathrm{q}, \mathrm{s}]$ into a sequence of contextualized hidden representations H: $\mathrm{H} = \mathrm{Encoder}([q, s])$. The decoder generates the MongoDB query Q, autoregressively, conditioned on the encoder representation: $P(Q|[q,s]) = \prod_{t=1}^{T} P(y_t|y_{<t}, H])$. With $\mathrm{y_t} = \mathrm{Decoder}(\mathrm{y_{<t}}, \mathrm{H})$ and $\mathrm{y_{<t}} = (\mathrm{y_1}, \mathrm{y_2}, \ldots, \mathrm{y_{t-1}})$ are previously generated tokens.

3. Method

We propose the QMQL (Question to Mongo Query Language) model to convert a natural-language question into its corresponding MongoDB query language (MQL). The model operates under the flexible schema constraints of document-oriented databases and effectively addresses semantic ambiguity in natural-language questions. Fig. 2 illustrates the pipeline for generating MongoDB queries from natural-language questions, designed to assist end users in the data-querying process. The user-facing interface of the proposed pipeline takes a user query, formulated in natural language, and automatically constructs a corresponding MQL query to be executed over the MongoDB process data.

The flexible schema of a MongoDB database is modelled as a graph, where each node represents a collection and each edge represents a reference relationship between collections. To be specific, the node $\mathrm{n_i}$ that corresponds to a collection $\mathrm{C_i}$ can be defined as:

$$\mathrm{n_i} = (\mathrm{C_i}, \mathrm{Fields}(\mathrm{C_i})) \quad (7)$$

with $Fields(\mathrm{C_i})$ is the union of all field names of all documents belonging to the collection $\mathrm{C_i}$. As a result, each field name is included only once, thereby removing redundancy and yielding a compact node-level representation.

$$\mathrm{Fields}(\mathrm{C_i}) = \bigcup_{\mathrm{j=1}}^{|\mathrm{C_i}|}\{\mathrm{k}|\ (\mathrm{k}\colon \mathrm{v}) \in \mathrm{d_{ij}}\} \quad (8)$$

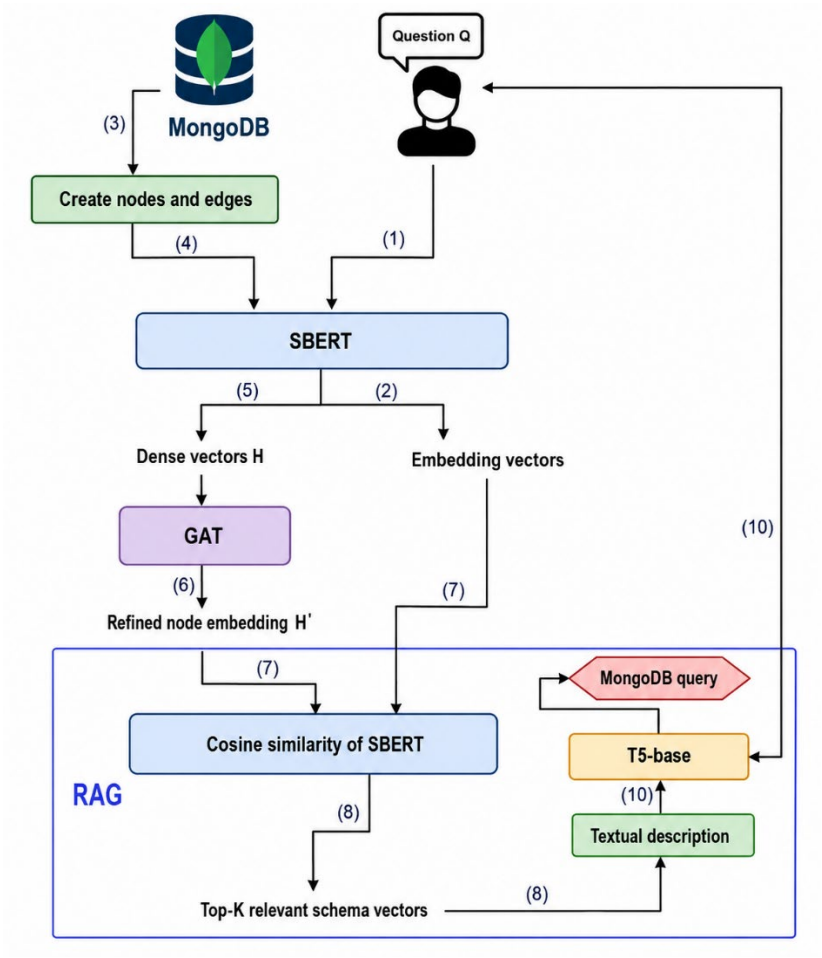

Fig. 2. QMQL pipeline.

The extraction of collection names and their corresponding lists of fields is performed using the MongoDB command db.list_collection_name() for retrieving collection names and db[column_name].find({}, {_id:0}) for obtaining the fields of each collection, excluding the _id field.

An edge e_{ij} is created between nodes n_i and n_j, if at least one field in collection C_i references the primary key of the collection. In practice, such references are frequently inferred using heuristics like naming conventions (for example, fields ending with _id that match the name of another collection). Formally:

$$e_{ij} = (n_i, n_j) \text{ if } \exists k \in \text{Fields}(C_i) \text{ such that } k \rightarrow C_j.\_id. \quad (9)$$

thus, the schema graph G is defined as $G = (N, E)$, where N and E denotes the sets of nodes and edges, respectively. Since database schemas may evolve over time through the addition of new collections, modifications to fields, or the introduction of new inter-collection relationships, the QMQL model must identify such structural changes and recompute the corresponding schema graph embeddings. To detect schema evolution, the QMQL framework periodically checks structural metadata by computing the number of essential schema components, namely: (i) the number of collections, (ii) the number of inter-collection references, and (iii) the total number of fields across collections. These structural indicators serve as lightweight signatures of the schema state. More formally, the schema evolution is detected when at each monitoring interval t: $(N_t, E_t, F_t) \neq (N_{t-1}, E_{t-1}, F_{t-1})$. If any of these numbers change, the schema is updated. This technique provides an efficient way for detecting structural modifications without reprocessing the full database content.

After detecting a change, the following steps are executed:

- The graph is recreated to reflect the additional collections, updated field sets, or updated relationships.
- Each collection node is re-encoded using its modified metadata representation (collection name and list of fields).
- A forward pass through the GAT layer recomputes attention coefficients and updates node embeddings.

QMQL integrates a Graph Attention Network (GAT) to capture fine-grained dependency relationships among nodes in the graph by assigning adaptive attention weights to their neighbors, thereby enabling selective and contextual feature aggregation (refer to Steps 1 to 6). This representation is then combined with a Retrieval-Augmented Generation (RAG) framework that leverages SBERT for semantic retrieval and T5-base for generating the MongoDB queries that are directly executable to return answers (refer to Steps 7 to 9). Although these components work together within a unified architecture, each provides distinct scientific insights, resulting in accurate, versatile, and schema-aware query generation.

We first introduce the initial component for query generation, namely a GAT-based schema representation method designed to enhance flexible document-oriented schemas. GAT operates on graph-structured data, where it takes node embeddings derived from the graph as input and produces refined node representations.

The natural language question $Q = \{q_1, q_2, \dots, q_{|Q|}\}$ is a sequence of $|Q|$ tokens. The MongoDB database schema consists of N collections $C = \{C_1, C_2, \dots, C_N\}$ and each collection C_i consists of a set of documents $C_i = \{d_{i1}, d_{i2}, \dots, d_{i|C_i|}\}$. Each document d_{ij} in collection C_i is represented by a set of key-value pairs: $d_{ij} = \{(k_1 : v_1), (k_2 : v_2), \dots, (k_{|K_{ij}|} : v_{|K_{ij}|})\}$ with $|K_{ij}|$denotes the number of keys in document d_{ij} of the collection C_i.

The input to a GAT layer is a set of dense embedding vectors representing the MongoDB schema. These embeddings are obtained using the SBERT encoder, which maps the textual description of each collection, composed of the collection name and its associated fields into fixed-size dense vector representations (refer to Steps 4 and 5). Formally, this process is defined as:

$$h = Encoder_{SBert}(\text{name of } C_i, Fields(C_i)) = \{\vec{h}_1, \vec{h}_2, \dots, \vec{h}_N\} \quad (10)$$

where N denotes the number of collections in the database. In parallel, the user question Q expressed in natural language is encoded using the SBERT encoder to obtain its embedding $\overrightarrow{e_Q} = \text{SBERT}(Q)$ (refer to Steps 1 and 2).

The output of the GAT layer consists of the refined node features $\vec{h}'_i$ of the node n_i (refer to Step 6). These representations are subsequently fed into the RAG framework, where SBERT is used to compute the similarity between each node embedding $\vec{h}'_i$ and the user question embedding $\overrightarrow{e_Q}$ with $\text{score}(n_i) = \cos(\overrightarrow{e_Q}, \vec{h}'_i)$ (refer to Steps 7 and 8).

The top-K nodes with the highest similarity scores are selected as the relevant schema vectors, providing both the structural and semantic context for the subsequent T5-base generation stage. In our work, we set K=5 based on experiment results, balancing retrieval completeness and efficiency.

The T5-base generator, fine-tuned on MongoDB generation, takes as input the user's natural language question along with the original textual description of the relevant schema, including the collection name and its associated fields, which together represent the context provided to T5-base. For complex questions, we employ a Zero-Shot Chain-of-Thought [45] prompting strategy, guiding T5-base to reason step-by-step based on the relevant schema context before generating the MongoDB query.

4. Results and Discussion

To evaluate the performance of the proposed QMQL transformation model, a set of experiments was conducted on a PC equipped with an Intel Core i5 processor, 16 GB of RAM, and an NVIDIA RTX GPU to accelerate model training and inference. The T5-base model is fine-tuned via supervised learning on a dataset of natural-language questions paired with their corresponding MongoDB queries. The M2Q2+ dataset is constructed using a hybrid approach that combines manual annotation with data augmentation techniques [46], followed by quality assessment [22]. It extends the original M2Q2 dataset [47] by enriching the collection of Movie question-query pairs with more complex queries and by incorporating two additional domains: student training and finance.

consists of JSON documents containing question-query pairs drawn from three diverse domains: movies, student training, and finance, thereby enabling an evaluation of the proposed model's robustness and domain generalisation capability in interpreting and understanding user questions. Table 1 summarises the main properties of our dataset.

Table 1. Dataset Description.

Property	Description
Total size	520k (k=100.000)
Domain distribution	Movies 120k, students training (200k) , and finance (200k)
Quality control	Manual verification+ schema consistency checks, and removal of ambiguous or invalid queries
Input example: "Movies"	List movie titles along with their comments.
Output example "Movie"	db.movies.aggregate([{$lookup: {from: "comments", localField: "_id", foreignField: "movie_id", as: "movieComments"} }]);
Input example "Student training"	List students by grades where any exam score is greater than 90.
Output example "Student training"	db.grades.find({"scores": { $elemMatch: { type: "exam", score: { $gt: 90 } }}});
Input example: "Finance"	Get the five accounts with the highest credit limits.
Output example "Finance"	db.accounts.find().sort({ limit: -1 }).limit(5);

4.1. Determining K for the top-K relevant schema components

The initial evaluation of our model aims to assess the RAG framework's ability to retrieve the top-K most relevant schemas using SBERT. The best value of K is found using two essential performance measures: precision and recall, which are computed with reference to the ground truth produced during dataset construction. We start with K = 1 and gradually increase the value. For each K, several experimental runs are carried out, and the average precision and recall scores are presented. In the context of our study, we define the following evaluation measures:

$$Precision = \frac{|relevant\ schema \cap retrieved\ schema|}{|retrieved\ schema|} \tag{11}$$

$$\text{Recall} = \frac{|\text{relevant schema} \cap \text{retrieved schema}|}{|\text{relevant schema}|} \tag{12}$$

Table 2 reports the results of the top-k schema retrieval experiments. For each value of K, a total of 700 natural questions across three domains are evaluated; therefore, we measure the average of precision and recall. These results are used to evaluate the retrieval module's performance and to determine the optimal number of schema items to send to the downstream query-generating model.

The results show that the small value of K (K<3) leads to low average precision and recall, indicating insufficient schema coverage. Performance improves at K=4 and reaches its optimum at K=5, with both Precision and Recall at 0.98, indicating an outstanding balance between relevance and coverage. Increasing K results in slightly lower precision and recall, indicating that retrieving more schema pieces adds noise without boosting coverage. As a result, K = 5 is selected as the fixed number of the most relevant schema elements for generating MongoDB queries from user natural-language questions.

Table 2. Results of the selected top-K relevant schema.

K	AVG(Precision)	AVG(Recall)
1	0.40	0.45
2	0.45	0.45
3	0.45	0.42
4	0.53	0.55
5	**0.98**	**0.98**
6	0.98	0.95
7	0.97	0.94
8	0.96	0.90
9	0.96	0.90

Compared with models that rely exclusively on code-pretrained generation, explicitly establishing query generation on the top-K schema components returned improves execution accuracy significantly, especially under schema drift. QMQL retains greater robustness to domain changes by employing schema-aware embeddings from GAT and semantic matching from SBERT, whereas code-pretrained generators experience significant reductions in execution accuracy.

4.2. Fine-tuning T5-base

After determining the optimal top-k value, the T5-base generator within the RAG framework is fine-tuned using the relevant schema elements as context. T5 takes as input a natural-language question augmented with the selected K relevant schema components and is trained to generate the corresponding MongoDB queries.

To ensure a thorough assessment and avoid schema leakage, the dataset is split into 80% training and 20% testing sets at the schema-component level rather than at the individual question-query pair level. Schema components are initially identified as atomic units. All question-query pairs derived from the same schema fragment are then simply allocated to a single split. The splits are designed to ensure that no schema fragment is present across all sets, hence guaranteeing that the model is assessed on novel schema structures during testing. In addition, to provide a fair and unbiased evaluation, the split is domain-based, with each subset containing representative samples from the movie, student training, and finance domains. This technique enables a reliable evaluation of the model's robustness across diverse application domains. During the training phase, two key tasks- preprocessing and fine-tuning steps are applied:

- Task-Specific Prefix Insertion: A Task-Specific Prefix is prepended to each input sequence to explicitly define the fine-tuning objective and enrich the contextual representation. The prefix, "Translate Question to MongoDB Query," is added to the input sequence prior to tokenization and encoding.
- Lowercase normalization: The input data is converted to lowercase to eliminate inconsistencies arising from case variations, thereby ensuring uniform text representation and more stable model training.

In addition, hyperparameters that control the fine-tuning process are selected via a grid search based on validation performance. The final model configuration is trained using the optimal hyperparameters. Table 3 presents the hyperparameters considered for fine-tuning, their respective value ranges, and the optimal values.

Table 3. Hyperparameters for fine-tuning T5-base.

Hyperparamter	Value ranges	Optimal value
Learning rate	1e-5, 3e-5, 5e-5	3e-5
Batch size	8, 16	8
Number of epochs	5, 10, 15, 20	10
Maximum input length	256, 512	512
Maximum target length	128, 256	256
Optimizer	Adam, AdamW, Adafactor	AdamW
Loss Function	Cross-entropy, Label-smoothinh	Cross-entropy

Table 3 shows the hyperparameters that optimize validation performance. A learning rate of 3e-5 and a batch size of 8 ensure consistent convergence while maintaining generation accuracy. Training for 10 epochs provides the optimal balance of underfitting and overfitting. Larger input and target lengths (512 and 256, respectively) enable the model to fully leverage schema context and generate complete MongoDB queries. The AdamW optimizer routinely beats other optimizers, while cross-entropy loss is the most effective method for structured query generation, resulting in improved syntactic and semantic accuracy.

4.3. Computational cost analysis

To evaluate the feasibility of the proposed QMQL framework in interactive environments, we analyze the computational costs of its main components: the GAT model for schema refinement, SBERT for word-sense disambiguation, and T5-base for query generation.

The GAT module operates on the schema graph G=(N, E) rather than on the database contents, where the number of nodes N corresponds to schema elements (collections and fields F). Although GAT is well-known for its graph attention layer, which is computationally efficient because it avoids costly adjacency matrix operations and allows for parallelization across nodes [48], we investigate the encoding time costs required to compute node embeddings for a graph, which is typically the primary bottleneck in real-time or large-scale applications. We measured the GAT encoding time for schema graphs of various sizes. Therefore, we fixed the feature dimension at F=32 and the number of attention heads at H=4. Each measurement represents a forward pass across the full collection graph. The results are summarized in Table 4.

Table 4. Hyperparameters for fine-tune T5-base.

N	E	Runtime (ms)
3	10	0
6	14	0.11
7	16	0.11
10	25	0.11
20	50	0.15
50	100	0.84
100	125	0.96

The results show that for small graphs (N ≤ 10 collections), encoding occurs basically immediately (<0.2 ms). As the number of collections and relationships expands, the runtime increases. Notably, even for 100 collections with 125 relationships, the encoding time remains under 1 ms. The encoding process operates only on the schema graph, rather than on database records. As a result, the computational cost remains constant regardless of the quantity of documents stored. Even with increased schema configurations, the duration remained under 1 ms, demonstrating that GAT-based schema refinement is not a computational bottleneck. These findings demonstrate that QMQL is well-suited for interactive and real-time deployment, especially since schema evolution events occur significantly less often than data updates.

To evaluate the computational cost of the second component of QMQL, namely SBERT for word-sense disambiguation, we measure the retrieval latency associated with computing the query embedding and determining the most relevant schema components (the top-5 relevant schema components). SBERT calculates semantic similarity scores between the user query and the schema items after encoding them both into dense vector representations. These similarity ratings are then used to return the top-k relevant schema elements. The retrieval time is determined mostly by the number of schema components that must be semantically compared to the user query. Given a user query, we utilize the same schema sizes as provided in Table 4, and the associated latency findings are shown in Fig. 3.

As illustrated in Fig. 3, the similarity computation scales roughly linearly with the number of schema components. However, retrieval time is negligible for schema sizes N≤100. Because schema embeddings are precomputed, cached, and reused during inference.

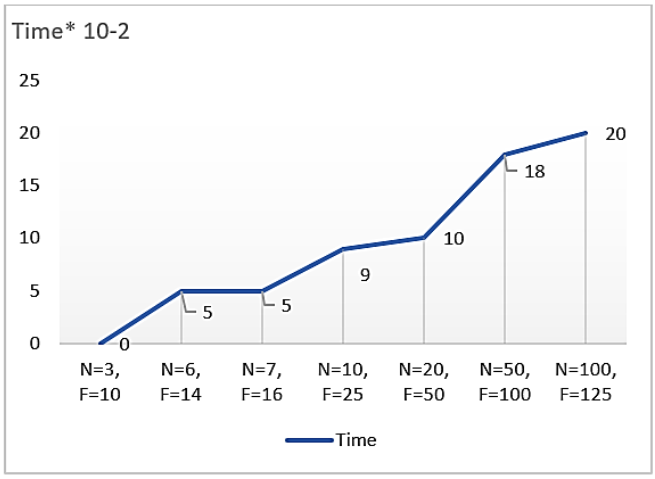

Fig. 3. SBERT retrieval across different schema sizes.

T5-base, with about 220 million parameters, achieves a balance between expressive capacity and computational efficiency. The autoregressive decoding algorithm generates the output query token by token; therefore, the generation latency is proportional to the length of the output sequence. To evaluate generation latency, we measure it as a function of the size of the relevant input schema and the complexity of the generated output query. We tested a total of 500 queries, covering a diverse range of MongoDB operations (40% Find with conditional operators, 30% aggregate with $group, $match and $project, and 30% join with $lookup). Fig. 4 showcases the results.

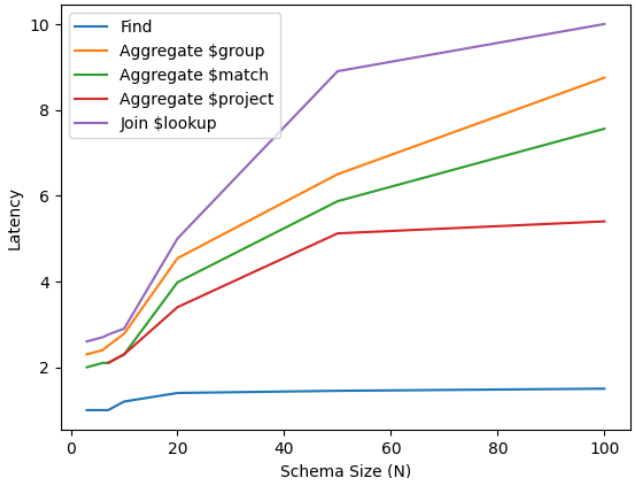

Fig. 4. Lantency comparison of MongoDB queries vs Schema size.

The computational cost experiment shows significant differences in how MongoDB operations scale as schema size expands. The Find operation is nearly consistent across all values of N, increasing only slightly from 1 to 1.5 as the schema evolves from 3 to 100 collections. Practically, this indicates that simple retrieval operations have low overhead in interactive settings. Aggregation operators: $group, $match, and $project exhibit latency that increases steadily with schema size. The evolution of their pattern is roughly linear, implying that their processing cost varies in proportion to the number of relevant schema components in the query. Among these, the $group has the largest cost, followed by $match, while the $project is rather low. This is consistent with their core execution mechanisms: $group manages accumulator structures and performs hash-based grouping; $match evaluates predicates; and $project mostly handles field-level changes. Results show that, while aggregation remains reasonable, its cost becomes more obvious as schema size increases.

The Join $lookup operator experiences the most substantial development. Its latency increases significantly beyond moderate schema sizes (e.g., N≥20). At N=100, $lookup has the highest latency

among all analyzed operations. This behavior represents the inherent complexity of cross-collection joins, which necessitate matching documents across collections, creating intermediate results, and sometimes expanding nested structures. Even under controlled settings, the join procedure has significantly more computational cost than single-collection aggregations.

At huge schema sizes, $lookup is consistently the most computationally expensive operator, followed by $group, $match, and $project, while Find is still the most efficient. This indicates that cross-collection joins are the primary cause of overhead in schema-driven query processing.

Overall, the framework performs well in retrieval and typical aggregation tasks. However, join-based queries add substantial latency and must be carefully optimized. Limiting useless $lookup operations and restricting queries to the most important schema components is critical for maintaining scalability as the schema increases.

4.4. Performance comparison

The QMQL model is evaluated by comparing the generated MongoDB query language (MQL) statements with the corresponding ground truth queries associated with each user question. Concretely, there are three types of evaluation metrics that are used to assess the model effectiveness, including exact set match accuracy (EM), execution accuracy (EX) [7], and Bilingual Evaluation Understudy (BLEU) [49].

Exact Set Match Accuracy (EM) is calculated by comparing the predicted MongoDB query $\hat{Q}$ with the ground-truth MongoDB query Q.

$$\text{Score}(\hat{Q}, Q) = \begin{cases} 1, & \hat{Q} = Q \\ 0, & \hat{Q} \neq Q \end{cases} \tag{13}$$

Formally, the EM is calculated by:

$$\text{EM} = \frac{\sum_{n=1}^{N} \text{Score}(\hat{Q}_n, Q_n)}{N} \tag{14}$$

where, N denotes the total number of samples. EM evaluates the model performance by strictly comparing differences in MongoDB queries, but humans' MQL annotations are often biased since a

Natural language questions may correspond to multiple MQL queries. Execution Accuracy (EX) is calculated by comparing the output results of executing the ground-truth MQL query and the predicted MQL query on the database contents shipped with the test set. We treat the predicted query as correct or equivalent query only if the results of executing the predicted MQL query $\hat{V}$ and the ground-truth MQL query V are same:

$$\text{Score}(\hat{V}, V) = \begin{cases} 1, & \hat{V} = V \\ 0, & \hat{V} \neq V \end{cases} \tag{15}$$

Similarly with EM, the EX is calculated by:

$$\text{EX} = \frac{\sum_{n=1}^{N} \text{Score}(\hat{V}_n, V_n)}{N} \tag{16}$$

We primarily test query generation quality using exact match accuracy (EM) and execution accuracy (EX), which directly evaluate structural correctness and semantic equivalence by ensuring that the generated query either matches the reference query exactly or yields the same execution result. However, these metrics are inherently meticulous and do not account for partially correct queries that differ in syntax but are otherwise similar to the reference formulation. As a result, we use BLEU to assess the syntactic similarity between generated and reference questions, offering a more detailed evaluation of query formulation quality. BLEU supplements EM and EX by identifying incremental improvements in query structure that are not captured by binary precision metrics.

BLEU (Bilingual Evaluation Understudy) is a structure-aware measure often used to evaluate machine translation by comparing n-grams between the output text and one or more reference texts and then computing a score based on the precision of these n-grams:

$$\text{BLEU} = \text{BP} \times \exp \sum_{n=1}^{N} (w_n \log P_n) \quad (17)$$

BP (Brevity Penalty) adjusts the score for the shorter translations. It is calculated as min(1, (reference_length/translated_length)).

The performance of QMQL is compared with four widely used Large Language Model-based generators, namely LLaMA 3.0 [50], GPT 4.0 [51], Manus AI [52], and CodeT5 [53] specifically designed for code generation tasks. The training is conducted on three subsets corresponding to distinct application domains: movies, student training, and finance, to achieve domain generalisation, which means the ability to generalise to domains that are unseen during training. The quantitative results of this evaluation are reported in Table 5.

Table 5. Results of the performance comparison.

Model	Subset of Movies			Subset of Student-training			Subset of Finance		
	EMA	EM	BLEU	EMA	EM	BLEU	EMA	EM	BLEU
LLAMA	0.63	0.56	0.42	0.59	0.53	0.40	0.55	0.48	0.48
GPT	0.70	0.64	0.48	0.67	0.70	0.71	0.60	0.59	0.46
Manus	0.78	0.85	0.81	0.78	0.84	0.81	0.72	0.79	0.80
CodeT5	0.79	0.87	0.85	0.80	0.88	0.85	0.79	0.88	0.86
QMQL	0.89	0.91	0.95	0.89	0.92	0.96	0.88	0.90	0.94

Table 5 demonstrates that QMQL consistently out performs all sate of the art LLM models across the three domains and all assessment metrics. General-purpose LLMs (LLaMA, GPT, and Manus) perform moderately well but suffer with accurate query matching, particularly in schema-sensitive domains like student training and finance. Manus consistently outperforms GPT and LLaMA in most domains and metrics, demonstrating superior skills in creating MongoDB queries; however, it falls short of QMQL, which benefits from explicit schema retrieval and task-specific fine-tuning.

CodeT5 benefits from code-driven pretraining and outperforms typical LLMs, although it lacks explicit schema grounding. In contrast, QMQL achieves the highest EM, EMA, and BLEU scores, demonstrating the efficacy of combining RAG-based schema retrieval with task-specific fine-tuning. The performance gap is more pronounced in complex domains, indicating that explicit schema information is required for accurate MongoDB query creation.

4.5. Discussion

The studies show how the proposed QMQL model generates accurate MongoDB queries across multiple domains. The initial examination of the RAG framework highlighted the importance of selecting the most relevant Top-K schema components. Using small K values (e.g., K = 1 or K = 2) resulted in low precision and recall, indicating insufficient schema coverage. Increasing K beyond 5 somewhat affected precision due to noise. The best value, K=5, strikes a compromise between relevance and coverage, ensuring the query generator obtains sufficient context.

A key factor contributing to the effectiveness of the QMQL model is its use of SBERT to integrate user questions and schema elements. Ambiguity in natural language inquiries sometimes arises when many schema components share the same name or when relationships between collections are implicit. QMQL tackles this issue by combining graph-based schema encoding with dense semantic retrieval. The schema graph G=(N,E) represents the structure of a document-oriented database. GAT embeddings spread information across connected nodes, allowing the model to encode both a collection's local fields and its context in the graph. This distinguishes identical field names across collections and clarifies relational dependencies. BERT embeddings of questions and schema components serve as a semantic filter, selecting only the top-K relevant schema elements. This ensures that the generating module

depends solely on semantically aligned components, hence eliminating the lexical ambiguity inherent in natural language.

The combination of structural context (by GAT) and semantic relevance (via SBERT) mitigates structural and lexical ambiguity, allowing QMQL to generate precise and contextually grounded queries. Empirical results show that this technique outperforms models based only on code-pretrained generation, especially in schema drift or domain shift scenarios.

The evaluation, based on exact match accuracy (EM), execution accuracy (EX), and BLEU score, shows that QMQL consistently outperforms the state-of-the-art models across all domains. General-purpose linear models (LLaMA, GPT, and MANUS) achieve satisfactory results but struggle with exact query matching in schema-sensitive domains such as student education and finance. Among them, Manus consistently outperforms GPT and LLaMA, demonstrating its superior ability to generate structured queries. CodeT5, with its code-oriented pre-training, outperforms the classic linear models but still lacks explicit schema integration. QMQL achieves the highest EM, EX, and BLEU scores across all domains, demonstrating the high efficiency of RAG-based schema retrieval, combined with task-specific fine-tuning, for generating Text-to-MQL queries. The advantage of QMQL is more pronounced in complex domains, confirming that explicit schema consideration is essential for generating accurate and executable MongoDB queries. The computational costs of the proposed QMQL framework are heavily influenced by its three major components: Graph Attention Networks (GAT), Sentence-BERT (SBERT), and T5. Each has unique computational properties that affect the system's overall efficiency.

In reality, the QMQL framework achieves a compromise between both components by restricting the number of relevant schema elements (top-K selection) and constraining generated query complexity. The GAT component scales with schema size, SBERT with the textual length of the question and schema, and T5 with generated query length. As a result, the system's total latency can be successfully reduced by carefully selecting K, restricting input sequence lengths, and optimizing generation length, all while maintaining high efficiency in schema grounding and query formulation. These results validate the robustness of QMQL in heterogeneous domains and its ability to generalize beyond specific training contexts.

5. Conclusion

This work presents QMQL, a schema-aware text-to-MongoDB query generation framework that integrates SBERT-based semantic search with a RAG-enhanced T5 generator. By explicitly extracting the most relevant schema elements and conditioning query generation on this structured context, QMQL effectively handles semantic ambiguity and schema flexibility in NoSQL databases. Experimental results obtained in three different domains demonstrate that QMQL consistently outperforms leading language models and code-oriented reference methods in terms of exact match accuracy, execution accuracy, and BLEU score. The analysis also confirms that selecting an optimal set of Top-K schemas and applying task-specific fine-tuning are key factors in ensuring high query accuracy and robustness.

Future work will explore extending this approach to other NoSQL databases, as well as incorporating multilingual semantic resources and addressing complex structures to further enhance generalization and interpretability. Exploring fine-tuning techniques and optimized parameter adaptation could reduce training costs while maintaining performance. Finally, integrating user feedback and reinforcement learning could improve query accuracy and personalization in real-world deployments.

Acknowledgment

The author would like to think Mr Zakaria Mihoubi for his valuable assistance in the building and extension of the dataset.

Declarations

Author contribution. The author confirms they responsible for all phases of this contribution.

Funding statement. This research received no external funding.
Conflict of interest. The authors declare no conflict of interest.
Additional information. No additional information is available for this paper.

References

[1] M. Rathore and S. S. Bagui, "MongoDB: Meeting the Dynamic Needs of Modern Applications," *Encyclopedia*, vol. 4, no. 4, pp. 1433–1453, Sep. 2024, doi: 10.3390/encyclopedia4040093.

[2] React Query Builder, "React Query Builder Documentation." [Online]. Available: https://react-querybuilder.js.org/.

[3] M. Al Khatib, A. Alshammari, and M. Al-Rakhami, "Rethinking Concept Drift Detection in Data Streams: A Large Language Model-Based Adaptive Framework," in *Proceedings of the LSGDA Workshop at VLDB 2024*, 2024, pp. 81–90. [Online]. Available: https://vldb.org/workshops/2024/proceedings/LSGDA/LSGDA24.09.

[4] J. Qi *et al.*, "RASAT: Integrating Relational Structures into Pretrained Seq2Seq Model for Text-to-SQL," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2022, pp. 3215–3229. doi: 10.18653/v1/2022.emnlp-main.211.

[5] L. Shi, Z. Tang, N. Zhang, X. Zhang, and Z. Yang, "A Survey on Employing Large Language Models for Text-to-SQL Tasks," *ACM Comput. Surv.*, vol. 58, no. 2, pp. 1–37, Jan. 2026, doi: 10.1145/3737873.

[6] M. Pourreza *et al.*, "CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL," Oct. 2024, pp. 1–30. Accessed: May 31, 2026. [Online]. Available: http://arxiv.org/abs/2410.01943.

[7] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," in *Advances in Neural Information Processing Systems 36*, San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023, pp. 36339–36348. doi: 10.52202/075280-1577.

[8] Q. Li, T. You, J. Chen, Y. Zhang, and C. Du, "LI-EMRSQL: Linking Information Enhanced Text2SQL Parsing on Complex Electronic Medical Records," *IEEE Trans. Reliab.*, vol. 73, no. 2, pp. 1280–1290, Jun. 2024, doi: 10.1109/TR.2023.3336330.

[9] G. Jeong *et al.*, "Improving Text-to-SQL with a Hybrid Decoding Method," *Entropy*, vol. 25, no. 3, p. 513, Mar. 2023, doi: 10.3390/e25030513.

[10] W. Zhang, K. Zeng, X. Yang, T. Shi, and P. Wang, "Text-to-ESQ: A Two-Stage Controllable Approach for Efficient Retrieval of Vaccine Adverse Events from NoSQL Database," in *Proceedings of the 14th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, New York, NY, USA: ACM, Sep. 2023, pp. 1–10. doi: 10.1145/3584371.3613008.

[11] S. Benhur, N. Nayan, P. Manoharan, A. Venugopalan, K. S. Rajabhupati, and N. Sundaramahalingam, "NL2EQ: Generating Elasticsearch Query DSL from Natural Language Text Using Large Language Models," in *Lecture Notes in Networks and Systems*, vol. 1424 LNNS, Springer Science and Business Media Deutschland GmbH, 2025, pp. 223–242. doi: 10.1007/978-3-031-92605-1_15.

[12] K. M. Hossen, M. N. Uddin, M. Arefin, and M. A. Uddin, "BERT Model-based Natural Language to NoSQL Query Conversion using Deep Learning Approach," *Int. J. Adv. Comput. Sci. Appl.*, vol. 14, no. 2, pp. 810–821, Feb. 2023, doi: 10.14569/IJACSA.2023.0140293.

[13] M. Hornsteiner, M. Kreussel, C. Steindl, F. Ebner, P. Empl, and S. Schönig, "Real-Time Text-to-Cypher Query Generation with Large Language Models for Graph Databases," *Futur. Internet*, vol. 16, no. 12, p. 438, Nov. 2024, doi: 10.3390/fi16120438.

[14] M. Kobeissi, N. Assy, W. Gaaloul, B. Defude, B. Benatallah, and B. Haidar, "Natural language querying of process execution data," *Inf. Syst.*, vol. 116, no. June, p. 102227, Jun. 2023, doi: 10.1016/j.is.2023.102227.

[15] J. Lu, Y. Song, Z. Qin, H. Zhang, C. Zhang, and R. C.-W. Wong, "Bridging the Gap: Enabling Natural Language Queries for NoSQL Databases through Text-to-NoSQL Translation," pp. 1–16, Feb. 2025, Accessed: May 31, 2026. [Online]. Available: http://arxiv.org/abs/2502.11201.

[16] Z. Qin, Y. Song, J. Lu, Y. Song, S. Li, and C. J. Zhang, "MultiTEND: A Multilingual Benchmark for Natural Language to NoSQL Query Translation," in *Findings of the Association for Computational Linguistics: ACL 2025*, Stroudsburg, PA, USA: Association for Computational Linguistics, Aug. 2025, pp. 24632–24657. doi: 10.18653/v1/2025.findings-acl.1265.

[17] P. Do, T. H. V. Phan, and B. B. Gupta, "Developing a Vietnamese Tourism Question Answering System Using Knowledge Graph and Deep Learning," *ACM Trans. Asian Low-Resource Lang. Inf. Process.*, vol. 20, no. 5, pp. 1–18, Sep. 2021, doi: 10.1145/3453651.

[18] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021, doi: 10.1109/TNNLS.2020.2978386.

[19] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, Neural information processing systems foundation, May 2020, pp. 9459–9474. Accessed: May 31, 2026. [Online]. Available: https://arxiv.org/pdf/2005.11401

[20] C. Raffel *et al.*, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, Sep. 2023, Accessed: May 31, 2026. [Online]. Available: http://arxiv.org/abs/1910.10683

[21] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2019, pp. 3980–3990. doi: 10.18653/v1/D19-1410.

[22] A. Aggoune and Z. Mihoubi, "Towards Efficient Dataset Development: A Case Study of M2Q2+ in Movie QA Systems," in *2024 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, IEEE, Nov. 2024, pp. 1–5. doi: 10.1109/ICAASE64542.2024.10850962.

[23] Mukesh Reddy Dhanagari, "MongoDB and Data Consistency: Bridging the Gap between Performance and Reliability," *J. Comput. Sci. Technol. Stud.*, vol. 6, no. 2, pp. 183–198, Jun. 2024, doi: 10.32996/jcsts.2024.6.2.21.

[24] C. M. Sowandi, I. Leonita, S. Hartanto, P. Arisaputra, and D. David, "Exploring the Effectiveness, Features, and Compatibility of MongoDB and MySQL: A Comprehensive Comparison of NoSQL and Relational Databases," *MIND (Multimedia Artif. Intell. Netw. Database) J.*, vol. 8, no. 2, pp. 217–229, Dec. 2023, Accessed: May 31, 2026. [Online]. Available: https://ejurnal.itenas.ac.id/index.php/mindjournal/article/view/9145.

[25] M. Nuriev, R. Zaripova, O. Yanova, I. Koshkina, and A. Chupaev, "Enhancing MongoDB query performance through index optimization," *E3S Web Conf.*, vol. 531, p. 03022, Jun. 2024, doi: 10.1051/e3sconf/202453103022.

[26] MongoDB Inc., "Sample Mflix Dataset." Accessed: Jun. 04, 2026. [Online]. Available: https://www.mongodb.com/docs/atlas/sample-data/sample-mflix/.

[27] MongoDB Inc., "Atlas Architecture Center." Accessed: May 31, 2026. [Online]. Available: https://www.mongodb.com/docs/atlas/architecture/current/.

[28] N. Bansal, S. Sachdeva, and L. K. Awasthi, "Query-based denormalization using hypergraph (QBDNH): a schema transformation model for migrating relational to NoSQL databases," *Knowl. Inf. Syst.*, vol. 66, no. 1, pp. 681–722, Jan. 2024, doi: 10.1007/s10115-023-02017-y.

[29] S. Bharany, K. Kaur, S. E. M. Eltaher, A. O. Ibrahim, S. Sharma, and M. M. M. A. Elsalam, "A Comparative Study of Cloud Data Portability Frameworks for Analyzing Object to NoSQL Database Mapping from ONDM's Perspective," *Int. J. Adv. Comput. Sci. Appl.*, vol. 14, no. 10, pp. 805–814, Oct. 2023, doi: 10.14569/IJACSA.2023.0141086.

[30] A. Samydurai, K. Revathi, L. Karthikeyan, B. Vanathi, and K. Devi, "An Enhanced Entity Model for Converting Relational to Non-Relational Documents in Hospital Management System Based on Cloud Computing," *IETE Tech. Rev.*, vol. 39, no. 6, pp. 1449–1462, Nov. 2022, doi: 10.1080/02564602.2021.2016075.

[31] A. Aggoune and M. S. Namoune, "P3 Process for Object-Relational Data Migration to NoSQL Document-Oriented Datastore," *Int. J. Softw. Sci. Comput. Intell.*, vol. 14, no. 1, pp. 1–20, Sep. 2022, doi: 10.4018/IJSSCI.309994.

[32] A. Aggoune and M. S. Namoune, "Metadata-driven Data Migration from Object-relational Database to NoSQL Document-oriented Database," *Comput. Sci.*, vol. 23, no. 4, pp. 495–519, Nov. 2022, doi: 10.7494/csci.2022.23.4.4375.

[33] MongoDB Inc., "Sample Datasets." [Online]. Available: https://www.mongodb.com/docs/atlas/sample-data/.

[34] A. G. Vrahatis, K. Lazaros, and S. Kotsiantis, "Graph Attention Networks: A Comprehensive Review of Methods and Applications," *Futur. Internet*, vol. 16, no. 9, p. 318, Sep. 2024, doi: 10.3390/fi16090318.

[35] J. Li *et al.*, "Graphix-T5: Mixing Pre-trained Transformers with Graph-Aware Layers for Text-to-SQL Parsing," *Proc. AAAI Conf. Artif. Intell.*, vol. 37, no. 11, pp. 13076–13084, Jun. 2023, doi: 10.1609/aaai.v37i11.26536.

[36] Y. Yang, Z. Peng, F. Zhou, X. Yao, and Y. Zhang, "Conversational Text-to-SQL: A Comprehensive Survey of Paradigms, Challenges, and Future Directions," in *Communications in Computer and Information Science*, vol. 2700 CCIS, Springer Science and Business Media Deutschland GmbH, 2025, pp. 1–19. doi: 10.1007/978-3-032-08049-3_1.

[37] [W. Zhao, L. Zhao, F. Wu, Z. Zheng, H. Jin, and B. Gu, "Enhancing Interaction Graph of Data Schema and Syntactic Structure with Pre-trained Language Model for Text-to-SQL," in *Communications in Computer and Information Science*, vol. 2301, Springer Science and Business Media Deutschland GmbH, 2025, pp. 159–173. doi: 10.1007/978-981-96-1024-2_12.

[38] P. Lekheshwar Balley and S. V. Sonekar, "Design of an Integrated Model Using R-GCN, TPOT, and Transformers for Efficient NoSQL Data Processing and Analysis," *J. Neonatal Surg.*, vol. 14, no. 6S, pp. 298–314, Mar. 2025, doi: 10.52783/jns.v14.2237.

[39] D. Thakur, J. K. Saini, and S. Srinivasan, "DeepThink IoT: The Strength of Deep Learning in Internet of Things," *Artif. Intell. Rev.*, vol. 56, no. 12, pp. 14663–14730, Dec. 2023, doi: 10.1007/s10462-023-10513-4.

[40] H. Han *et al.*, "Retrieval-Augmented Generation with Graphs (GraphRAG)," pp. 1–88, Jan. 2025, Accessed: Jun. 04, 2026. [Online]. Available: https://arxiv.org/pdf/2501.00309.

[41] Y. Gao, Y. Xiong, M. Wang, and H. Wang, "Modular RAG: Transforming RAG Systems into LEGO-like Reconfigurable Frameworks," pp. 1–17, Jul. 2024, Accessed: Jun. 04, 2026. [Online]. Available: http://arxiv.org/abs/2407.21059.

[42] W. X. Zhao, J. Liu, R. Ren, and J.-R. Wen, "Dense Text Retrieval Based on Pretrained Language Models: A Survey," *ACM Trans. Inf. Syst.*, vol. 42, no. 4, pp. 1–60, Jul. 2024, doi: 10.1145/3637870.

[43] E. Eldele *et al.*, "Self-Supervised Contrastive Representation Learning for Semi-Supervised Time-Series Classification," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 12, pp. 15604–15618, Dec. 2023, doi: 10.1109/TPAMI.2023.3308189.

[44] E. Daraghmi, L. Atwe, and A. Jaber, "A Comparative Study of PEGASUS, BART, and T5 for Text Summarization Across Diverse Datasets," *Futur. Internet*, vol. 17, no. 9, p. 389, Aug. 2025, doi: 10.3390/fi17090389.

[45] S. S. Gu, Y. Iwasawa, T. Kojima, Y. Matsuo, and M. Reid, "Large Language Models Are Zero-Shot Reasoners," in *Advances in Neural Information Processing Systems 35*, San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2022, pp. 22199–22213. doi: 10.52202/068431-1613.

[46] A. Aggoune, "A Survey on Dataset Development Techniques for QA Systems ⋆," in *A Comparative Study of Retrieval-Augmented Generation Methods in Enterprise Knowledge Management*, 2024, pp. 1–12. Accessed: Jun. 04, 2026. [Online]. Available: https://ceur-ws.org/Vol-3922/paper1.

[47] A. Aggoune and Z. Mihoubi, "M2Q2: A Text-to-MQL Dataset for Movie QA Systems," in *Lecture Notes in Networks and Systems*, vol. 1393 LNNS, Springer Science and Business Media Deutschland GmbH, 2026, pp. 441–453. doi: 10.1007/978-3-031-90893-4_29.

[48] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," in *International Conference on Learning Representations (ICLR)*, International Conference on Learning Representations, ICLR, Feb. 2018, pp. 1–12. Accessed: Jun. 04, 2026. [Online]. Available: http://arxiv.org/abs/1710.10903.

[49] G. Datta, N. Joshi, and K. Gupta, "Analysis of Automatic Evaluation Metric on Low-Resourced Language: BERTScore vs BLEU Score," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 13721 LNAI, Springer Science and Business Media Deutschland GmbH, 2022, pp. 155–162. doi: 10.1007/978-3-031-20980-2_14.

[50] A. Grattafiori *et al.*, "The Llama 3 Herd of Models," pp. 1–92, Nov. 2024, Accessed: Jun. 04, 2026. [Online]. Available: http://arxiv.org/abs/2407.21783.

[51] A. Taloni *et al.*, "Comparative performance of humans versus GPT-4.0 and GPT-3.5 in the self-assessment program of American Academy of Ophthalmology," *Sci. Rep.*, vol. 13, no. 1, p. 18562, Oct. 2023, doi: 10.1038/s41598-023-45837-2.

[52] M. Shen, Y. Li, L. Chen, Z. Fan, Y. Li, and Q. Yang, "From Mind to Machine: The Rise of Manus AI as a Fully Autonomous Digital Agent," pp. 1–21, Mar. 2026, Accessed: Jun. 04, 2026. [Online]. Available: http://arxiv.org/abs/2505.02024.

[53] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2021, pp. 8696–8708. doi: 10.18653/v1/2021.emnlp-main.685.