Neuro-Symbolic Query Compiler

Yuyao Zhang[1], **Zhicheng Dou**[1]*, **Xiaoxi Li**[1], **Jiajie Jin**[1]
Yongkang Wu[3], **Zhonghua Li**[3], **Qi Ye**[3] and **Ji-Rong Wen**[1,2]
[1]Gaoling School of Artificial Intelligence, Renmin University of China
[2]Beijing Key Laboratory of Research on Large Models and Intelligent Governance
[3]Huawei Poisson Lab
{2020201710, dou}@ruc.edu.cn

Abstract

Precise recognition of search intent in Retrieval-Augmented Generation (RAG) systems remains a challenging goal, especially under resource constraints and for complex queries with nested structures and dependencies. This paper presents **QCompiler**, a neuro-symbolic framework inspired by linguistic grammar rules and compiler design, to bridge this gap. It theoretically designs a minimal yet sufficient Backus-Naur Form (BNF) grammar $G[q]$ to formalize complex queries. Unlike previous methods, this grammar maintains completeness while minimizing redundancy. Based on this, **QCompiler** includes a Query Expression Translator, a Lexical Syntax Parser, and a Recursive Descent Processor to compile queries into Abstract Syntax Trees (ASTs) for execution. The atomicity of the sub-queries in the leaf nodes ensures more precise document retrieval and response generation, significantly improving the RAG system's ability to address complex queries.

1 Introduction

In the field of cognitive science, the human brain demonstrates a sophisticated synergy between two cognitive systems (Hitzler et al., 2022): On the one hand, through **neural-networks-based computation**, humans can rapidly process information from complex sensory inputs. On the other hand, with **symbolic-system-based logical reasoning**, humans can analyze abstract rules such as language, mathematics, and causality, performing calculations and inferences through symbols and rules. These two systems complement each other, enabling humans to flexibly handle a range of complex tasks from perception to reasoning, exhibiting a powerful generalization capability that cannot be achieved by a single mechanism alone.

In the field of Artificial Intelligence, Artificial Neural Networks (ANNs) have shown strong fitting capabilities but struggle to extrapolate and generalize in a world of constantly updated knowledge (Hasson et al., 2020). Recently, **R**etrieval-**A**ugmented **G**eneration (RAG) technique addresses this limitation to some extent by introducing a retrieval process that allows ANNs to access external knowledge beyond their training data (Lewis et al., 2020; Zhao et al., 2024). This enhancement improves the ability of neural networks to process information in open domains and real-world scenarios. However, this improvement has a limited upper bound: as user queries become more complex or require reasoning, the chance of retrieving all relevant documents at once decreases significantly, leading to poor performance of RAG systems, as shown in Figure 1(a).

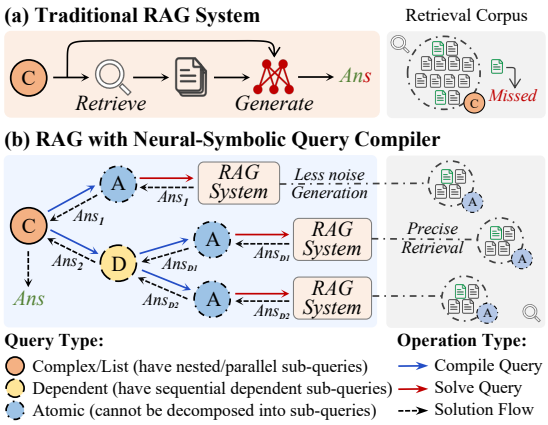

Figure 1: Illustration of how **QCompiler** enhances the RAG system.

The handling of complex user queries for ANNs presents a significant challenge: These queries often contain **implicit intents**, **nested logical structures**, and **intricate dependencies**, making it difficult to arrive at answers in a single step. For example, the query "I want to find an introduction and reviews of J.K. Rowling's most popular book and check if the local library has it" may require the system to use ANNs to extract key information, while relying on symbolic rules for task decompo-

*Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2025, pages 12138–12155

sition and reasoning—enabling multi-turn interaction with both databases and users. The system cannot execute other queries without first identifying which is the most popular book. This naturally raises the first question: **How can we effectively process complex queries to recognize search intent precisely?**

A straightforward idea is to leverage the powerful ANNs' capabilities to achieve this in an end-to-end manner (Ma et al., 2023; Asai et al., 2023; Chan et al., 2024). Unlike the human brain, however, ANNs often struggle with tasks that require **explicit symbolic reasoning**. This is in part because they may not encounter sufficient data related to specific symbolic reasoning, leading to implicit reasoning modeling, making it difficult to handle complex queries that require explicit analysis and decomposition. This raises a second question: **How can we replicate the synergy of two cognitive systems in human brain, neural computation and symbolic reasoning, to effectively address complex queries in real-world scenarios?**

Many studies have investigated approaches for rewriting, disambiguating, and decomposing complex queries using symbolic and heuristic rules (Min et al., 2019; Khot et al., 2020; Chan et al., 2024). We can regard complex queries as a Domain Specific Language (DSL) generated by a certain complete grammar G (Wang et al., 2024), and such approaches can be viewed as selecting implicitly or explicitly a subset G' of grammar G (denoted as $G' \subseteq G$) to generate queries. There has also been research on the use of LLMs to handle complex queries, with iterative and agentic RAG systems showing strong problem planning & solving capabilities (Verma et al., 2024; Chen et al., 2024).

However, these methods face two main issues: (1) Although the processing of complex queries may cover all implied intentions, from the perspective of grammar-generated languages, this coverage may not be minimal (Wolfson et al., 2020), resulting in unnecessary complexity and resource waste. (2)LLM-based iterative and agentic RAG systems rely heavily on their performance. This reliance often worsens the trade-off between performance and efficiency: achieving high performance typically requires redundant retrievals or frequent API calls for multi-round iterations, leading to higher computational costs and increased latency.

To address these questions, we propose **QCompiler**, inspired by grammars in linguistics and compiler theory. We first summarize four query types and design a Backus-Naur Form (BNF) grammar $G[q] \subset G' \subseteq G$ that is minimally sufficient to formalize these queries. Then we train a small language model to translate natural-language queries into these BNF-based expressions, which can be parsed into an Abstract Syntax Tree (AST). The grammar G[q] constrains the search space during language model generation, while the parsed abstract syntax tree (AST) enables symbolic reasoning by recursively applying grammar rules at each node. This process efficiently handles nested structures and dependencies of complex queries, resembling a compiler that can translate high-level code into machine-executable instructions.

This framework can be seamlessly customized into the existing RAG system for better query understanding because: (1) The parsed AST can capture the implicit search intents, nested structure, and dependencies of the original complex query. (2) The sub-queries in leaf nodes are more precise because they represent well-defined atomic subqueries, reducing ambiguity and targeting specific intent for accurate document retrieval and generation, as shown in Figure 1(b). (3) It allows developers in production environments to verify the correctness of each sub-query node and, when necessary, modify or intervene in the reasoning process. The results across four multi-hop benchmarks show QCompiler's remarkable understanding capability of complex queries.

Our contributions in this paper are as follows:

1. We theoretically propose a minimal yet sufficient Backus-Naur form (BNF) grammar $G[q]$ to formalize complex queries and provide a foundation for precise search intent recognition.

2. We propose **QCompiler**, which synergizes neural computation and symbolic reasoning to compile complex queries into Abstract Syntax Trees, capturing their nested structures and dependencies.

3. We experimentally demonstrate the accuracy of this tree structure representation in expressing complex queries, which improves efficiency and accuracy by retrieving fewer but more precise documents and providing more accurate responses.

2 Related Work

2.1 Query Understanding

Query understanding involves rewriting, disambiguating, decomposing, and expanding the orig-

inal query methods (Ma et al., 2023; Min et al., 2019; Khot et al., 2020; Asai et al., 2023; Chan et al., 2024). Approaches to handling complex multi-hop queries focus mainly on the decomposition rule (Min et al., 2019; Wolfson et al., 2020).To enhance the ability of Question Answering, researchers focus on constructing benchmarks for complex multi-hop questions, either manually or using symbolic rules, to evaluate existing systems (Yang et al., 2018; Ho et al., 2020; Trivedi et al., 2022b). The main solutions currently are mainly focused on the use of relevant documents and parametric knowledge of LLMs to process queries (Asai et al., 2023; Trivedi et al., 2022a; Shao et al., 2023; Chan et al., 2024). QCompiler integrates neural networks and reasoning based on symbolic rules. It naturally includes the aforementioned aspects of query understanding during the parsing and execution of complex queries.

2.2 Neuro-Symbolic AI & Grammar Rule

Neuro-symbolic AI combines the computational power of neural networks with the precise rule-based reasoning of symbolic systems (Hitzler et al., 2022; Olausson et al., 2023; Ahmed et al., 2022; Dinu et al., 2024). This paradigm has shown significant potential in the treatment of complex and even long-tail tasks (Gupta and Kembhavi, 2023; Yao et al., 2022; Schick et al., 2024; Shen et al., 2024). Recent studies highlight the benefits of parameterizing symbolic rules, such as grammars, to enhance ANN's symbolic reasoning capabilities in specific domains (Dinu et al., 2024; Wang et al., 2024). Drawing inspiration from lexical and syntactic analysis in context-free grammars and compiler design (Wang et al., 2024; Aho and Ullman, 1969; Alfred et al., 2007), QCompiler introduces a parameterized grammar-based model to compile complex queries. This approach progressively translates queries into intermediate expressions and parses them into ASTs, achieving efficient parsing, reasoning, and inference.

2.3 Retrieval-Augmented Generation System

RAG systems integrate retrieval and generative models to access external knowledge during inference (Lewis et al., 2020; Zhao et al., 2024; Guan et al., 2024), overcoming the static nature of parametric models. More advanced approaches incorporate modules for query rewriting (Ma et al., 2023), retrieval necessity determination (Tan et al., 2024; Jiang et al., 2023b), document re-ranking, refinement (Jiang et al., 2023a; Li et al., 2023), and others (Jin et al., 2024). Iterative and agentic RAG systems can also perform tree-structured (Chan et al., 2024) or graph-structured (Chen et al., 2024; Li et al., 2024) reasoning during processing by leveraging mechanisms such as reflection (Asai et al., 2023) and planning (Trivedi et al., 2022a; Shao et al., 2023). However, challenges such as redundancy, high computational costs, and reliance on the performance of the base model persist.

3 Methodology

In this work, we focus on the understanding and representation of complex queries. We give an overview of **QCompiler** and how it works in Figure 2, and details of grammar, components used to implement each step, training, and inference in the following sections.

3.1 Mathematical Definition of Query Types

To effectively process complex queries and capture user intentions, we conceptualize different queries as four basic types: ***Atomic Query***, ***Dependent Query***, ***List Query***, and ***Complex Query***. Let Q be the set of all possible queries and:

DEFINITION 1

<***Atomic Query***> A query $q \in Q$ is called an **atomic query** if it cannot be decomposed into a combination of sub-queries. For $q, \forall q_1, \cdots q_n \in Q$, s.t.

$$q \neq q_1 \circ q_2 \circ \cdots \circ q_n$$

where $\circ$ denotes a query composition operation.

In other words, an atomic query is an indivisible single-turn query that cannot be further decomposed. It requires no external context or dependencies for its execution, such as "*Who is the director of The Titanic?*".

DEFINITION 2

<***Dependent Query***> A query q is called a **dependent query** if it can be decomposed into two parts. For $q, \exists\ q_1, q_2 \in Q$, s.t.

$$q = q_1 \circ q_2, \text{and } q_2 = f(q_1)$$

where f is a function that maps the result of q_1 to q_2.

A dependent query consists of two parts, with the latter part's execution reliant on the results of the former part and cannot be executed in parallel, such

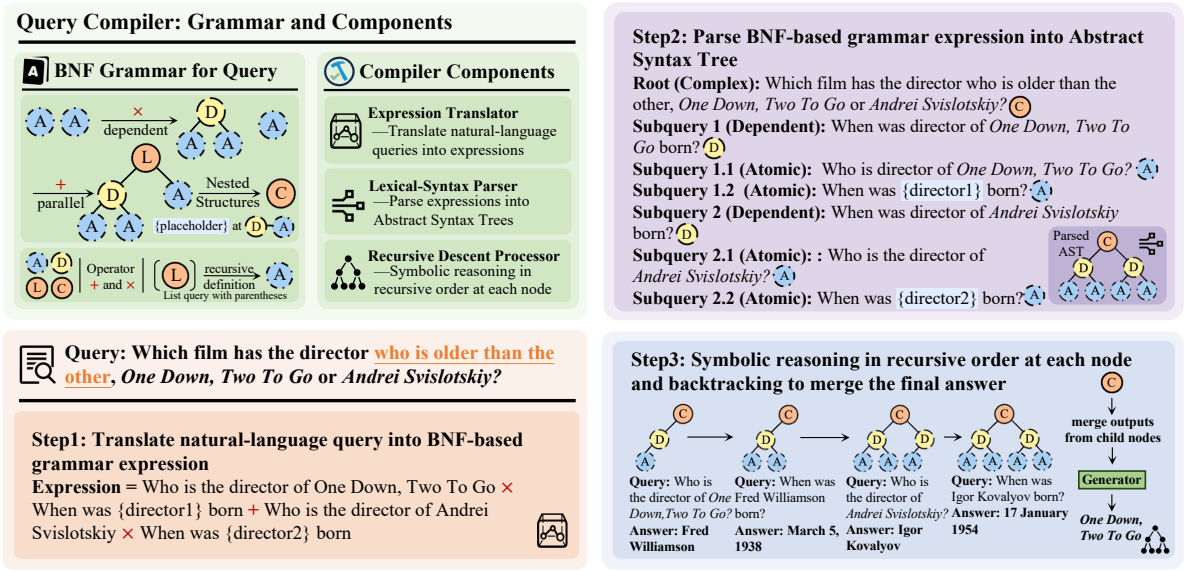

Figure 2: Illustration of **QCompiler**, including the grammars, components, and an example of processing complex queries with QCompiler.

as "*When was the director of The Titanic born?*" containing two queries with dependencies: "*Who is the director of The Titanic?*" and "*When was James Cameron born?*".

DEFINITION 3

<***List Query***> A query q is called a **list query** if it can be decomposed into multiple independent sub-queries. For $q, \exists\, q_1, \cdots q_n \in Q$, s.t.

$$q = q_1 \circ q_2 \circ \cdots \circ q_n$$

and for $\forall i \neq j,\ q_i \neq f(q_j)$.

Thus, a list query consists of multiple parts that are independent and can be executed in parallel to accelerate the inference of a whole system, such as "*Who is older, James Cameron or Steven Allan Spielberg?*" containing two queries without dependencies: "*When was James Cameron born*" and "*When was Steven Allan Spielberg born*".

DEFINITION 4

<***Complex Query***> A query is called a **complex query** if it can be decomposed into multiple sub-queries, which may include atomic, dependent, and list queries. For $q, \exists\, q_1, q_2, \ldots, q_n \in Q$, s.t.

$$q = q_1 \circ q_2 \circ \cdots \circ q_n$$

and for at least one pair (i, j), st. $q_i = f(q_j)$.

Thus, a complex query involves combining multiple types of queries with nested logical structure and intricate dependencies, forming a structured query that cannot be classified solely as an atomic query, dependent query, or list query, such as *Who is older, the director of The Titanic or Steven Allan Spielberg?* contains a dependent query and an atomic query without dependencies.

Under this definition, all queries in real-world scenarios can be described and formalized.

3.2 Backus-Naur Form (BNF) Grammar

With the query definition mentioned above, in this section, we present the specialized Backus-Naur Form (BNF) grammar for complex queries.

BNF is a context-free grammar widely used to precisely describe syntax rules in programming languages, protocols, and Domain-Specific Languages (DSLs) (Aho and Ullman, 1969; Alfred et al., 2007; Wang et al., 2024). Following these works, a BNF grammar G is typically defined as a 4-tuple $G = (N, T, P, S)$, where:

(1) N is a set of non-terminal symbols that represent intermediate structures in grammar.

(2) T is a set of terminal symbols, representing concrete characters or tokens in the language, with $N \cap T = \emptyset$.

(3) $S \in N$ is the start symbol.

(4) P is a set of production rules, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$. Each production rule $A \rightarrow \alpha$ is often written as: $\langle A \rangle ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, where $\alpha_i \in (N \cup T)^*$ represents possible expansions of A.

Then we define the BNF grammar for complex queries according to this paradigm as follows:

Non-terminal symbols N. Non-terminals denote intermediate structures of grammar, abbreviated as <***Atomic***>, <***List***>, <***Dependent***>, <***Complex***>.

Terminal symbols T. It is divided into two subsets: the atomic query set $Q_{\text{atomic}} \subset Q$ and the operator set O. Each $q \in Q_{\text{atomic}}$ represents an atomic query string, such as "*Who is the director of The Titanic?*". The operator set O contains '+' and '×'. '+' connects two independent queries, allowing them to be answered in parallel; × connects two queries in a dependent relationship, where the latter query relies on the result of the former query.

Start Symbol S. For every query $q \in Q$, the start symbol corresponds to a <***Complex***>, which serves as the root of the grammar.

Production Rules P. Together with the definition in Section 3.1, we further refer to BNF to define the production rules for complex queries.

PRODUCTION RULES OF QUERY

<***Atomic***>::= $q \in Q_{atomic}$ | (<***List***>)
<***Dependent***>::=<***Atomic***>|<***Dependent***>×<***Atomic***>
<***List***>::=<***Dependent***>|<***List***>+<***Dependent***>
<***Complex***>::= <***List***>

In production rules, the operator '×' is assigned a higher precedence than the operator '+' to ensure that the parsing process is deterministic and unambiguous. Furthermore, we use parentheses for grouping and priority control, and the **expressions enclosed in parentheses can also be considered as a production rule for atomic queries**. This recursive definition is similar to those found in many programming languages and general-purpose grammars, allowing nested queries to be formalized naturally without complicating the grammar or introducing additional non-terminal variables.

We provide a proof of the completeness and minimality of grammar $G[q]$ in Appendices A and B.

3.3 Key Components of QCompiler

With the queries and their BNF grammar defined, the next step is to process complex queries. Drawing inspiration from compiler design, we develop a compiler instance that integrates three key components: a Query Expression Translator, a Lexical-Syntax Parser, and a Recursive Descent Processor.

Query Expression Translator. This component uses a language model to translate natural language queries into BNF-based expressions. It ensures that the user's search intents are precisely captured and represented. To achieve this, we train a language model to specifically translate natural language queries into BNF-based expressions, as illustrated in Figure 2 Step1.

Lexical-Syntax Parser. This component performs symbolic reasoning on query expressions, using tokens from lexical analysis to construct an Abstract Syntax Tree (AST) based on the BNF grammar. The AST represents the structure of the query, capturing nested structure and dependency, and serves as a bridge between the representation of high-level query and the downstream reasoning processes, as illustrated in Figure 2 Step2.

Recursive Descent Processor. This component interprets AST recursively, executing the subqueries by resolving their dependencies and performing placeholder replacements. It manages the data flow between different query nodes, handling execution of sub-queries in the AST, as illustrated in Figure 2 Step3.

3.4 Training of Query Expression Translator

To enable the language model to understand the grammar and respond in the desired format, we optimize it using the following objective function:

$$\mathcal{L} = -\sum_{\langle q_i, e_i \rangle \in \mathcal{D}_T} \log P(e_i \mid q_i, G[q]), \quad (1)$$

where $\mathcal{D}_T$ represents the training dataset consisting of query-expression pairs $\langle q_i, e_i \rangle$, q_i is the input query, e_i is the corresponding expression following the grammar $G[q]$ and $G[q]$ denotes the BNF grammar instruction. The construction of training data is detailed in Appendix F. The trained model can translate the original query into BNF-expression.

3.5 Validation and Inference

The trained Query Expression Translator may still generate invalid expressions, leading to the construction of invalid ASTs. These problems can be categorized into two types: (1) **Erroneous Dependency**: Placeholder content appears in query nodes without the corresponding dependencies. (2) **Missing Dependency**: Query nodes with dependencies lack the necessary placeholders to extract the required information.

To address these issues, we designed a recursive validation algorithm based on Depth-First Search (DFS) to check the legality of ASTs, which is illustrated in Appendix D. During inference, we sample

Methods	2Wiki			HotpotQA			Musique			Bamboogle		
	EM	Acc	F1	EM	Acc	F1	EM	Acc	F1	EM	Acc	F1
Direct Generation												
Qwen2.5-7B-Instruct	26.9	28.5	32.2	17.2	20.2	26.7	5.9	9.1	13.6	8.8	14.4	16.9
Sequential RAG System												
Naive RAG (TopK = 5)	25.8	29.5	32.7	33.8	39.1	44.9	11.5	15.9	20.8	18.4	21.6	28.3
Naive RAG (TopK = 10)	28.2	31.7	34.8	35.3	39.2	46.1	12.8	18.1	20.0	13.6	15.2	22.1
Iterative RAG System												
Self-RAG	11.3	32.0	23.5	18.3	34.0	30.5	6.3	12.3	15.1	3.2	16.8	13.1
IR-CoT	24.3	37.3	34.4	33.6	42.6	45.6	12.4	19.7	21.0	21.6	26.4	34.2
Iter-Retgen	29.8	34.2	36.2	37.0	41.7	49.1	15.3	19.0	22.9	23.2	24.0	31.8
Query Understanding												
RQ-RAG	26.8	35.8	36.8	28.9	33.5	36.3	19.8	24.8	27.8	24.8	26.4	34.7
QCompiler (ours)	**44.5**	**53.5**	**51.9**	**38.5**	**46.1**	**50.2**	**25.0**	**35.4**	**34.8**	**37.1**	**41.9**	**49.2**

Table 1: Comparison of different methods. The best results are in bold. The base generator model is Qwen-2.5-7B-Instruct, and **QCompiler** is a fine-tuned Llama3.2-3B-Instruct model.

the outputs at various temperature settings and then select a valid AST for subsequent processing.

3.6 Why Can QCompiler Improve Existing RAG Systems?

QCompiler can improve RAG systems in multiple ways: (1) Unlike existing end-to-end approaches, QCompiler is a lightweight framework that focuses on generating **structured intermediate representations** for complex queries by compiling them into ASTs to capture their implicit intentions, nested structures, and intricate dependencies. This process inherently handles the rewriting, disambiguation, decomposition, and expansion of complex queries. (2) The atomicity of the sub-queries in the leaf nodes ensures precise document retrieval and answer generation, significantly improving the RAG system's ability to address complex queries. (3) In practical deployment scenarios, developers can even design extensive post-processing logic to refine the AST compiled by QCompiler. These features make QCompiler highly adaptable to integration with existing RAG systems.

4 Experiment

In this section, we validate the effectiveness of our method through a series of experiments.

4.1 Datasets and Metrics

We select four multi-hop benchmarks to validate the effectiveness of our approach in understanding complex queries. These datasets include **2WikiMultihopQA** (Ho et al., 2020), **HotpotQA** (Yang et al., 2018), **Musique** (Trivedi et al., 2022b), and **Bamboogle** (Press et al., 2022). We use 2WikiMultihopQA, HotpotQA, and Musique's training set to construct QCompiler's fine-tuning datasets. We provide statistical descriptions of these benchmarks in Appendix E and the training details of QCompiler in Appendix F.

For evaluation, we use Extract Match (EM), Accuracy (Acc), and the F1 score to evaluate the results of the system responses.

4.2 Baselines

We mainly select the following four categories of baseline models:

Direct Generation of LLM. To better reflect the effectiveness of the retrieval augmentation process and the reasoning capabilities of complex RAG systems, we first compared the results of allowing the response model to directly generate answers.

Sequential RAG System. (1) **Naive RAG**: Directly uses original query for retrieval and generation. Here we evaluate the generation with Top-5 and Top-10 retrieved documents separately for comparison.

Iterative RAG System. (1) **Self-RAG** (Asai et al., 2023): Trains a language model to dynamically determine when to retrieve external information and critiques its outputs using specialized reflection tokens. (2) **IR-CoT** (Trivedi et al., 2022a): Alternates between retrieval steps and Chain-of-Thought reasoning, allowing each step to inform and refine the other. (3) **Iter-RetGen** (Shao et al., 2023): Synergizes retrieval and generation itera-

Methods	2Wiki			HotpotQA			Musique			Bamboogle		
	EM	Acc	F1	EM	Acc	F1	EM	Acc	F1	EM	Acc	F1
Llama3.x series												
Llama3.2-1B-Instruct	**45.1**	**54.1**	**52.5**	36.9	45.1	49.0	**25.1**	35.3	**35.2**	34.7	38.9	46.7
Llama3.2-3B-Instruct	44.5	53.5	51.9	**38.5**	**46.1**	**50.2**	25.0	35.4	34.8	**37.1**	41.9	**49.2**
Llama3.1-8B-Instruct	44.6	53.7	52.0	38.0	45.6	49.9	25.1	**36.0**	35.1	36.6	42.1	49.2
Qwen2.5 series												
Qwen2.5-0.5B-Instruct	44.1	53.0	51.5	37.2	44.4	48.8	25.0	34.9	34.8	35.7	41.1	48.6
Qwen2.5-1.5B-Instruct	44.3	53.3	51.8	36.9	44.4	49.0	24.8	35.5	35.0	36.8	41.4	48.7
Qwen2.5-3B-Instruct	44.3	53.5	51.8	37.2	44.2	48.8	24.9	35.6	35.1	35.0	42.2	48.9
Qwen2.5-7B-Instruct	43.9	53.5	51.8	36.9	44.5	49.2	24.8	35.4	35.2	35.7	**42.9**	49.1

Table 2: The performance of **QCompiler** fine-tuned on different base models.

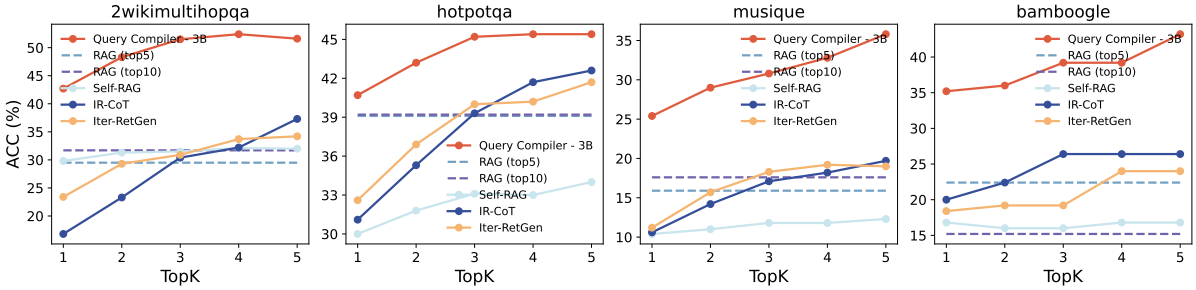

Figure 3: Illustration of the atomicity of sub-queries in leaf nodes, comparing with traditional RAG systems and Iterative RAG systems, **QCompiler** has few document retrievals and more accurate responses.

tively, where model output guides subsequent retrievals, and retrieved information enhances future generations.

Query Understanding. (1) **RQ-RAG** (Chan et al., 2024): Train a model to learn to rewrite, disambiguate, and decompose complex queries to retrieve documents and to generate the final response in an end-to-end manner.

We provide details on experiment implementations in the Appendix G.

4.3 Main Results

Table 1 presents the main results of various methods, including ours. ASTs compiled by QCompiler greatly improve the response model's capability without additional training for the model or retriever, achieving best performance on four benchmarks. Improvement is especially notable in challenging benchmarks like 2WikiMultihopQA and Musique.

4.4 Analysis of Results

The observed performance improvements can be attributed to the following factors:

(1) Retrieval&Generation Limitations of Original Queries. Retrieval based on the original query often misses key information within the top-k documents due to insufficient context. Even larger k may fail to retrieve all relevant documents while introducing significant noise into the generation process, inherently limiting the performance of traditional RAG systems.

(2) Limited Iterative Query Planning. For iterative RAG systems with constrained base model size or performance, the ability to plan new queries is weak, often requiring more iterations to complete a response. The sub-queries generated during these iterations can be viewed as derivations from a larger grammar G' (rather than the minimal grammar $G[q]$) formalizing the original query, introducing redundancy for both the retrieved documents and the iterations, and further limiting the performance of iterative RAG systems.

(3) Improved Query Understanding with QCompiler. QCompiler is fine-tuned on a large dataset of grammar-based compiled expressions. It demonstrates a superior understanding of complex queries. It resembles generating a complete plan for the complex query in one step and performs rea-

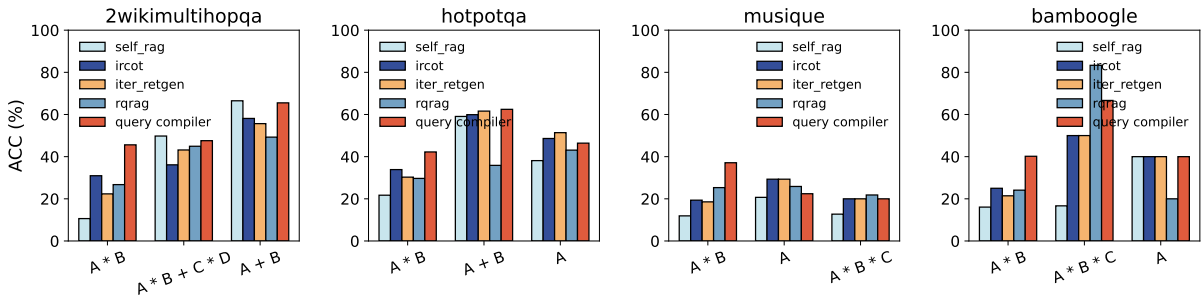

Figure 4: The number of correctly answered queries for each type compiled by **QCompiler** across different methods.

soning based on predefined symbolic rules, which can provide an accurate answer to the query.

5 Analysis

In this section, we focus on analyzing how **QCompiler** can improve the performance of the RAG system.

5.1 Scaling law in QCompiler

We train QCompiler using base models of varying sizes with the same training data and experimental settings, and evaluate their performance in Table 2. The results reveal that performance across query compilers of different sizes is nearly identical. This suggests that the grammar-based generation task is relatively straightforward to learn, meaning that larger base models (e.g., Llama3.1-8B-Instruct and Qwen-2.5-7B-Instruct) do not necessarily outperform smaller ones (e.g., Llama3.2-3B-Instruct). This points to a limitation in current benchmarks for multi-hop queries, which may lack sufficient complexity and diversity, enabling smaller, distilled models to perform comparably well on existing benchmarks.

5.2 Analysis of Leaf Node's Atomicity

An essential premise of QCompiler is the atomicity of sub-queries in leaf nodes. As illustrated in Figure 1 and Section 3.1, a query represented by a leaf node in the AST should be an indivisible single-turn query that cannot be further decomposed into smaller components. For knowledge-intensive tasks, such single-turn queries must be precise enough to answer questions by retrieving as few documents as possible from the corpus. We compared the performance of different methods when retrieving different numbers of Top-K documents per query node, as shown in Figure 3. The results show that for QCompiler, retrieving only a small number of documents per query node (**even just one document**) is sufficient to achieve strong performance across different benchmarks.

5.3 Analysis of Query Types

We use QCompiler to compile queries into their respective expression types. In Figure 4, we present data on the three most common query types, documenting the percentage of correct responses achieved for each type. The findings indicate: (1) QCompiler provides moderate improvement for single-hop questions since it applies a single cycle of refinement and processing in a recursive-descent manner, unlike iterative RAG systems that repeatedly refine these queries. (2) For list queries structured as $A + B$, QCompiler also provides a moderate improvement, suggesting that these queries are not difficult and can be managed by iterative RAG systems. (3) QCompiler excels with dependent queries, especially those formed as $A \times B$. This highlights the limitations of current iterative RAG systems: in multi-hop questions, the critical challenge lies in pinpointing the initial query and its answer correctly, a key factor that limits the system's effectiveness.

6 Conclusion

In this paper, we present **QCompiler**, a neuro-symbolic framework inspired by linguistic grammar rules and compiler design. We first give a minimal yet sufficient grammar $G[q]$ to formalize and generate complex queries. Then QCompiler can efficiently compile each complex query into an Abstract Syntax Tree that contains these types and captures its nested structures and complex dependencies, demonstrating a strong ability to understand and analyze complex queries. The atomicity of the sub-queries in the leaf nodes ensures precise document retrieval and response generation. This lightweight framework enables seamless integration with existing RAG systems, highlighting

its broad applicability and flexibility in practical deployment scenarios.

7 Limitation

Due to the limitations of existing multi-hop datasets, we lack more complex scenarios to train and validate the performance of the grammar-based QCompiler. For example, a key issue is the absence of benchmarks that feature complex queries that use parentheses to control the execution order, which may limit the generalization of the trained model. Additionally, in this paper, we focus solely on supervised fine-tuning to train QCompiler. Future improvement strategies include, but are not limited to, constructing more diverse and complex benchmarks for training and evaluation, and employing reinforcement learning with step-level reward models to generate more optimal expressions.

8 Acknowledgment

This work was supported by Beijing Municipal Science and Technology Project No. Z231100010323009, National Science and Technology Major Project No. 2022ZD0120103, National Natural Science Foundation of China No. 62272467, Beijing Natural Science Foundation No. L233008, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China No. 22XNKJ34. The work was partially done at the Engineering Research Center of Next-Generation Intelligent Search and Recommendation, MOE.

References

Kareem Ahmed, Stefano Teso, Kai-Wei Chang, Guy Van den Broeck, and Antonio Vergari. 2022. Semantic probabilistic layers for neuro-symbolic learning. *Advances in Neural Information Processing Systems*, 35:29944–29959.

Alfred V Aho and Jeffrey D Ullman. 1969. Translations on a context free grammar. In *Proceedings of the first annual ACM symposium on Theory of computing*, pages 93–112.

V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools*. pearson Education.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511*.

Chi-Min Chan, Chunpu Xu, Ruibin Yuan, Hongyin Luo, Wei Xue, Yike Guo, and Jie Fu. 2024. Rq-rag: Learning to refine queries for retrieval augmented generation. *arXiv preprint arXiv:2404.00610*.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Jiangning Liu, Wenwei Zhang, Kai Chen, and Feng Zhao. 2024. Mindsearch: Mimicking human minds elicits deep ai searcher. *arXiv preprint arXiv:2407.20183*.

Marius-Constantin Dinu, Claudiu Leoveanu-Condrei, Markus Holzleitner, Werner Zellinger, and Sepp Hochreiter. 2024. Symbolicai: A framework for logic-based approaches combining generative models and solvers. *arXiv preprint arXiv:2402.00854*.

Kaisi Guan, Qian Cao, Yuchong Sun, Xiting Wang, and Ruihua Song. 2024. Bsharedrag: Backbone shared retrieval-augmented generation for the e-commerce domain. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, pages 1137–1158. Association for Computational Linguistics.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962.

Uri Hasson, Samuel A Nastase, and Ariel Goldstein. 2020. Direct fit to nature: an evolutionary perspective on biological and artificial neural networks. *Neuron*, 105(3):416–434.

Pascal Hitzler, Aaron Eberhart, Monireh Ebrahimi, Md Kamruzzaman Sarker, and Lu Zhou. 2022. Neuro-symbolic approaches in artificial intelligence. *National Science Review*, 9(6):nwac035.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. Constructing a multi-hop qa dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6609–6625.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023a. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839*.

Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023b. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*.

Jiajie Jin, Yutao Zhu, Xinyu Yang, Chenghao Zhang, and Zhicheng Dou. 2024. Flashrag: A modular toolkit for efficient retrieval-augmented generation research. *arXiv preprint arXiv:2405.13576*.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

Tushar Khot, Daniel Khashabi, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2020. Text modular networks: Learning to decompose tasks in the language of existing models. *arXiv preprint arXiv:2009.00751*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Yangning Li, Yinghui Li, Xingyu Wang, Yong Jiang, Zhen Zhang, Xinran Zheng, Hui Wang, Hai-Tao Zheng, Philip S Yu, Fei Huang, et al. 2024. Benchmarking multimodal retrieval augmented generation with dynamic vqa dataset and self-adaptive planning agent. *arXiv preprint arXiv:2411.02937*.

Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. 2023. Compressing context to enhance inference efficiency of large language models. *arXiv preprint arXiv:2310.06201*.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283*.

Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2019. Multi-hop reading comprehension through question decomposition and rescoring. *arXiv preprint arXiv:1906.02916*.

Theo X Olausson, Alex Gu, Benjamin Lipkin, Cedegao E Zhang, Armando Solar-Lezama, Joshua B Tenenbaum, and Roger Levy. 2023. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. *arXiv preprint arXiv:2310.15164*.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. 2022. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. *arXiv preprint arXiv:2305.15294*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Jiejun Tan, Zhicheng Dou, Yutao Zhu, Peidong Guo, Kun Fang, and Ji-Rong Wen. 2024. Small models, big insights: Leveraging slim proxy models to decide when and what to retrieve for llms. *arXiv preprint arXiv:2402.12052*.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022a. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509*.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022b. Musique: Multi-hop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554.

Prakhar Verma, Sukruta Prakash Midigeshi, Gaurav Sinha, Arno Solin, Nagarajan Natarajan, and Amit Sharma. 2024. Planxrag: Planning-guided retrieval augmented generation. *arXiv preprint arXiv:2410.20753*.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2024. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36.

Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. 2020. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang,

Wentao Zhang, and Bin Cui. 2024. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473*.

Appendix

A Proof of BNF Grammar's Completeness for Complex Queries

Our goal is to prove that any complex query can be generated using the production rules of the given grammar.

A.1 Lemmas

LEMMAS

Lemma1: All <***Atomic Query***> are <***Dependent Query***>.
Lemma2: All <***Dependent Query***> are <***List Query***>.

According to the productions rules, these can be easily derived.

A.2 Inductive Proof

COMPLETENESS OF GRAMMAR

Theorem: Any valid query string constructed using query terms and operators '+' and '∗' can be generated by the given grammar.

Proof

Base Case: Any atomic query $q \in Q_{atomic}$ is an <***Atomic Query***> and also a <***Dependent Query***>. Therefore, it is a <***List Query***> and consequently a <***Complex Query***>.

Inductive Hypothesis Assume that the prodution rule can generate all valid query strings of length less than n. Here, the definition of length n refers to a query containing n atomic query strings.

Inductive Step For a query string of length n, there are the following cases:

Case 1: The query is a dependent query in the form: '<***Dependent Query***>×<***Atomic Query***>'. By the inductive hypothesis, <***Dependent Query***> and <***Atomic Query***> can be generated by the production rule (because length $< n$). Therefore, this query can be generated by the production rule.

Case 2: The query string is a list query, in the form: '<***List Query***>+<***Dependent Query***>'. By the inductive hypothesis, both <***List Query***> and <***Dependent Query***> can be generated by the production rule. Therefore, this query can be generated by production rule.

Case 3: The query string is a parenthesized list query in the form: '(<***List query***>)'. By **Case 1**, <***List Query***> can be generated by the syntax. So the query can be generated by syntax.

Conclusion By mathematical induction, query strings of length less than n, along with the inductive step for strings of length n, can be generated by the grammar. Therefore, the grammar is complete.

B Proof of Minimality of the Grammar

To prove that the given grammar is minimal, we must show that: (1) The production rule generates the target query language. (2) No symbols or rules are redundant in the production rules. (3) Simpler production rules cannot generate the same language.

B.1 Generation Completeness

The completeness of the grammar has previously been proven, confirming that the grammar can generate all valid queries.

B.2 Elimination of Redundant Symbols

The non-terminal symbols in the grammars are:

- <***Atomic Query***>: Defines the basic single-turn query, indispensable.
- <***Dependent Query***>: Introduces the '×' operator for sub-queries with dependencies, which cannot be replaced by other terminal and non-terminal symbols.
- <***List Query***>: Introduces the '+' operator for sub-queries without dependencies.
- <***Complex Query***>: Serves as the starting symbol for the entire production rule and cannot be omitted.

Therefore, all symbols serve a distinct purpose and cannot be removed without reducing the generative power of the syntax.

B.3 Elimination of Redundant Rules

The production rules are as follows:

PRODUCTION RULES OF QUERY

<***Atomic***>::= $q \in Q_{atomic}$ | (<***List***>)
<***Dependent***>::=<***Atomic***>|<***Dependent***>×<***Atomic***>
<***List***>::=<***Dependent***>|<***List***>+<***Dependent***>
<***Complex***>::= <***List***>

Each rule is necessary to express the target query language:

1. Removing <***Dependent Query***> makes it impossible to generate queries connected by '×'.

2. Removing <***List Query***> makes it impossible to generate queries connected by '+'.

3. Removing <***Complex Query***> removes the start symbol of the production rule.

Therefore, no rule is redundant or replaceable.

Nonequivalence of simpler syntax

We attempt to construct a simpler syntax:

1. If <***Dependent Query***> is replaced directly with <***Atomic Query***>, queries with dependencies involving '×' cannot be expressed.

2. If <***List Query***> is omitted, queries without dependencies involving '+' cannot be expressed. These simplifications fail to generate the target query language, confirming that there is no simpler equivalent grammar.

B.4 Conclusion

The grammar is minimal, as it generates the target query language, contains no redundant symbols or rules, and cannot be simplified without reducing its ability of producing complex queries.

C Comparison between QCompiler and QDMR

Based on the results and evaluation of existing benchmarks, our method closely resembles QDMR (Wolfson et al., 2020) in terms of performance representation, because both methods use a seq2seq model to learn decomposition rules to break down multiple search intents of complex questions. However, through deeper syntactical analysis, we observed that many of the operators defined in QDMR, such as **PROJECT**, **FILTER**, **AGGREGATE**, **BOOLEAN**, and **COMPARATIVE**, while fully capable of capturing all search intents of the original query, are actually forming a redundant grammar $G' \subseteq G$. From a linguistic standpoint, these operators form a complete but not a minimal grammar, which may be poor for optimization. Our minimal BNF grammar $G[q] \subset G'$, it only defines dependent and parallel relationships via '×' and '+', making it clear and easy to represent original complex queries.

Moreover, QDMR specifically trains a model to generate the final answer, which contrasts with the design philosophy of QCompiler. Our aim is to provide a plug-and-play query parsing module within the RAG system, rather than adapting the existing answer format from the benchmark. (6) This approach fosters greater flexibility and allows the components to be reused by the open source community and in practical development.

D Algorithm of Query AST Validation

Algorithm 1 Query AST Validation Algorithm

Require: A Root Node node
Ensure: Returns **True** if the query is valid, **False** otherwise

```
if node.type = '<Atomic Query>' then
    if flag = 0 and node.placeholder exists then
        return False
    end if
    if flag = 1 and not node.placeholder exists then
        return False
    end if
    return True
end if
if node.type = '<Dependent Query>' then
    Let left ← node.children[0]
    Let right ← node.children[1]
    return valid_query(left, flag = 0) and valid_query(right, flag = 1)
end if
if node.type = '<List Query>' then
    for child ∈ node.children do
        if not valid_query(child, flag = flag) then
            return False
        end if
    end for
    return True
end if
```

E Statistical Description of Four Benchmarks

We provide a statistical description of the four evaluation benchmarks in Table 3.

Name	Source	Train	Valid	Test
HotpotQA	wiki	90,447	7,405	/
2WikiMultiHopQA	wiki	15,000	12,576	/
Musique	wiki	19,938	2,417	/
Bamboogle	wiki	/	/	125

Table 3: Multi-hop QA Datasets

For each of HotpotQA, 2WikiMultiHopQA, and Musique, we select the first 1,000 samples from

the valid set for evaluation, while for Bamboogle, we use the entire test set for evaluation.

F Training Details of Query Compiler

We build an instruction-tuning dataset from HotpotQA, 2WikiMultiHopQA, and Musique to train the *Query Expression Translator*. Using Chain-of-Thought prompting and few-shot examples (which is presented in Table 5), we prompt the Qwen2.5-72B-Instruct model to generate valid expressions for each query in the training set.

Next, we use the algorithm described in Appendix D to validate the syntax trees parsed from the expressions, filtering out invalid expressions and get valid query-expression pairs for training.

We fine-tune the base model for 1 epoch using LoRA (Hu et al., 2021), setting the $r = 16$, $\alpha = 32$, batch_size $= 64$, and the learning rate to $5e - 5$.

G Experiment Implementations

Corpus and Retriever We use Wikipedia dump 2018 (Karpukhin et al., 2020) as retrieved corpus and bge-base-en-v1.5 (Xiao et al., 2023) as dense retriever. For each retrieval process, we concatenate instruction '*Represent this sentence to search relevant passages:*' and query to retrieve relevant top-k documents.

Generator We use Qwen2.5-7B-Instruct as the base generator to give a response to every query.

Metric The three evaluation metrics we use are as follows:

(1) Exact Match (EM): Measures whether the generated answer exactly matches the golden answer, ensuring strict consistency.

(2) Accuracy (Acc): Evaluates whether the golden answer is included within the generated answer.

(3) F1 Score: Computes the F1 score between the tokenized generated answer and the tokenized golden answer.

Baseline Settings

Self-RAG (Asai et al., 2023) We set max_depth = 2 and beam_width = 2, threshold = 0.2 for testing. Due to the output format of this method, the EM metric may be underestimated.

IR-CoT & Iter-RetGen (Trivedi et al., 2022a; Shao et al., 2023) We set max_iterations = 5 for testing. We found that increasing the number of iterations for these methods does not lead to better performance, so we set a maximum number of iterations accordingly.

RQ-RAG (Chan et al., 2024) We set max_depth = 4 for 2WikiMultihopQA and Musique, and 3 for HotpotQA and Bamboogle for testing. Since the number of sampled paths grows exponentially with depth and the token count for context increases significantly, we did not experiment with larger maximum depth settings.

H Efficiency Analysis

In this section, we analysis efficiency of QCompiler from two perspectives: token consumption and throughput.

Token Consumption The efficiency of RAG systems depends on various factors such as CPU, GPU, NPU, and storage of the machine. To highlight the efficiency of QCompiler, we focus on token consumption. Each method's token usage can be divided into three components: prompt tokens, retrieved document tokens, and response generation tokens.

QCompiler improves efficiency by precompiling an Abstract Syntax Tree for each query, making the process deterministic and eliminating redundant iterations. This avoids unnecessary prompt and response generation tokens that are common in LLM-based RAG systems. Moreover, The atomicity of the sub-queries in the leaf nodes ensures precise document retrieval and answer generation while maintaining competitive performance. This feature actually improves the efficiency.

We compare the average token consumption per query across different methods when benchmark performance is similar:

Methods	2wikimultihopqa	hotpotqa	musique	bamboogle
Self-RAG	2079.1	2034.2	1980.0	1945.8
Iter-Retgen	3628.0	2835.8	3430.1	3369.4
IR-CoT	2462.0	1917.9	2347.3	2320.4
QCompiler	**1176.3**	**1042.0**	**1061.7**	**1040.6**

Table 4: Token consumption of different methods.

Throughput The baseline methods use LLMs to generate new query content in each iteration, which limits their ability to execute independent sub-queries in parallel. For example, the query "Who is older, James Cameron or Steven Allan Spielberg?" includes two independent sub-queries, but existing methods must process them sequentially. In contrast, our compiled AST allows for parallel execution of child nodes in list queries, significantly improving throughput. This feature also

improves efficiency.

I Open-domain Question with QCompiler

Open-domain question answering is a current challenge in research, and the quality of the answers largely depends on the performance of the base model. For example, the implicit comparative question (e.g. Compare the market share and revenue growth of top 5 EV manufacturers in North America and Europe over the last 3 years.) may need agentic RAG systems to autonomously expand search intents during running time, and the current baselines have not been able to fully address these issues. To solve this problem, QCompiler can be integrated into agentic RAG systems to generate a AST for the remaining question in each iteration, but this is beyond the scope of this work.

In this context, our method still attempts to generate several sub-intents of the original query, retrieving relevant documents and integrating information to provide a reference for the final answer, which is shown in Table 8 and Table 9.

J Other Disccusions

We found that, under the grammar instructions $G[q]$ and the few-shot examples format, the 7B model already demonstrated relatively strong parsing capabilities. Although our experiments were conducted by building training data from the training sets provided by three benchmarks to fine-tune smaller models, this step was actually aimed at obtaining a specialized model with fewer parameters for the challenge of low-resource settings.

Similarly, the focus of this work is to design a minimal, and sufficient structured intermediate query representation for RAG systems. Therefore, we did not train the base model for generating the final answers, and hence avoided overfitting to the existing benchmarks.

Table 5: Prompt to generate training data

Chain-of-Thought Prompting

You are an expert in query intent understanding, tasked with decomposing complex queries into basic components. Follow the step-by-step procedure below to generate a BNF-compliant expression for each query.

==========

Query Types and Grammar Definitions

==========

Query Types:

1. AtomicQuery:
   - Simple, direct queries that require a factual answer.
   - Non-decomposable, orthogonal, and non-redu 2. DependentQuery:
   - Multi-step queries where each step depends on the result of the previous one.
   - Composed of multiple AtomicQueries with dependencies.
3. ListQuery:
   - Requires decomposition into multiple parallel, independent sub-queries.

BNF Definitions:

- Set of atomic query terms (W):
  All possible atomic query strings, where each atomic query is independent and non-redundant.
- Set of operators (O): '+' (parallel), '*' (dependent)
- <AtomicQuery> ::= w ∈ W | '(' <ListQuery> ')'
  - expressions enclosed in parentheses can also be considered as a production rule for <AtomicQuery>.
- <DependentQuery> ::= <AtomicQuery> | <DependentQuery> '*' <AtomicQuery>
  - <AtomicQuery> may include placeholders formatted as placeholder name.
  - '*' indicates that the next query depends on the result of the previous query.
- <ListQuery> ::= <DependentQuery> | <ListQuery> '+' <DependentQuery>
  - '+' denotes parallel relationships among queries.

==========

Example Workflows

==========

— Example —

query = How many Germans live in the colonial holding in Aruba's continent that was governed by Prazeres's country?

Step1: **Define atomic queries:**

- query1.1: Which continent is Aruba in?
- query1.2: Which country is Prazeres in?
- query2: Which colonial holding in {continent} was governed by {country}?
- query3: How many Germans live in {colonial_holding}?

Step2: **Queries Combination:**

Thought:

- query1.1 and query1.2 are parallel → Use '+'
- query2 depends on the results of both query1.1 and query1.2 → Use '*'
- query3 depends on query2 → Use '*'

- Combine:
  (query1.1 + query1.2) * query2 * query3

compiled_expression = (Which continent is Aruba in? + Which country is Prazeres in?) * Which colonial holding in {continent} was governed by {country}? * How many Germans live in {colonial_holding}?

==========

Task Instructions

==========

1. Decompose the user's query into an ordered set of AtomicQueries, ensuring each query is factual, independent, and non-redundant.
2. Determine whether sub-queries are parallel (use '+') or dependent (use '*').
3. If you use '*', the next query must have a placeholder referencing the previous step's result, e.g., {placeholder}.
4. Output your reasoning in two steps:
   - Step1: **Define atomic queries**
   - Step2: **Queries Combination**
   Then provide the final expression.
5. Maintain the same language as the input query when formulating AtomicQueries.
6. Follow the example format to ensure consistency.

Please decompose and compile each complex query into a BNF-compliant expression using '+', '*', '()', and '{placeholders}', then output in the specified format.

Table 6: An example of QComplier's Query Expression Translator and parsed AST

System Prompt

You are an expert in query intent understanding, tasked with decomposing complex queries into basic components. Follow the step-by-step procedure below to generate a BNF-compliant expression for each query.

==========
Query Types and Grammar Definitions
==========
Query Types:

1. AtomicQuery:
   - Simple, direct queries that require a factual answer.
   - Non-decomposable, orthogonal, and non-redundant.
2. DependentQuery:
   - Multi-step queries where each step depends on the result of the previous one.
   - Composed of multiple AtomicQueries with dependencies.
3. ListQuery:
   - Requires decomposition into multiple parallel, independent sub-queries.

BNF Definitions:

- Set of atomic query terms (W):
  All possible atomic query strings, where each atomic query is independent and non-redundant.
- Set of operators (O): '+' (parallel), '*' (dependent)
- <AtomicQuery> ::= w ∈ W | '(' <ListQuery> ')'
  - expressions enclosed in parentheses can also be considered as a production rule for <AtomicQuery>.
- <DependentQuery> ::= <AtomicQuery> | <DependentQuery> '*' <AtomicQuery>
  - <AtomicQuery> may include placeholders formatted as placeholder name.
  - '*' indicates that the next query depends on the result of the previous query.
- <ListQuery> ::= <DependentQuery> | <ListQuery> '+' <DependentQuery>
  - '+' denotes parallel relationships among queries.

==========
Task Instructions
==========

1. Decompose the user's query into an ordered set of AtomicQueries, ensuring each query is factual, independent, and non-redundant.
2. Determine whether sub-queries are parallel (use '+') or dependent (use '*').
3. If you use '*', the next query must have a placeholder referencing the previous step's result, e.g., {placeholder}.
4. Maintain the same language as the input query when formulating AtomicQueries.

Please decompose and compile each complex query into a BNF-compliant expression using '+', '*', '()', and '{placeholders}', then output in the specified format.

Input (Original Query)

I want to find an introduction and reviews of JK. Rowling's most popular book and checking if the local library has it?

Output (BNF Expression)

What is JK. Rowling's most popular book? * (Find an introduction to {book} + Find reviews of {book} + Does the local library have {book}?)

Parsed AST

ComplexQuery(value='What is JK. Rowling's most popular book? * (Find an introduction to {book} + Find reviews of {book} + Does the local library have {book}?)', [
  DependentQuery(value='What is JK. Rowling's most popular book? * (Find an introduction to {book} + Find reviews of {book} + Does the local library have {book}?)', [
    AtomicQuery(value='What is JK. Rowling's most popular book?'),
    ListQuery(value='(Find an introduction to {book} + Find reviews of {book} + Does the local library have {book}?)', placeholder=['book'], [
      AtomicQuery(value='Find an introduction to {book}', placeholder=['book']),
      AtomicQuery(value='Find reviews of {book}', placeholder=['book']),
      AtomicQuery(value='Does the local library have {book}?', placeholder=['book'])
    ])
  ])
])

Table 7: An example for **multi-hop** question with QCompiler

Example #1 Multi-hop question from Musique

Question:
Why did Roncalli leave the city where the creator of La Schiavona died?
Ground truth:
for the conclave in Rome; Rome; Roma

BNF Expression

Who is the creator of La Schiavona? * Where did {creator} die? * Why did Roncalli leave {city}?

Parsed AST

ComplexQuery(value='Who is the creator of La Schiavona? * Where did {creator} die? * Why did Roncalli leave {city}?', [
DependentQuery(value='Who is the creator of La Schiavona? * Where did {creator} die? * Why did Roncalli leave {city}?', [
DependentQuery(value='Who is the creator of La Schiavona? * Where did {creator} die?', [
AtomicQuery(value='Who is the creator of La Schiavona?'),
AtomicQuery(value='Where did {creator} die?', placeholder=['creator'])
]),
AtomicQuery(value='Why did Roncalli leave {city}?', placeholder=['city'])
])
])

Depth-Fisrt-Search Process in AST

AtomicQuery 1:
Who is the creator of La Schiavona?
Retrieved Documents:
Doc1, Title = Andrea Schiavone Andrea Meldolla, Content = ...
Doc2, Title = ..., Content = ...
...
Result:
The creator of La Schiavona is Titian. This information can be directly inferred from the document titled L̈a Schiavona,ẅhich states, P̈ortrait of a Lady, also known as La Schiavona (the woman from Dalmatia), is a 1510–12 portrait by Titian of an unknown woman.

AtomicQuery 2:
Where did **Titian** die? (get placeholder 'creator"s value from its left node.)
Retrieved Documents:
...
Result:
According to the documents, Titian died in Venice on 27 August 1576.

AtomicQuery 3:
Why did Roncalli leave **Venice**? (get placeholder 'city"s value from its left node.)
Retrieved Documents:
...
Result:
Based on the information provided in the documents, Roncalli left Venice after the death of Pope Pius XII on 9 October 1958 to participate in the papal conclave in Rome. Doc2 mentions, F̈ollowing the death of Pope Pius XII on 9 October 1958, Roncalli watched the live funeral on his last full day in Venice on 11 October. His journal was specifically concerned with the funeral and the abused state of the late pontiff's corpse. Roncalli left Venice for the conclave in Rome well aware that he was 'papabile', and after eleven ballots, was elected to.

Concatenated three sub-queries and answers to generate the final answer

Answer the question based the original question and its sub-questions and answers.
{orignal question}
{sub-qurstions and answers}
Answer: Because Pope Pius XII died on 9 October 1958, Roncalli left Venice to participate in the papal **conclave in Rome**.

Table 8: An example for **summarization** question with QCompiler

Example #2 Summarization Question

Question:
Given the current geopolitical tensions and supply chain disruptions, how might the semiconductor industry evolve over the next five years?

BNF Expression

What are the current geopolitical tensions affecting the semiconductor industry + What are the current supply chain disruptions affecting the semiconductor industry + How might the semiconductor industry adapt to the geopolitical tensions over the next five years + How might the semiconductor industry adapt to the supply chain disruptions over the next five years

Parsed AST

ComplexQuery(value='What are the current geopolitical tensions affecting the semiconductor industry + What are the current supply chain disruptions affecting the semiconductor industry + How might the semiconductor industry adapt to the geopolitical tensions over the next five years + How might the semiconductor industry adapt to the supply chain disruptions over the next five years', [
AtomicQuery(value='What are the current geopolitical tensions affecting the semiconductor industry'),
AtomicQuery(value='What are the current supply chain disruptions affecting the semiconductor industry'),
AtomicQuery(value='How might the semiconductor industry adapt to the geopolitical tensions over the next five years'),
AtomicQuery(value='How might the semiconductor industry adapt to the supply chain disruptions over the next five years')
])

Concatenated three sub-queries and answers to generate the final answer

The semiconductor industry might evolve by diversifying supply chains, forming public-private partnerships, engaging in collaborative efforts, increasing investment in R&D, focusing on advanced technologies, and adapting to market volatility. It will also enhance supply network resilience through dynamic business model reinvention, collaboration across the network, maintaining a flexible supply chain, and developing strategic resilience.

Table 9: An example for **implicit comparative** question with QCompiler

Example #3 Implicit Comparative Question

Question:
Compare the market share and revenue growth of top 5 EV manufacturers in North America and Europe over the last 3 years.

BNF Expression

Who are the top 5 EV manufacturers in North America * What is the market share and revenue growth of {manufacturer} in North America over the last 3 years + Who are the top 5 EV manufacturers in Europe * What is the market share and revenue growth of {manufacturer} in Europe over the last 3 years

Parsed AST

ComplexQuery(value='Who are the top 5 EV manufacturers in North America * What is the market share and revenue growth of {manufacturer} in North America over the last 3 years + Who are the top 5 EV manufacturers in Europe * What is the market share and revenue growth of {manufacturer} in Europe over the last 3 years', [
DependentQuery(value='Who are the top 5 EV manufacturers in North America * What is the market share and revenue growth of {manufacturer} in North America over the last 3 years', [
AtomicQuery(value='Who are the top 5 EV manufacturers in North America'),
AtomicQuery(value='What is the market share and revenue growth of {manufacturer} in North America over the last 3 years', placeholder=['manufacturer'])
]),
DependentQuery(value='Who are the top 5 EV manufacturers in Europe * What is the market share and revenue growth of {manufacturer} in Europe over the last 3 years', [
AtomicQuery(value='Who are the top 5 EV manufacturers in Europe'),
AtomicQuery(value='What is the market share and revenue growth of {manufacturer} in Europe over the last 3 years', placeholder=['manufacturer'])
])
])