We thank the reviewers for their constructive feedback! Below is our response, and we have revised the paper to address all comments. We have organized our reply according to the order of the Meta-Review, and labeling each response with [A(M)]. For your convenience, we highlight changes in blue in the text. Additionally, we provide the full version with the appendix for your reference: https://wangmengying.me/papers/uniqgen.pdf.

Signposts for reviewers:
- **R1**: W1(M1), W2(M2), W3(M3), O2(M4)
- **R2**: W1(M5), W2(M2), W3(M6)
- **R3**: D1(M6), D2(M5), D3(M7), D4(M8), D5(M9), W4(M10)

---

**[M1]** *Clarify Novelty and Guarantees (R1:W1).*

• **[R1:W1]** *Novelty over Chase and Backchase is unclear. The paper borrows the classic idea but does not state clear guarantees under LLM-guided pruning. Authors may need to define the exact assumptions and what is proved versus what is heuristic.*

**[A1]** We have revised our paper to clarify the following.

(1) UniQGen does not simply apply Chase and Backchase directly for query rewriting, but serves a nontrivial integration solution that extends it into a budgeted, runtime-validated, cross-language query generation pipeline by introducing an online-constructed, uncertainty-scored constraint IR (CTable) and a pluggable renderer, rather than relying on offline dependencies or training; it then offers relative guarantees *w.r.t.* a fixed referenced answer set $A$, provided by user or LLM oracle. The reported guarantee comes from the runtime validations for the generated query, concretely, $Q_U$ is accepted iff $G(A) \subseteq Q_U(G)$, which we call LLM-completeness, $Q_M$ is accepted iff it preserves coverage and satisfies $Q_M(G) \subseteq G(A)$, which we call LLM-soundness. We have highlighted these novelties in Sec.I (before Example 1).

(2) We have clarified the performance guarantees in Section III, and have included the formal proof in the Appendix. We show that our framework has provable guarantees as follows: (i) QChase ensures relative completeness, (ii) QBackchase ensures relative soundness, (iii) QChase and QBackchase eventually generate queries that satisfy consistency guarantee for associated constraints, while this constraint is not necessarily the original constraint. We clarified that this is a weaker guarantee, yet a quite pragmatic one considering the high cost checking the consistency of input constraints (typically Co-NP-hard). Indeed, our framework allows us to generate a query that is guaranteed to be consistent to a subset of constraints, even the input constraint may not be satisfiable.

**[M2]** *LLM Oracle Reliability (R1:W2, R2:W2).*

• **[R1:W2]** *The use of an LLM to judge answers can inflate recall or precision. It would be better if this work reports agreement with gold answers, shows variance across prompts, and adds ablations when the oracle is wrong.*

• **[R2:W2]** *The soundness and completeness guarantees are "relative" to LLM-generated reference answers. If the LLM oracle produces incorrect references, the guarantees become meaningless. The paper should more prominently acknowledge this limitation and analyze failure cases where LLM reference errors propagate.*

**[A2]** We addressed the following in our revision.

(1) We clarify a potential misunderstanding: although UniQGen uses an LLM oracle to obtain a reference set $A$ inside the Chase and Backchase loop, all reported EM/P/R/F1 results in Tables I–II are evaluated against the benchmark ground-truth answers (not against $A$). We added an explicit clarification sentence in Sec. IV-A. To address concerns about oracle reliability, we show that evaluator strength indeed matters. Specifically, we replace the LLM-generated reference set with ground truth on 100 GraphQ questions and observe an F1 improvement from 0.804 to 0.835 (Appendix V), indicating that oracle noise can propagate into the search process. We further add a concrete error-propagation example in Appendix III, illustrating how an incorrect reference set can lead to over-relaxation in QChase and the preservation of incorrect coverage in QBackchase. Finally, following reviwer's suggestion, we evaluate oracle stability by measuring model–ground-truth agreement on a random sample of 200 questions. The oracle achieves an average overlap rate above 80%. We also test several prompt designs (Appendix IV-D); among them, a mixture-of-experts (MoE) prompting strategy yields the strongest oracle quality, at the cost of increased token usage and latency.

(2) We agree that fact-checking is a critical and necessary Step to ensure a high-quality reference answer set. We have included a remark in Section II.C to acknowledge this limitation, and added an explicit *error propagation* case in Appendix III showing how an incorrect oracle (false positives/negatives in $A$) can force QChase to over-relax constraints and cause QBackchase to preserve incorrect coverage, yielding a final query that deviates from the intended semantics. We also included cases on how the reference set may be obtained in practice, which are supported by existing solutions. As fact-checking is itself a nontrivial task, we clarified that we shall leave it for future work to ensure the generated query will not only "be faithful" to the NL question, but also ensure high-quality answer. We think this is an interesting future topic. Thanks for the suggestion!

**[M3]** *Alignment of Baselines (R1:W3).*

• **[R1:W3]** *Some baselines target SPARQL only or rely on schema tuning. Authors could align entity linking, backends, and metrics, then mark any non comparable numbers as illustrative only.*

**[A3]** We added a unified evaluation protocol in Sec. IV-A for baseline alignment. All methods run on the same Neptune-hosted Freebase snapshot with consistent metrics (EM/P/R/F1) and the same evaluation script. UniQGen reuses Pangu 's entity-linking outputs to avoid confounding, and we clearly mark SPARQL-only baselines (ArcaneQA/Pangu) versus UniQGen's SPARQL+openCypher renderings. We also

clarify the methodological distinction between ArcaneQA/-Pangu, which are schema-aware and rely on KG schema/ontology tuning, and UniQGen and the Prompt-Only baseline, which are schemeless and derive constraints online from query-centric KG evidence.

**[M4]** *Add Real-World Use Case (R1:O2).*

• **[R1:O2]** *Add a real use case. Show an end to end scenario such as product catalog or clinical knowledge. Report P50 and P95 latency, accuracy against business labels, and total cost.*

**[A4]** We added a concrete end-to-end use case to illustrate UniQGen in an industry-motivated setting. In particular, the clinical scenario in Example 1 demonstrates the full pipeline from NL intent to an executable query, including resolving schema mismatch (*e.g.,* avoiding a non-existent edge such as `TREAT_ON` by matching the KG's `INDICATE_FOR` semantics) and injecting pragmatic constraints (*e.g.,* `Approved=True`) to return intent-aligned answers. For additional transparency, we also provide a step-by-step demonstration of the UniQGen pipeline in Appendix III. To complement benchmark-style reporting with operational metrics, we use WebQSP as a public proxy for user-facing industrial queries and evaluate UniQGen on our Neptune-hosted Freebase endpoint under production-style constraints. We verify "business labels" by re-executing the dataset-provided gold SPARQL queries on the same endpoint, with a successful re-execution rate of $89.69\%$, and report EM/F1 together with P50/P95 latency and per-query resource cost (token usage, average KG executions, and LLM calls) in Sec. IV-B. We also add a scalability analysis under query load on WebQSP (Fig. 4(c)), showing that UniQGen 's end-to-end generation latency remains stable up to at least 1500 concurrent requests under production-style constraints.

**[M5]** *Baseline Comparisons (R2:W1, R3:D2).*

• **[R2:W1]** *The paper cites GraphQ IR, noting it "remains training-based and lacks quality guarantees," but does not include it in experimental comparisons. Since both papers address multi-language query generation with intermediate representations, an experimental comparison would strengthen the evaluation and clarify practical differences between compilation-based (GraphQ IR) and rendering-based (UniQGen) approaches.*

• **[R3:D2]** *While the introduction mentioned different categories of KGQA approaches, the related work and experiment sections only discuss LLM-related methods. Referring to D1, there can be a more comprehensive comparison of effectiveness and efficiency with other methods, especially non-parametric or small model-based ones.*

**[A5]** We added more baseline compressions,

(1) Regarding R2:W1, GraphQ IR [1] that outputs a unified IR and compiles it to multiple query languages, which is different from UniQGen 's rendering-based, runtime-validated approach. We evaluate GraphQ IR on the overlapping benchmark it supports (GrailQA), and report the accuracy as following:

| Methods | I.I.D | Compositional | Zero-shot | Overall |
|---|---|---|---|---|
| UniQGen | 0.989 | 0.74 | 0.889 | 0.871 |
| GraphQ IR | 0.874 | 0.495 | 0.096 | 0.369 |

Due to the page limit, we put this table in the Appendix.V and provide a concise comparison in Sec.I. For GraphQ and WebQSP, GraphQ IR does not provide an official setting in its release; adapting it to these benchmarks would require constructing IR annotations (or an equivalent supervision signal) and training a new parser under their self-defined IR formalism. We therefore do not report GraphQ IR numbers on other benchmarks, such as GraphQ and WebQSP.

(2)Regarding R3:D2, our initial draft emphasized the LLM-enhanced KGQA line of work; we expanded the discussion to cover diverse KGQA paradigms beyond LLM prompting, including classical semantic parsing, non-parametric/case-based reasoning, neural-symbolic/GNN-based reasoning, and recent KG+LLM hybrids. Concretely, we collected representative methods with reported WebQSP results and summarize them in Table III, covering non-parametric or small-model baselines (e.g., CBR-KBQA, QGG, RNG-KBQA), GNN/neural-symbolic approaches (e.g., NSM, UniKGQA), and LLM-era KG-grounded systems (e.g., StructGPT, RoG-7B, G-Retriever). We also added a short appendix discussion that explains the categories and key trade-offs (training requirement, whether an executable query is produced, and metric differences such as F1/EM vs Hit@1), providing a clearer positioning of UniQGen relative to prior KGQA approaches.

**[M6]** *Address Latency Concerns (R2:W3, R3:D1).*

• **[R2:W3]** *At 40 seconds average per query, the paper claims to "outperform most existing approaches" in efficiency but does not report competitor latencies. For context, the ArcaneQA paper reports around 5s per query, suggesting UniQGen may be significantly slower than generation-based methods. The paper should provide clear head-to-head latency comparisons.*

• **[R3:D1]** *My main concern is the LLM time cost. Fig 4 reports an average query answering time of around 40 seconds. Is it reasonable in practical use? Please provide some elaboration or examples in realistic settings.*

**[A6]** We have revised the experimental study, and clarify the following on latency concerns.

(1) The reported $\sim$40s is a true *end-to-end* latency measured *from NL query to final answers*. Specifically, it covers entity linking, query-centric subgraph extraction, CTable construction, Chase/Backchase search (including LLM-based candidate generation and validation), and execution of candidate queries against the knowledge graph. Importantly, UniQGen achieves this latency *without any dataset-specific training or offline caching*, which significantly distinguishes it from many training-based KBQA systems, including ArcaneQA. In prac-

tical industry scenarios—where large-scale knowledge graphs (e.g., Amazon's product graph) continuously evolve—the substantial overhead from expensive dataset-specific training and storage-intensive offline caches significantly impairs adaptability. By contrast, UniQGen's training-free design and online query processing approach naturally support dynamic and large-scale industrial settings with minimal overhead.

(2) Regarding ArcaneQA, the cited 5.6s/question latency is measured under a specific evaluation protocol: an average over 1,000 randomly sampled queries in their online SPARQL-endpoint setup. This metric explicitly excludes the significant additional overhead associated with dataset-specific training and offline cache construction. Indeed, the ArcaneQA authors' own public reproduction discussions[1] confirm that runtime performance is heavily dependent on backend configuration, caching protocol, and endpoint locality. They acknowledge that even with pre-constructed SPARQL caches, training on Freebase can take around 12–20 hours per epoch, and inference on the full test set (around 13k queries) can take more than 10 hours or even days.

(3) Given these important differences, rather than making direct per-query comparisons across disparate protocols, we emphasize (i) our transparent end-to-end latency from query-to-answer (40s average on GrailQA, P50=20.12s), and (ii) the stable and fast KG execution latency for our generated final queries on a managed Neptune endpoint (Fig. 4(b): Minimal plan P50/P95=0.25s/0.28s; Universal plan $<$1s at P95).

(4) We have also revised the analysis of Fig.4 to claim that our methods outperform the baselines, replacing "most existing approaches", to avoid confusion. Thanks for catching this!

**[M7]** *Notation Consistency (R3:D3).*

• **[R3:D3]** *The query-induced subgraph is defined as $G'$ in Sec 3.A, while other sections use $Q$ and $Q(G)$ without specifying $G'$. Similar issues apply to the extracted nodes set $V'$.*

**[A7]** We thank the reviewer for pointing out a potential ambiguity between the query-centric extracted subgraph ($G'$) and the full KG ($G$) used for answering. We have revised our paper to remove this inconsistency.

(1) In our framework, $G'$ (with extracted nodes $V'$) is an intermediate artifact used only to gather local facts and construct the CTable $\mathcal{C}$, as described in Sec. III-A-III-B and Fig. 2. All generated graph queries are executed on the original knowledge graph $G$; accordingly, $Q(G)$ always denotes the answer set by executing a generated query $Q$, consistent with the definitions in Sec. II. To improve readability, we have also provided a notation glossary in Appendix I.

(2) To further remove any confusion, in the revision, we (i) added a concise clarification in Sec. III-A explicitly stating that $G'$ is used only for CTable construction while query execution is on $G$, and (ii) added $G'$ and $V'$ to the notation glossary in Appendix I.

**[M8]** *Elaborate Design Intuitions (R3:D4).*

[1]https://github.com/dki-lab/ArcaneQA/issues/5

• **[R3:D4]** *In Alg 1 line 10, why "new candidates are spawned by removing one constraint at a time from $C$"? What is the design intuition and possible outcome of removing a constraint?*

**[A8]** Alg. 1 performs a top-down beam search over subsets of constraints in the scored CTable (normalized to the monotone core). When a candidate fails the completeness check, we relax it by *removing constraints*. Under the monotone-core normalization, removing a constraint can only enlarge the retrieved answer set, making it a principled way to recover missing oracle answers and improve relative completeness. We remove *one* constraint per expansion to enumerate minimal relaxations: this preserves selectivity (avoiding overly aggressive relaxation), helps isolate which constraint blocks coverage, and keeps branching controllable under a fixed beam budget.

We stated the monotonicity property in Sec. II (Constraints). To make it clearer, we added an explicit remark after the Outline of Alg. 1, explaining why dropping constraints is a principled relaxation for recovering completeness, why we drop one constraint per step (minimal relaxations and tractable branching under a fixed beam budget), and how we mitigate the precision/cost trade-off via Eq. 1 and subsequent QBackchase tightening. We also provide a more comprehensive explanation in Appendix. II-C with a worked demonstration in Appendix. III.

**[M9]** *Why Big Gains on GraphQ (R3:D5)?*

• **[R3:D5]** *Why is the model improvement on the GraphQ dataset more significant than others? It is suggested to conduct more specific studies to prevent possible defects.*

**[A9]** We agree that the improvement on GraphQ is large and added two clarifications/analyses in the revision. First, we report dataset split statistics and train:test ratios across all three benchmarks in Sec.IV-A. GraphQ has the smallest train:test ratio (2,381/2,395 $\approx$ 0.99), compared to WebQSP (3,098/1,639 $\approx$ 1.89) and GrailQA (44,337/6,763 $\approx$ 6.56), which leads to the fact that training-based baselines are the most constrained on GraphQ, while UniQGen is training-free and therefore benefits most in this low-resource regime. Second, we added a fine-grained GraphQ breakdown by query complexity (Fig. 3(c)), showing the improvement is consistent across buckets and is most pronounced on the more complex questions. We also explicitly confirm that the gain persists under both SPARQL and Cypher renderings (Table. II), mitigating concerns about renderer-specific artifacts. We updated Sec. IV-B with these analyses.

**[M10]** *Discuss Practical Challenges (R3:W4).*

• **[R3:W4]** *From an industry perspective, the paper would benefit from additional discussion of practical graph query generation challenges and evaluation on real industrial workloads.*

**[A10]** We already outlined key practical challenges in Sec. I (before Example 1), which including hallucination/misalign-

ment, missing pragmatic constraints, high maintenance costs, and the lack of a unified, principled solution across diverse query languages, and discussed how UniQGen addresses them in Sec. III-D via query-centric constraint extraction, Chase/Backchase with execution-based validation, and deployment guardrails (bounded beam search, batching, caching, and timeouts). To clarify, we added a summary at the end of Sec. III-D that enumerates these practical pain points and the corresponding design/guardrails.

For industry workload evaluation, we complement benchmark metrics with operational measurements on a Neptune-hosted Freebase endpoint using WebQSP as a public proxy for user-facing industrial queries (Sec. IV-B). We report accuracy against verified labels (gold queries re-executed on the same endpoint) and P50/P95 latency, along with per-query resource/-cost proxies. We also include a scalability analysis under query load (Fig. 4(c)), demonstrating stable end-to-end latency up to at least 1500 concurrent requests under production-style constraints.

# Graph Query Generation with Constraint-guided Large Language Agents

Mengying Wang*[1], Nicolaas Paul Jedema[2], Rahul Pandey[2], RaviKiran Krishnan*[3], Jens Lehmann[2,4], Yinghui Wu[1]
[1]Case Western Reserve University; [2]Amazon AGI; [3]Meta; [4]Technische Universität Dresden
Email: {mxw767, yxw1650}@case.edu; {jedema, panrahu, jlehmnn}@amazon.com; ravi.krishnan@outlook.com

***Abstract*—Knowledge Graph Question Answering (KGQA) has advanced through structured query generation, yet most efforts target RDF/SPARQL, leaving Cypher and property graphs underexplored, despite increasing demand for unified KGQA in industry settings. We propose UniQGen, a novel constraint-based framework that employs LLM agents to dynamically extract and refine representative graph query clauses into executable, intent-aligned graph queries across query languages. The foundation of our method is a variant of Chase & Backchase, a family of algorithms for query optimization and reformulation. We extend Chase & Backchase with a dynamic reasoning process over query constraints that also interact with LLMs for query quality estimation. With a Cypher-supported Freebase graph deployed on Amazon Neptune, we extensively evaluate our approach on popular KGQA benchmarks (GraphQ, GrailQA, and WebQSP). We demonstrate that UniQGen outperforms state-of-the-art graph query generation techniques in both accuracy and efficiency, with F1 gains of 31.6% on GraphQ and 4.9% on GrailQA. Unlike prior methods, our framework does not require fine-tuning for schema matching, making it more extensible to schema-less graphs and semantics in query workloads, and is more suitable for enterprise-grade KGQA. We release Cypher outputs and a Neptune-ready Freebase snapshot to support reproducible, cross-language KGQA research.[1]**

***Index Terms*—query generation, knowledge graph, question answering, LLM application**

## I. Introduction

Graph querying have been routinely applied to support various tasks such as knowledge-graph question answering (KGQA). Current solutions have been deployed for two isolated scenarios: the Resource Description Framework (RDF) model, queried with SPARQL; or the Labeled Property Graph (LPG) model, queried by Cypher/Gremlin. This often leads to a "graph–model lock-in" effect [2]: once an organization commits to a paradigm, that choice propagates to the query language, toolchain, and developer expertise, making later migration costly and risky. In response, recent efforts advocate a unified interface that can cope with different data models, including RDF, RDF*, and LPG [3], [4]. For example, Amazon Neptune's OneGraph [2] advocates a unified store exposed through multiple query languages. Recent openCypher-over-RDF demonstrations [5] offer a storage-level bridge that allows Cypher syntax to query RDF data.

To enable a unified interface for KGQA, it also calls for a "query-level" effort: a query generation mechanism that can automatically convert user intent to mainstream graph query classes such as Cypher or SPARQL, mitigating "language lock-in" effect. Most public KGQA benchmarks support a single query class, such as RDF/SPARQL-centric (e.g., GrailQA [6], WebQSP [7]). This leaves a gap between academic solutions and industrial practice, despite languages such as Cypher underpins many production stacks including popular graph databases such as Neo4j [8].

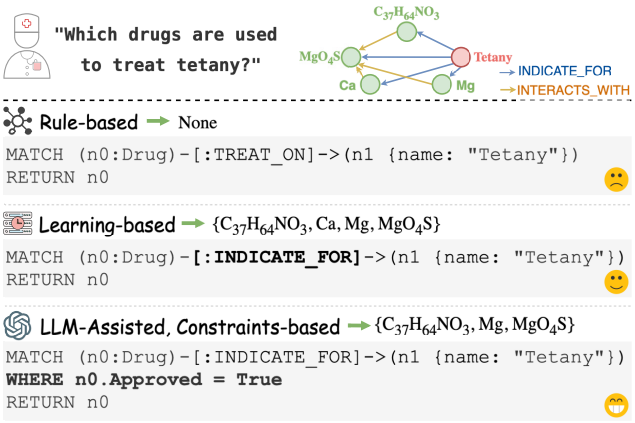


Fig. 1: Given a query: *"Which drugs are used to treat tetany?"* **Rule-based** methods assume a non-existent relation and returns None; **Learning-based** methods reach the disease but over-includes Ca, failing to address pragmatic constraints; **LLM-assisted, constraint-based** methods generation captures implicit intent and respects ontology, yielding clinically appropriate answers.

Meanwhile, LLM-based methods become attractive for graph query generation due to their strong capabilities in both contextual understanding and code generation [9]–[12], achieving state-of-the-art results on multiple SPARQL benchmarks [13], [14]. Nevertheless, in industrial settings they exhibit well-known challenges: (i) *Hallucination/misalignment*: syntactically valid queries do not match the KG's schema/-topology (top row of Fig. 1); (ii) *Missing pragmatic constraints*: queries fails to return answers for desired search intent, such as "approved drug" (middle row of Fig. 1); (iii) *High maintenance cost*: heavy rely on schema-specific fine-tuning makes updates and migration of solutions brittle and costly, especially for large ontologies and KGs (e.g., Freebase/Wikidata); (iv) *Lack of unified, principled solution*: training and maintaining separate models per query language is costly, and cold-start for understudied languages like Cypher hinders cross-language adoption and fair comparison.

* Work done while at Amazon AGI.
[1]Resources will be released after passing internal reviews at Amazon.

In response, we propose UniQGen, a schemaless, training-free solution that turns Chase and Backchase algorithm into a budgeted, cross-language query generation pipeline by introducing an online-constructed, uncertainty-scored constraint IR (CTable) and an LLM-assisted pluggable renderer, rather than relying on offline dependencies or training; it offers relative guarantees *w.r.t.* a fixed reference answer set $A$ (user- or oracle-provided) by accepting candidates only after runtime validations.

**Example 1:** As shown in Fig. 1, for query "Which drugs are used to treat tetany?", the rule-based methods directly extract semantic information from the natural language query, which assumes a direct `TREAT_ON` edge returns no result when the KG encodes indications as `INDICATE_FOR`; learning-based methods capture the schema through costly training processes, while they may identify `INDICATE_FOR`, still output an over-inclusive set because it lacks implicit constraints (e.g., approval status). In contrast, the constraint-based LLM-assisted method constructs a compact constraint table that (i) fixes the topology with typed edges, and (ii) makes the implicit intent explicit (`Approved=True`); a Chase & Backchase loop then prunes irrelevant atoms and returns a precise, executable query.

**Contributions.** We summarize our contributions as follows:

- **Deployment-friendly unified pipeline** (UniQGen)**.** We design an autonomous pipeline that unifies constraint extraction, CTable construction, and a Chase & Backchase-based query refinement process to generate grammatically-correct, semantically reasonable queries with low generation latency, for schema-less KGQA.
- **LLM-assisted constraint-based reformulation method.** We adapt Chase & Backchase to graph query generation: LLM agents explore and rank candidate constraints via beam search, while Chase & Backchase enforces relative completeness, soundness, and minimality. It extends to other query languages with modest effort.
- **Experiments and Resources.** We build on Amazon Neptune with a deployed Freebase endpoint that supports Cypher execution for fair cross-language comparison with SPARQL-only SOTA. We release a well-curated Freebase snapshot with a one-step deployment recipe on Amazon Neptune to enable rapid, large-scale evaluation [15], along with gold Cypher pairs aligned with popular SPARQL benchmarks to facilitate future research.

**Related Works** We categorize the related works as follows:

*Graph Query Generation*. Recent efforts most leverage LLMs to improve the quality of SPARQL query generation [16], [17]. However, existing approaches often require large training data and are tied to knowledge graph schemas, limiting their adaptability [18]. Cypher query generation for property graphs has received less attention [1]. Our approach addresses unified query generation in a schemaless and training-free manner, making it adaptable to a wide range of querying scenarios, and refined by Chase & Backchase algorithm [19] with quality guarantees. This is not addressed in prior methods.

*LLM-enhanced KGQA*. LLMs have been used to structure NL inputs, perform multi-step reasoning, and produce executable queries in KGQA area [20], [21]. These systems typically emphasize semantic parsing quality on SPARQL-centric benchmarks. Inspired by [22], UniQGen employs LLM agents not just for static query generation but also for dynamically extracting constraints from NL queries and interacting with KGs. This allows for more interactive, contextually aware, and refined query outputs, ensuring high-quality results.

*Unified KGQA*. There is growing demand, especially in industry, for KGQA systems that operate seamlessly across RDF/SPARQL and LPG/Cypher stacks, yet most efforts focus on data-model [3], [4] or engine-level [2] unification, but lack a light-weight, modular way to transfer NL intent into executable queries across languages. GraphQ IR [1] proposes an intermediate representation(IR) compiled into multiple query languages, but it remains training-based and lacks quality guarantees. Emerging Text2Cypher benchmarks [23] begin addressing Cypher generation, yet broad, fair cross-language evaluation remains limited. Our work fills this gap by introducing a constraint-based IR refined through Chase & Backchase within an LLM -assisted agent pipeline, deploying Freebase graph on Amazon Neptune to activate mature SPARQL benchmarks for Cypher query generation and evaluation.

## II. Query Generation under Constraints

**Knowledge Graphs and Queries.** Let $G = (V, E)$ be a knowledge graph with entities $V$ and typed edges $E$ represented as triples $\langle s, p, o\rangle$. A graph query $Q$ (e.g., Cypher) denotes a conjunctive pattern with filters; its answer set $Q(G)$ is the set of bindings that satisfy the pattern. We assume a referenced LLM oracle that provides a *reference answer* set $A$ and its grounding $G(A) \subseteq V$ for the given NL query $Q_n$.

**Constraints.** We introduce a class of *constraint tables* (CTable) as a concise, language-agnostic abstraction for graph queries, which contains: (i) *typed triple patterns* $\langle c_s, c_p, c_o\rangle$, where entries may be constants or variables; (ii) *value constraints* on literals (e.g., $x \,\mathtt{op}\, c$, $\mathtt{op} \in \{=, >, <, \geq, \leq, \neq\}$); and (iii) *structural constraints*, such as joins, `OPTIONAL`/`UNION`, bounded paths, etc. In practice, CTable also captures *implicit/pragmatic* conditions not explicit in the NL query (e.g., `Approved=True` in Fig. 1). Each constraint $c$ can carry an uncertainty score $u \in [0, 1]$ reflecting specificity to guide search; constraints with no matches are pruned.

**Quality Measures**. We consider *relative* criteria w.r.t. the reference answers. We say a generated query $Q$ is (i) LLM-*sound*, iff. $Q(G) \subseteq G(A)$; (ii) LLM-*complete*, iff. $G(A) \subseteq Q(G)$. (iii) *Consistency* iff. every binding in $Q(G)$ satisfies all $c \in \mathcal{C}$. For monotone conjunctive queries, *removing* constraints enlarges the answer set (hence may improve the answer completeness, yet by sacrificing soundness) while *adding* constraints makes answers more specific (hence may improve soundness but risks sacrificing completeness).

**Problem Formulation.** Given an NL query, graph $G$, a scored CTable $\mathcal{C}$ extracted from the NL query, and a reference set

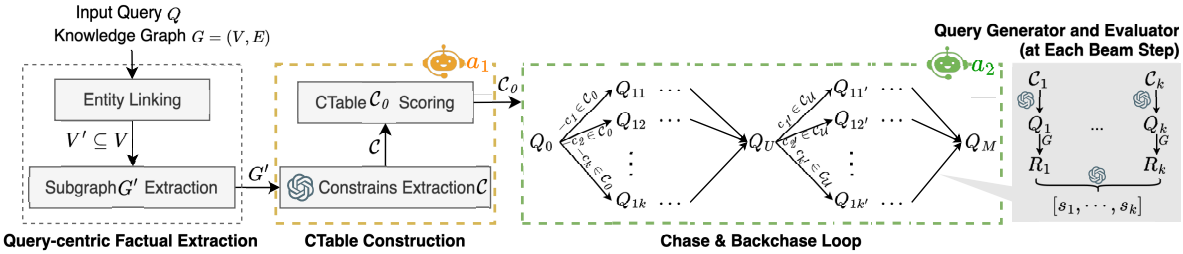


Fig. 2: UniQGen Framework Overview

$A$, we aim to generate a graph query $Q$ that (i) is *consistent* with $\mathcal{C}$; (ii) jointly maximizes LLM-*completeness* and LLM-*soundness* (reported as EM/F1); and (iii) is *minimal* (no redundant constraints).

The foundation of our proposed unified solution is to search over subsets of $\mathcal{C}$ with a beam strategy, organized in three major phases: *Chase* phase (relaxation for completeness), starting with a universal query that enforces all constraints, and *Backchase* (tightening for soundness/minimality), terminating at minimal queries, such that adding one more constraint compromises soundness; and *Rendering* the selected logical plan (conjunctive constraints) to the target query language.

## III. UniQGen: Unified Query Generation Framework

UniQGen turns user intent into an executable query in three stages: (i) query-centric fact extraction; (ii) CTable construction; and (iii) LLM-assisted Chase & Backchase, followed by LLM-prompted or deterministic rendering to the target query language. Fig. 2 shows the end-to-end pipeline, with Agents $a_1$ and $a_2$ orchestrating CTable construction and Chase & Backchase, respectively.

### A. Query-centric Fact Extraction

Given an input NL query $Q_n$ and a knowledge graph $G = (V, E)$, UniQGen first performs entity linking to obtain mentioned nodes $V' \subseteq V$ (using off-the-shelf entity linkers [24] or fact extractors [25]). For each $v \in V'$, it induces a local subgraph $G'(v)$ from the $k$-hop neighborhood (typically $k \leq 3$), and unions them to form $G' = \bigcup_{v \in V'} G'(v)$. The induced subgraph $G'$ serves as the factual context for CTable construction; queries are executed on $G$ evaluated using $Q(G)$.

### B. CTable Construction

**Constraint Table Extraction**. Agent $a_1$ constructs CTable $\mathcal{C}$ by extracting relevant triples from $G$, focusing on the obtained subgraph $G'$. $a_1$ samples and parameterizes triples from $G'$ based on their relevance to the input query $Q_n$. The triples are chosen for their ability to define meaningful relations. When hinted by $Q_n$, $a_1$ augments $\mathcal{C}$ with *implicit/pragmatic* constraints (e.g., `Approved=True` in Fig. 1).

To standardize processing, we normalize CTable into a *monotone core* that retains only positive atoms, constant filters, and bounded path existence; we rewrite `OPTIONAL` as explicit branches and keep `UNION` as separate branches, while deferring non-monotone constraints (negation/anti-join, at-most/exactly $k$, aggregates, `ORDER/LIMIT`) to post-validation.

**Constraint Table Scoring**. For each constraint $c_i \in \mathcal{C}$, Agent $a_1$ assigns an *uncertainty score* $u_i \in [0, 1]$ that reflects specificity: let $n_i$ be the number of matches of $c_i$ in $G$ and $m = \max_{c_j \in \mathcal{C}} n_j$, then $u_i = \frac{n_i}{m}$. Higher $u_i$ indicates a less specific (more permissive) constraint; constraints with $n_i = 0$ are pruned. These one-time scores guide beam priorities in the Chase and Backchase phase (Sec III-C). We present the detailed procedure and cost analysis in Appendices II-III [26].

### C. Chase and Backchase Loop

Given a scored constraint table $\mathcal{C}_0$ (normalized to the monotone core), and an oracle reference set $A$ for the NL query $Q_n$, UniQGen runs a two-phase generation (*Chase* and *Backchase* [19]) to balance LLM-*completeness* and LLM-*soundness*, while preserving *minimality*. Agent $a_2$ comprises (i) a Query Generator that renders a candidate query from a constraint set; and (ii) an Evaluator that scores candidates against $A$, and applies post-checks for any non-monotone constraints. To reduce cost and latency, all candidates in a beam step are evaluated in a *single* batched LLM call.

**Query Chase Phase**. Given a CTable $\mathcal{C}_0$ constructed from the input natural language query $Q_n$, Agent $a_2$ performs the *Query Chase* phase to iteratively refine an initial graph query $Q_0$ derived from the full CTable $\mathcal{C}_0$, using beam search to explore candidate queries and remove unnecessary constraints. The objective of QChase is to derive a *universal query* $Q_U$ that satisfies all necessary constraints in $\mathcal{C}_0$, ensuring the consistency and relative completeness. QChase coordinates beam search with a dynamically maintained search tree, where each node $v_Q$ refers to a candidate query $Q$ with the associated structure that tracks the following. (1) retrieved answer $r$, (2) a fraction of CTable $Q.\mathcal{C}$ that contains the constraints yet to be enforced to refine the query $Q$, (3) a performance score $p$ to be estimated by Evaluator, and an overall score quality $s$ to be calculated for guiding the beam search.

At each node $v_Q$, QChase spawns a set of children in the search tree, each referring to a new candidate query $v'_Q$ that is obtained by removing a constraint (a triple pattern) from $Q.\mathcal{C}$. The score $s'$ is then estimated for $Q'$ as

$$s'(c, Q) = \alpha(1 - c.u) + (1 - \alpha)Q.p \quad (1)$$

Where $c.u$ is the uncertainty score of $c$, $Q.p$ is the performance score of the parent query, and $\alpha \in [0, 1]$ is a weighting factor to balance $c.u$ and $Q.p$. Indeed, the less constraints a query $Q$ poses, the more likely it "covers" ground truth, yet at a cost of introducing more entities, and less precise answer.

**Algorithm 1** QChase: completeness-guided beam search

**Input:** NL Query $Q_n$; CTable $\mathcal{C}_0$; Query Generator $\gamma$; Evaluator $\varepsilon$; Beam width $b$; Threshold $\tau$; Graph $G$
**Output:** Universal Query $Q_U$.

1: **set** $B := \varnothing$, $L := \varnothing$;
2: **initialize** $s_0 := 0$, $B$.append($(\mathcal{C}_0, s_0)$);
3: **while** $B \neq \emptyset$ **do**
4: **for** $(\mathcal{C}, s) \in B$ **do**
5: $Q := \text{Gen}(\gamma, \mathcal{C})$, $r := Q(G)$, $p := 0$;
6: $L$.append($(\mathcal{C}, Q, r, p, s)$);
7: $L := \text{Eva}(\varepsilon, L, Q_n)$;
8: $l_U := \arg\max_{l \in L} l.p$, $Q_U := l_U.Q$;
9: **if** $l_U.p \geq \tau$ **then** break;
10: $B := \{(l.\mathcal{C} \setminus \{c\}, s(c, l)) \mid l \in L, c \in l.\mathcal{C}\}$;
11: **if** $|B| \leq b$ **then** continue;
12: $B := \{(\mathcal{C}, s) \in B \mid \text{top } b \text{ by } s\}$, $L := \varnothing$;
13: **return** $Q_U$

*Outline*. The procedure executed by Agent $a_2$ is detailed in Alg. 1. The beam $B$ encodes an initial query with all constraints in $\mathcal{C}_0$ as a conjunctive query, with the full CTable $\mathcal{C}_0$ and a score $s_0 = 0$ (lns 1-2). For each candidate $(\mathcal{C}, s) \in B$, Query Generator $\gamma$ generates a query $Q$, which is executed on $G$ to retrieve results $r$ (lns 4-5). List $L$ is used to maintain all the relevant information for the candidate queries as a label $(\mathcal{C}, Q, r, p, s)$, where $p$ is initially a placeholder (ln 6). After all candidates are processed, Evaluator $\varepsilon$ (1) verifies the LLM-relative completeness *w.r.t.* a reference answer set from an LLM oracle , as a hard constraint, and (2) assigns performance scores $p$ for each query (ln 7). If any $l.p$ exceeds the threshold $\tau$, the corresponding query $l.Q$ is returned as the universal query $Q_U$ (lns 8-9). Otherwise, new candidates are spawned by removing one constraint at a time from $\mathcal{C}$ and calculating new scores $s$ for each descendant (ln 10). The candidates with top $b$ scores $s$ are selected for next beam step (lns 11-12). Stop when a candidate reaches $p \geq \tau$ or the beam is exhausted; the best candidate so far is returned as the *universal* plan $Q_U$.

*Remarks*. Since CTable is normalized to the monotone core (Sec. II), dropping a constraint relaxes the candidate query and can enlarge its answer set. Therefore, when a candidate fails the completeness check against the reference set $A$, removing constraints is a principled mechanism to recover missing answers. We drop *one* constraint at a time (Alg. 1, ln 10) to enumerate *minimal* relaxations and keep the branching factor controllable under a fixed beam budget. Relaxation may introduce spurious answers and larger intermediate results; we mitigate this via uncertainty-aware beam scoring (Eq. 1) and subsequently tighten the universal query in QBackchase.

**Query Backchase Phase**. In QBackchase, Agent $a_2$ follows a bottom-up approach, starting with subqueries derived from the universal query $Q_U$, where each subquery corresponds to exactly one constraint from the CTable $Q_U.\mathcal{C}$. These subqueries serve as the building blocks for constructing minimal queries that satisfy the relative soundness *w.r.t.* the answer set $A$.

*Outline*. For each constraint $c_i \in Q_U.\mathcal{C}$, a subquery $Q_i$ is generated to cover only the single constraint $c_i$. Similar to QChase (Alg. 1), in *Backchase Phase*, Agent $a_2$ employs beam search to iteratively combine these subqueries. At each step, constraints are reintroduced from the subqueries, and candidate queries are evaluated based on their ability to produce valid results on the knowledge graph $G$. Evaluator $\varepsilon$ is responsible for evaluating the soundness of the answer sets retrieved by candidate queries. The score $s$ is calculated (Eq. 1) to guide the beam search and select the most promising candidates.

We present the full procedure in Appx. II, and showcase a complete illustration in Appendix III [26].

**Quality guarantee**. The QChase and QBackchase together as an integrated framework eventually guarantee the consistency. The QChase phase (resp. QBackchase) phase ensures relative completeness (resp. soundness) for refining the generated queries *w.r.t.* answer set $A$, provided either by the $NL$ queryer or generated by the LLM agent. These guarantees are ensured by guarding the correctness and completeness (resp. soundness) of the retrieved answers against $A$ at runtime. Moreover, among all validated complete Cypher queries, the one that maximizes (resp. minimize) the number of constraints in $\mathcal{C}$ will be returned, ensured by the "top-down"(resp. "bottom-up") strategy of UniQGen.

*Remarks*. The above quality guarantees are relative to a proper reference set. The latter can be obtained from crowd-sourced annotation, query-by-example, or querying high-quality master data. Fact-checking remains necessary yet out of scope of this work, and we leave it for future work.

### *D. Deployment Considerations*

**Runtime Environment & Integration.** Our framework runs on AWS EC2 `c5.4xlarge` instances (16 vCPU, 32 GiB) behind an autoscaling group. We deployed Freebase (N-Triples, original support SPARQL only) on Amazon Neptune, supported by Neptune Analytics's MLM [5] feature, enabling unified evaluation across both openCypher and SPARQL against the same graph using deployed endpoints without duplicating logic. Agents are model-agnostic and implemented with LangChain. We invoke LLMs through Amazon Bedrock, which allows us to swap model families without code changes and provides an interface for inferring OpenAI models.

**Adaptation & Portability.** As UniQGen is *training-free*, adapting to a new KG or schema drift requires only updating synonym/hint lists; the agents will discover the updated knowledge bases at runtime as described. The query rendering layer is pluggable: the same plan can be emitted as Cypher or SPARQL via a mapping set. When any constraint lacks a direct mapping, we fall back to prompt-based code generation and gate acceptance via answer-set validation, which keeps cross-language evaluation feasible.

**Cost Control & Guardrails.** To keep per-query cost predictable and tail latency bounded, UniQGen (i) batches Evaluator into one LLM call per beam step for fairness

and efficiency scoring; (ii) parallelizes candidate generation and execution within each beam step; (iii) bounds beam width/depth and adapts them using early completeness signals; (iv) short-circuits empty/degenerate plans; and (v) caches entity links, $k$-hop subgraphs, and reference answer sets. We also enable beam parameters auto-tuning to fit the cost budget.

**Onboarding Artifacts.** We provide a one-step deployment recipe (IaC) and a well-curated graph snapshot to load Freebase into Amazon Neptune, build indices, and provision endpoints; a harness to run common Freebase-based KGQA benchmarks in *dual-language* mode (Cypher/SPARQL); and *gold Cypher pairs* aligned with popular SPARQL benchmarks to reduce cold-start for LPG/Cypher-based KGQA.

**Summary.** These considerations translate the industrial pain points in Sec. I into concrete guardrails: (i) *integration and portability* via Neptune dual-language endpoints and model-agnostic agents; (ii) *robustness under schema mismatch/drift and LLM mis-grounding* via query-centric extraction, a pluggable rendering layer, and CaB refinement; and (iii) *predictable latency/cost under SLAs* via batched evaluation, parallel execution, bounded beam width/depth with early stopping, short-circuiting degenerate plans, and caching. The provided IaC recipe and benchmark harness further reduce cold-start and maintenance overhead in practice.

## IV. Experiment Study

### A. Experiment Settings

**Datasets**. To mirror the real-world enterprise KGQA scenarios our Amazon team targets - querying production-scale graphs, we select the Freebase as knowledge base (2.9B triples, 116M entities [27]) and adopt three well-established benchmarks that span everyday, multi-hop, and schema-diverse questions: (i) GraphQ [28], a characteristic-rich testbed with 2,381 training and 2,395 test questions (train:test $\approx$ 0.99); (ii) GrailQA [6], covering three generalization settings (i.i.d., compositional, zero-shot), with 44,337 training questions and a dev split of 6,763 questions (train:test $\approx$ 6.56); and (iii) WebQSP [7], with 3,098 training and 1,639 test questions (train:test $\approx$ 1.89). As UniQGen is training-free, we use only the test sets, which consist of 2395, 6763, and 1639 questions, respectively.

**Methods**. (i) UniQGen: our unified pipeline, where "*Universal*" refers to the results from the universal query, "*Minimal*" refers to the minimal query, and "*Optimal*" selects the better of the two per question, evaluated with GPT-4o; (ii) Prompt-Only: a prompt-only variant using the same prompt as Query Generator in Agent $a_2$, evaluated with GPT-4o; (iii) ArcaneQA [29]: a generation model combining program induction and contextual encoding, SPARQL only; and (iv) Pangu [13]: a state-of-the-art framework with a symbolic agent for exploration and a neural agent for evaluation, models fine-tuned with specific benchmarks and schema, SPARQL only. To ensure a fair comparison and isolate entity-linking effects, we adopt the same entity-linking results as Pangu.

**Evaluation Metrics** We evaluated the quality of generated queries with the following metrics, against ground truth answers provided in benchmarks: (i) ExactMatch, the proportion of queries that return exactly the same answer as the ground truth; (ii) Precision(P), Recall(R), and F1: *Average* scores for all queries in terms of their completeness and soundness of the returned answers; and (iii) query-generation latency, measured in per-query time, reported as median (P50) and tail (P95) metrics. P50 denotes the median latency, and P95 is the value below which 95% of the queries complete.

**Evaluation protocol.** All methods query the same Neptune-hosted Freebase snapshot. ArcaneQA and Pangu are SPARQL-only baselines, while UniQGen is evaluated under both SPARQL and openCypher renderings. We compute EM/P/R/F1 on answer sets using the same evaluation script for all settings. To reduce confounding effects from entity linking, UniQGen reuses the off-the-shelf entity-linking outputs provided by Pangu. ArcaneQA and Pangu are schema-aware methods that are tuned with KG schema/ontology, whereas UniQGen and the Prompt-Only are schemeless methods. All reported scores are evaluated against the benchmark ground-truth answers rather than $A$.

### B. Results and Analysis

**Effectiveness & Query Quality**. We summarize query quality results in Table I (GrailQA) and Table II (GraphQ/WebQSP). UniQGen consistently surpasses baselines across all datasets and query types, notably on GraphQ, highlighting its effective constraint exploration without requiring costly model training or fine-tuning. This efficiency and adaptability showcase its industrial scalability for diverse query scenarios.

*Accuracy Variation Across Datasets.* UniQGen sees the largest gains on GraphQ, the most low-resource benchmark among the three, with the smallest train:test ratio ($\approx$ 0.99). This setting constrains training-based baselines more severely, while UniQGen remains training-free and thus generalizes better. We further diagnose GraphQ by query complexity (Fig. 3(c)): while all methods degrade as hops increase, UniQGen degrades more gracefully than Pangu, and the gap widens on multi-hop questions, indicating that constraint-guided search and execution-based validation particularly help compositional queries under limited supervision. The trend also remains consistent across both renderings, mitigating concerns that the gain is renderer-specific. In contrast, WebQSP is more diverse and amenable to in-context learning, shows UniQGen achieving results comparable to or slightly below Pangu. On GrailQA, UniQGen consistently leads on I.I.D. and Zero-shot subsets but slightly trails on Compositional due to complex reasoning, which benefits more from few-shot adaptation. Interestingly, the winning renderer flips by dataset: Cypher leads on GraphQ while SPARQL leads on WebQSP. This aligns with schema bias: GraphQ 's patterns map cleanly to Cypher's labeled-node/edge model and bounded relationship lengths, while WebQSP frequently touches Freebase's RDF specifics (mediator/CVT nodes, directional predicates, typed literals), which are most naturally expressed with SPARQL triples and property paths.

| Method | I.I.D. | | | | Compositional | | | | Zero-shot | | | | Overall | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | R | F1 | EM | P | R | F1 | EM | P | R | F1 | EM | P | R | F1 | EM |
| Prompt-Only | 0.291 | 0.293 | 0.291 | 0.276 | 0.300 | 0.312 | 0.303 | 0.277 | 0.313 | 0.313 | 0.313 | 0.310 | 0.305 | 0.308 | 0.306 | 0.295 |
| ArcaneQA | 0.887 | 0.896 | 0.887 | 0.869 | 0.744 | 0.763 | 0.739 | 0.690 | 0.728 | 0.741 | 0.727 | 0.706 | 0.769 | 0.782 | 0.767 | 0.741 |
| Pangu | 0.892 | 0.900 | 0.891 | 0.869 | 0.790 | 0.803 | **0.787** | 0.738 | 0.827 | 0.835 | 0.826 | 0.812 | 0.834 | 0.843 | 0.833 | 0.809 |
| UniQGen-Universal | 0.908 | 0.909 | 0.903 | 0.886 | 0.763 | 0.779 | 0.747 | 0.702 | 0.896 | 0.895 | 0.893 | 0.885 | 0.869 | 0.872 | 0.863 | 0.844 |
| UniQGen-Minimal | 0.903 | 0.905 | 0.898 | 0.883 | 0.740 | 0.759 | 0.727 | 0.682 | 0.885 | 0.886 | 0.882 | 0.874 | 0.857 | 0.862 | 0.851 | 0.833 |
| UniQGen-Optimal | **0.914** | **0.912** | **0.908** | **0.893** | **0.792** | **0.807** | 0.779 | **0.740** | **0.900** | **0.899** | **0.898** | **0.890** | **0.879** | **0.881** | **0.874** | **0.857** |

TABLE I: GrailQA results by split; compare UniQGen in Cypher with SPARQL-only methods; best per column in **bold**, 2nd best underlined.

| Query Language | Method | GraphQ | | | | WebQSP | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | P | R | F1 | EM | P | R | F1 | EM |
| SPARQL | ArcaneQA | 0.346 | 0.400 | 0.343 | 0.320 | 0.753 | 0.781 | 0.748 | 0.706 |
| | Pangu | 0.614 | 0.655 | 0.610 | 0.584 | 0.792 | 0.821 | 0.793 | 0.758 |
| | UniQGen-Universal | 0.750 | 0.789 | 0.756 | 0.740 | 0.785 | **0.937** | **0.794** | 0.648 |
| | UniQGen-Minimal | 0.753 | 0.792 | 0.759 | 0.744 | 0.785 | **0.937** | **0.794** | 0.647 |
| | UniQGen-Optimal | 0.763 | 0.805 | 0.769 | 0.752 | 0.785 | **0.937** | **0.794** | 0.648 |
| CYPHER | Prompt-Only | 0.221 | 0.225 | 0.223 | 0.214 | 0.290 | 0.270 | 0.270 | 0.260 |
| | UniQGen-Universal | 0.778 | 0.810 | 0.773 | 0.740 | – | – | – | – |
| | UniQGen-Minimal | 0.747 | 0.776 | 0.744 | 0.714 | – | – | – | – |
| | UniQGen-Optimal | **0.809** | **0.835** | **0.803** | **0.775** | **0.793** | 0.818 | 0.788 | **0.781** |

TABLE II: GraphQ & WebQSP results across SPARQL and Cypher; all queried over the same Neptune-managed endpoint; best per column in **bold**, second best underlined.

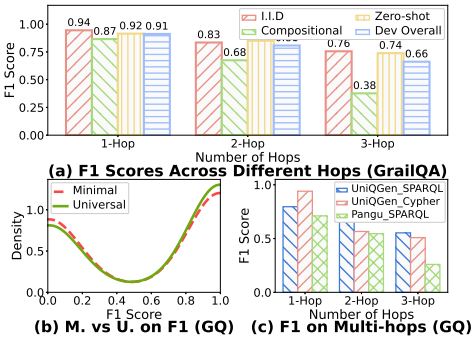


Fig. 3: Effectiveness & Query Quality

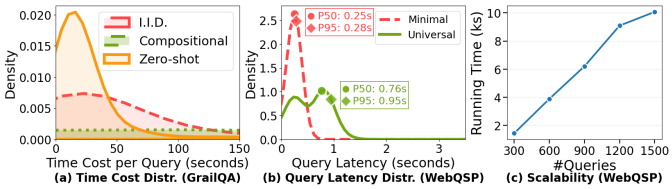


Fig. 4: Efficiency Analysis

*Complexity of Input Query Classes (Multi-Hop).* Fig. 3(a) shows declining F1 scores with increasing hops, especially noticeable in the compositional subset (56% drop from 1-hop to 3-hop). Multi-hop reasoning complexity inherently poses challenges, as it requires integrating information across multiple relationships. Despite UniQGen being training-free, it maintains competitive performance in multi-hop scenarios in most cases, confirming the practicality of constraint-guided planning for complex patterns in real-world applications.

*F1 Score Distribution: Insights.* Fig. 3(b) shows a bimodal distribution, with peaks near 0 and 1, which reflects the effect of Chase & Backchase Loop: when CTable fails to capture critical constraints, early pruning yields low scores; when aligned, following Chase & Backchase refinement effectively boosts soundness and completeness, yielding high scores. Notably, Universal queries tend to outperform Minimal ones, validating our Chase-first design for completeness.

**Efficiency.** Unlike training-based systems that typically require *multi-days* fine-tuning on a Freebase-scale graph [30], UniQGen has *no* training cost; the deployment overhead is purely on runtime planning and query execution. As shown in Fig 4(a), it takes in total on average around 40 seconds per query generation, outperforming most baselines. This suggests that UniQGen is feasible even for intricate multi-hop queries. Additionally, queries from the Compositional subset require longer processing times, which is consistent with the impact of more complex queries in that set.

We use WebQSP as a public proxy for user-facing industrial queries and evaluate UniQGen under production-style constraints on our Neptune-hosted Freebase endpoint (timeout=60s, beam width/max_depth=5/5, caching=ON). To ensure label consistency with our execution setting, we re-execute the dataset-provided gold SPARQL queries on the same endpoint and use the resulting answer sets as business labels, with a successful re-execution rate of 89.69%. On this verified workload, UniQGen achieves EM/F1 of 0.78/0.79; Fig. 4(b) reports *KG execution latency* of the generated plans, where the Minimal plan achieves P50/P95 of 0.25s/0.28s and the Universal plan remains under 1s at P95. We further report per-query *resource cost*: token usage (1570.43 input/1823.29 output), per-query 2 LLM calls, and average 17 KG executions, showing a practical quality, latency, and cost trade-off. We also add a scalability analysis under query load on WebQSP (Fig. 4(c)), showing that UniQGen 's end-to-end generation latency remains stable up to at least 1500 concurrent requests, indicating predictable performance under production-style constraints.

We also report an ablation study to justify critical design choices and more real-world use case. Due to limited space, we present our findings in the Appendix.

## V. CONCLUSIONS

We have introduced UniQGen, a unified solution to generate graph queries from natural language inputs for KGQA tasks. We have highlighted its principled deployment-friendly framework based on a variant of Chase & Backchase process that optimizes the satisfaction of constraints obtained by LLMs, quantified by relative soundness and completeness measures over reference answer sets. Our experimental study has verified its ability to generate high-quality Cypher and SPARQL queries, and its scalability over large datasets and queryload.

## REFERENCES


[1] L. Nie, S. Cao, J. Shi, J. Sun, Q. Tian, L. Hou, J. Li, and J. Zhai, "Graphq ir: Unifying the semantic parsing of graph query languages with one intermediate representation," in *EMNLP*, 2022.

[2] O. Lassila, M. Schmidt, O. Hartig, B. Bebee, D. Bechberger, W. Broekema, A. Khandelwal, K. Lawrence, C. M. Lopez Enriquez, R. Sharda *et al.*, "The onegraph vision: Challenges of breaking the graph model lock-in," *Semantic Web*, vol. 14, no. 1, pp. 125–134, 2022.

[3] R. Angles, A. Hogan, O. Lassila, C. Rojas, D. Schwabe, P. Szekely, and D. Vrgoč, "Multilayer graphs: a unified data model for graph databases," in *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2022.

[4] E. Gelling, G. Fletcher, and M. Schmidt, "Bridging graph data models: Rdf, rdf-star, and property graphs as directed acyclic graphs," *arXiv preprint arXiv:2304.13097*, 2023.

[5] M. Schmidt, B. Bebee, W. Broekema, M. Elzarei, C. M. L. Enriquez, M. Neyman, F. Schmedding, A. Steigmiller, B. Thompson, G. Varkey *et al.*, "opencypher over rdf: Connecting two worlds," 2024.

[6] Y. Gu, S. Kase, M. Vanni, B. Sadler, P. Liang, X. Yan, and Y. Su, "Beyond iid: three levels of generalization for question answering on knowledge bases," in *Proceedings of the Web Conference*, 2011.

[7] W.-t. Yih, M. Richardson, C. Meek, M.-W. Chang, and J. Suh, "The value of semantic parse labeling for knowledge base question answering," in *ACL*, 2016.

[8] J. Guia, V. G. Soares, and J. Bernardino, "Graph databases: Neo4j analysis." in *ICEIS (1)*, 2017, pp. 351–356.

[9] A. Drozdov, N. Schärli, E. Akyürek, N. Scales, X. Song, X. Chen, O. Bousquet, and D. Zhou, "Compositional semantic parsing with large language models," in *ICLR*, 2022.

[10] Z. Li, S. Fan, Y. Gu, X. Li, Z. Duan, B. Dong, N. Liu, and J. Wang, "Flexkbqa: A flexible llm-powered framework for few-shot knowledge base question answering," 2024. [Online]. Available: https://arxiv.org/abs/2308.12060

[11] X. Ye, S. Yavuz, K. Hashimoto, Y. Zhou, and C. Xiong, "Rng-kbqa: Generation augmented iterative ranking for knowledge base question answering," *arXiv preprint arXiv:2109.08678*, 2021.

[12] D. Agarwal, R. Das, S. Khosla, and R. Gangadharaiah, "Bring your own kg: Self-supervised program synthesis for zero-shot kgqa," in *NAACL*, 2024.

[13] Y. Gu, X. Deng, and Y. Su, "Don't generate, discriminate: A proposal for grounding language models to real-world environments," in *ACL*, 2023.

[14] Y. Shu, Z. Yu, Y. Li, B. F. Karlsson, T. Ma, Y. Qu, and C.-Y. Lin, "Tiara: Multi-grained retrieval for robust question answering over large knowledge bases," 2022. [Online]. Available: https://arxiv.org/abs/2210.12925

[15] N. Shirvani-Mahdavi, F. Akrami, M. S. Saeef, X. Shi, and C. Li, "Comprehensive analysis of freebase and dataset creation for robust evaluation of knowledge graph link prediction models," in *International Semantic Web Conference*. Springer, 2023, pp. 113–133.

[16] M. R. A. H. Rony, U. Kumar, R. Teucher, L. Kovriguina, and J. Lehmann, "Sgpt: a generative approach for sparql query generation from natural language questions," *IEEE Access*, vol. 10, pp. 70 712–70 723, 2022.

[17] Z. Nie, R. Zhang, Z. Wang, and X. Liu, "Code-style in-context learning for knowledge-based question answering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 17, 2024, pp. 18 833–18 841.

[18] P. A. K. K. Diallo, S. Reyd, and A. Zouaq, "A comprehensive evaluation of neural sparql query generation from natural language questions," *IEEE Access*, 2024.

[19] A. Deutsch, L. Popa, and V. Tannen, "Query reformulation with constraints," *ACM SIGMOD Record*, vol. 35, no. 1, pp. 65–73, 2006.

[20] P. Sen, S. Mavadia, and A. Saffari, "Knowledge graph-augmented language models for complex question answering," in *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, 2023, pp. 1–8.

[21] D. Le, K. Zhao, M. Wang, and Y. Wu, "Graphlingo: Domain knowledge exploration by synchronizing knowledge graphs and large language models," in *ICDE*, 2024.

[22] B. Jin, C. Xie, J. Zhang, K. K. Roy, Y. Zhang, Z. Li, R. Li, X. Tang, S. Wang, Y. Meng *et al.*, "Graph chain-of-thought: Augmenting large language models by reasoning on graphs," in *ACL*, 2024.

[23] Y. Feng, S. Papicchio, and S. Rahman, "Cypherbench: Towards precise retrieval over full-scale modern knowledge graphs in the llm era," *arXiv preprint arXiv:2412.18702*, 2024.

[24] I. L. Oliveira, R. Fileto, R. Speck, L. P. Garcia, D. Moussallem, and J. Lehmann, "Towards holistic entity linking: Survey and directions," *Information Systems*, vol. 95, p. 101624, 2021.

[25] G. Bekoulis, C. Papagiannopoulou, and N. Deligiannis, "A review on fact extraction and verification," *CSUR*, vol. 55, no. 1, pp. 1–35, 2021.

[26] M. Wang, N. Jedema, R. Pandey, R. Krishnan, J. Lehmann, and Y. Wu, "Graph query generation with constraint-guided large language agents," 2025. [Online]. Available: https://wangmengying.me/papers/uniqgen.pdf

[27] W. Zheng, J. X. Yu, L. Zou, and H. Cheng, "Question answering over knowledge graphs: question understanding via template decomposition," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1373–1386, 2018.

[28] Y. Su, H. Sun, B. Sadler, M. Srivatsa, I. Gür, Z. Yan, and X. Yan, "On generating characteristic-rich question sets for QA evaluation," in *EMNLP*, 2016.

[29] Y. Gu and Y. Su, "Arcaneqa: Dynamic program induction and contextualized encoding for knowledge base question answering," in *Proceedings of the 29th International Conference on Computational Linguistics*, 2022, pp. 1718–1731.

[30] Pangu, "Pangu github issue: Compute resources." 2025. [Online]. Available: https://github.com/dki-lab/Pangu/issues/6

[31] R. Das, M. Zaheer, D. Thai, A. Godbole, E. Perez, J.-Y. Lee, L. Tan, L. Polymenakos, and A. Mccallum, "Case-based reasoning for natural language queries over knowledge bases," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 9594–9611.

[32] Y. Lan and J. Jiang, "Query graph generation for answering multi-hop complex questions from knowledge bases." Association for Computational Linguistics, 2020.

[33] X. Ye, S. Yavuz, K. Hashimoto, Y. Zhou, and C. Xiong, "Rng-kbqa: Generation augmented iterative ranking for knowledge base question answering," in *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: Long papers)*, 2022, pp. 6032–6043.

[34] G. He, Y. Lan, J. Jiang, W. X. Zhao, and J.-R. Wen, "Improving multi-hop knowledge base question answering by learning intermediate supervision signals," in *Proceedings of the 14th ACM international conference on web search and data mining*, 2021, pp. 553–561.

[35] J. Jiang, K. Zhou, X. Zhao, and J.-R. Wen, "Unikgqa: Unified retrieval and reasoning for solving multi-hop question answering over knowledge graph," in *The Eleventh International Conference on Learning Representations*.

[36] J. Jiang, K. Zhou, Z. Dong, K. Ye, W. X. Zhao, and J.-R. Wen, "Structgpt: A general framework for large language model to reason over structured data," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 9237–9251.

[37] L. LUO, Y.-F. Li, G. Haffari, and S. Pan, "Reasoning on graphs: Faithful and interpretable large language model reasoning," in *The Twelfth International Conference on Learning Representations*.

[38] X. He, Y. Tian, Y. Sun, N. Chawla, T. Laurent, Y. LeCun, X. Bresson, and B. Hooi, "G-retriever: Retrieval-augmented generation for textual graph understanding and question answering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 132 876–132 907, 2024.

## Appendix I: Glossary of Key Notations

* The examples related to the Olympic Games are based on the context provided in Example 2.

$Q_n$ – Natural-language query/question (input).

$Q$ – A structured Cypher query generated from a natural language query.
*Example:* `MATCH (c:City)-[:hosted]->(e:OlympicGames)`

$G$ – Knowledge graph containing entities (nodes) and relations (edges).

$Q(G)$ – Answer set obtained by executing query $Q$ on $KG$.

$V'$ – Entity-linked node set extracted from $Q_n$, $V' \subseteq V$;

$G'$ – Query-induced context subgraph for constraint extraction, $G' = \bigcup_{v \in V'} G'(v)$;

$A$ – The reference answer set provided by an oracle LLM in response to a natural language query.
*Example:* For "Olympics cities in the USA," $A$="Atlanta","St.Louis","LosAngeles".

$G(A)$ – All $KG$ entities corresponding directly to answers in the reference set $A$.

$\mathcal{C}$ (CTable) – A structured representation (Constraint Table) of all constraints, capturing triple patterns extracted from $KG$ to support accurate query generation.
*Example:* Entries: *(City, hosted, OlympicGames)*, *(OlympicGames, year, ¿1896)*.

$c \in \mathcal{C}$ – An individual constraint within the CTable $\mathcal{C}$, defined as a triple pattern or condition.
*Example: (City, hosted, OlympicGames)* (single constraint).

$u$ – Constraint uncertainty score (0–1), quantifying the specificity of each constraint. Lower $u$ means higher specificity.
*Example:* Constraint *(year¿1900)* have a higher uncertainty score than *(year¿2000)*, indicating broader applicability.

$p$ – Performance score assigned by the Evaluator to a query, measuring how well its result matches the oracle answer set $A$.
*Example:* If query results fully match Oracle answers $A$, then $p \approx 1.0$

$s$ – Overall score for candidate queries used to rank queries during beam search, combining constraint uncertainty $u$ and query performance $p$.

$Q_U$ **(Universal Query)** – Query derived in the Chase phase, ensuring completeness by initially including all constraints from $\mathcal{C}$.
*Example:* A universal query retrieves all cities hosting Olympics (both Summer and Winter), maximizing completeness.

$Q_M$ **(Minimal Query)** – Query generated in the Backchase phase by minimizing constraints from $Q_U$, ensuring soundness.
*Example:* A minimal query includes only essential constraints, e.g., filters specifically to "Summer Olympics.".

$b$ **(Beam Width)** – Number of candidate queries retained at each iteration (beam step) during Chase & Backchase processes.
*Example:* With $b = 5$, only the top 5 candidates by score $s$ proceed to the next iteration.

## Appendix II: Algorithms

### A. CTable *Construction*

**Algorithm 2** CTable Construction

**Input:** Natural language query $Q_n$, Knowledge Graph $G$, hop size $k$, Match Cap $\tau$, Agent $a_1$
**Output:** Scored Constraint Table $\mathcal{C}_0$

$E \leftarrow$ EntityLinking($Q_n$) ▷ Link entities in the NL query
$G' \leftarrow$ Subgraph($G$, $E$, $k$) ▷ Extract $k$-hop subgraph
**Initialize** CTable $\mathcal{C} := \emptyset$
**for** each triple $t = \langle s, p, o \rangle$ in $G'$ **do**
  **if** $t$ is relevant to the $Q_n$ **then**
    $\mathcal{C} := \mathcal{C} \cup \{t\}$
Extend $\mathcal{C}$ by using $a_1$ to inject additional constraints (e.g., filters, type conditions)
**for** each constraint $c_i \in \mathcal{C}$ **do**
  $n_i \leftarrow$ MatchCount($G$, $c_i$) ▷ Count supports in $KG$
**Let** $m := \min(\max_i n_i, \tau)$
**for** each $c_i \in \mathcal{C}$ **do**
  $u_i := \frac{n_i}{m}$ ▷ Compute uncertainty score
$\mathcal{C}_0 := \{(c_i, u_i) \mid u_i > 0\}$ ▷ Prune constraints with zero matches
**return** $\mathcal{C}_0$

Algorithm 2 describes the procedure used by Agent $a_1$ to construct a scored constraint table (CTable) from a natural language query $Q_n$. Initially, EntityLinking function identifies entities $E$ explicitly mentioned in $Q_n$. Then, a relevant subgraph $G'$ is

extracted from the knowledge graph $G$ by retrieving nodes and edges within $k$-hop neighborhoods of these linked entities. Relevant triples within $G'$ that align with the query intent are collected to form the initial CTable, which Agent $a_1$ subsequently enriches by injecting inferred constraints, such as additional type restrictions and filtering conditions. Each constraint $c_i$ in the CTable is evaluated by counting its support $n_i$, the number of matching triples in $G$. An uncertainty score $u_i$ is then computed as the normalized support count relative to the maximum support count (capped by $\tau$). Constraints without any matches are pruned, resulting in the final scored constraint table $\mathcal{C}_0$.

### *B. Query Backchase*

We present more details of Query Backchase, as shown in Algorithm 3. Similar as its Chase counterpart, Backchase works with (1) a query Generator $\gamma$, to explore sub-queries of the input universal query $Q_U$ following a bottom-up manner (by"merging" from small, single-edge patterns into larger sub-queries); and (2) an Evaluator $\epsilon$ that consults an LLM to identify answers by evaluating an input sub-query $Q$ generated from $\gamma$. The algorithm follows a beam search process, starting with the universal query $Q_U$, and initializes a set of sub-queries by decomposing $Q_U$ into triple patterns $c$ (line 2). It coordinates a the backchase then by generating and verifying a re-combination of these sub-queries into larger counterparts (lines 4-8), by "adding back" the constraints as part of the subqueries from the extracted constraint table CTable. The Evaluator (line 8), during the beam search based exploration of sub-queries, ensures the hard constraints by verifying if the current subquery $Q_n$ remains to be LLM-relative completeness, as a hard constraint. As is, the algorithm ensures an invariant that it only explores sub-queries that are guaranteed to be consistent and LLM-relative complete.

**Consistency Guarantee**. We clarify that the above process ensures a consistency guarantee for the output query $Q$ and its associated CTable $\mathcal{C}$. This is not a strong consistency guarantee which ensures $Q$ is always consistent with the input constraint table $\mathcal{C}_0$, yet a pragmatic guarantee to have reasonable query output, especially when $\mathcal{C}_0$ may be ambiguous or inconsistent – in the latter case, no query with answers can simultaneously satisfy all the constraints at the same time.

We consider a pair $(Q, Q.\mathcal{C})$, initialized as $(Q_U, Q_U.\mathcal{C}_0)$. It suffices to show that the process has the invariant that for any output $(\mathcal{C}, Q)$ with updates that change $\mathcal{C}_0$ to $\mathcal{C}$ and $Q_U$ to $Q$, $Q$ has answer that satisfy all the constraints in $Q.\mathcal{C}$. This can be shown by an inductive analysis, for a reasoning path as a sequence of updates in Chase and BackChase to the pair $(Q, Q.\mathcal{C})$. (1) The Query Chase Phase initializes CTable $\mathcal{C}_0$ with all constraints and a universal query $Q_U$ that contain all constraints in $\mathcal{C}_0$. If it outputs $(Q_U, Q_U.\mathcal{C}_0)$ without any Chase or BackChase step, then either $\mathcal{C}_0$ is $\emptyset$ (as $B$=, line 3 of Algorithm 1), or $Q_U$ contains a set of query nodes without any edges *i.e.,* no triple constraints can be enforced with an answer that satisfies all $\mathcal{C}_0$ – a case that $\mathcal{C}_0$ is inconsistent. In either case, $Q_U$ is trivially consistent with $\mathcal{C}_0$. (2) Assume at step $i$ of the search, Chase derives $(Q, Q.\mathcal{C})$ where $Q$ is consistent with $Q.\mathcal{C}$. At any spawning step $i+1$ in QChase, $Q$ is updated to $Q'$ by removing a triple pattern $c$ from $\mathcal{C}$. Consider the pair $(Q', Q'.\mathcal{C})$, we have $Q(G) \subseteq Q'(G)$, for a fixed set of node variables. There are two cases: (a) the new answers in $Q'(G) \setminus Q(G)$ also satisfy $Q'\mathcal{C}$. In this case, $(Q', Q'.\mathcal{C})$ ensures consistency guarantee. (b) at least one answer in $Q'(G) \setminus Q(G)$ does not satisfy $Q'.\mathcal{C}$ due to the removal of the triple pattern $c$. Then there must exist a step in QBackChase, posed on the subquery $Q_i$ that either is itself $c$, or a subquery $Q''$ of $Q'$ such that $Q''$ is derived by adding back $c$, with verified answer $Q''(G)$ that satisfy $Q''.\mathcal{C}''$. In both cases, it eventually outputs $(Q'', Q''.\mathcal{C}'')$ that satisfy the consistency guarantee. This proves the consistency guarantee.

**Relative Completeness Guarantee (QChase)**. We provide a similar inductive proof to show the completeness guarantee ensured by QChase phase, following our construction in the consistency analysis. (1) Given the reference set $A$, if QChase output $(Q_U, \emptyset)$, *i.e.,* $\mathcal{C}$ has no triple pattern to be removed, and $Q_U$ is a set of query nodes without edge constraint – hence include all the nodes in $G$ with a simple type match only, hence $A \subseteq Q_U(G)$. (2) Assume at step $i$, $(Q, Q.\mathcal{C})$ ensures relative completeness, *i.e.,* $A \subseteq Q(G)$. Then, at any follow-up step that spawns a new query from $Q$ and yields $Q'$ by removing a triple constraint, we have $A \subseteq Q(G) \subseteq Q'(G)$ because the removal of a triple constraint does not reduce the matched answers. Hence the relative completeness is ensured at any step of QChase. Note that QChase alone does not necessarily satisfy relative soundness, as relaxing constraints may introduce new answers not in $A$. This is coped with QBackchase.

**Relative Soundness Guarantee (QBackChase)**. Following our construction in the consistency and relative completeness analysis, consider the initialization of QBackChase that starts with a set of pairs $(Q_i, C)$, where each $Q_i$ is a single edge query of the input query $Q_U$ from QChase. As $(Q_U, Q_U.\mathcal{C})$ ensures relative completeness, given the proof of relative completeness guarantee of QChase, we have $A \subseteq Q_U(G)$. Clearly for each $Q_i$ as a subquery of $Q_U$, $A \subseteq Q_U(G)$. The QBackChase proceeds by combining subqueries together and outputs once it verifies that $Q' = \bigcup Q_j$, where $Q'(G) \subseteq A \subseteq Q_j(G)$, given the corresponding constraint $Q'.C'$ as the *conjunction*, *i.e.,* $Q'.\mathcal{C} = \bigwedge Q_j.\mathcal{C}_j$. Hence, for every output $(Q', Q'.\mathcal{C}')$, the relative soundness holds. This completes the proof of relative soundness guarantee.

**Minimality guarantee**. We advocate a desirable property as follows. Given a NL query $Q_0$ with a reference answer $A$ from an LLM, and a set of constraints encoded in a CTable $\mathcal{C}$, a Cypher query $Q$ is a *minimal complete rewriting* of $Q_0$, if (1) $Q$ is LLM-relevant complete that is consistent with $\mathcal{C}$, and (2) for any sub-query obtained by removing a constraint from $Q_0$ in

**Algorithm 3** Query Backchase (QBackchase)

**Input:** Universal Query $Q_U$; NL Query $Q_n$; Query Generator $\gamma$; Evaluator $\varepsilon$; Beam width $b$; Threshold $\tau$; Graph $G$;
**Output:** Minimal Query $Q_M$.

**set** $B := \varnothing$, $L := \varnothing$;
**for each** $c_i \in Q_U.\mathcal{C}$ **do**
    $\mathcal{C}_i := \{c_i\}$, $B$.append($(\mathcal{C}_i, 0)$);
**while** $B \neq \emptyset$ **do**
    **for** $(\mathcal{C}, s) \in B$ **do**
        $Q :=$ Gen$(\gamma, \mathcal{C})$, $r := Q(G)$, $p := 0$;
        $L$.append($(\mathcal{C}, Q, r, p, s)$);
    $L :=$ Eva($\varepsilon$, $L$, $Q_n$); ▷ calcu. $l.p$, $\forall l \in L$
    $l_M := \arg\max_{l \in L} l.p$, $Q_M := l_M.Q$;
    **if** $l_M.p \geq \tau$ **then** break;
    $B := \{(l.\mathcal{C} \cup \{c\}, s(c, l)) \mid l \in L, c \in (Q_U.\mathcal{C} \setminus l.\mathcal{C})\}$
    **if** $|B| \leq b$ **then** continue;
    $B := \{(\mathcal{C}, s) \in B \mid$ top $b$ by $s\}$, $L := \varnothing$;
**return** $Q_M$

$\mathcal{C}$, $\mathcal{C}$ is no longer LLM-relevant complete. The minimality property ensures that the generated query contains a set of rich and accurate search conditions (constraints) that can accurately express the original NL queries, as close as possible.

We verify that QBackchase ensures to produce minimal complete rewriting of an input NL query $Q_0$. To see this, observe the following. (1) The sub-queries, as a conjunct of single constraint patterns (single-edge queries), are either discarded in the beam search due to that they return incomplete or empty answers, or are kept to be more accurate (selective) by enriching with more relevant constraints, which remains consistent, and LLM-relevant complete. (2) For any children spawned from any returned query $Q$, QBackchase ensures to discard them due to the violation of LLM-relevant complete.

For time cost, Backchase incurs the same cost as its Chase counterpart, in $O(L(B + |\mathcal{C}| \log |\mathcal{C}|))$ time, where $B$ is the beam size, and $L$ refers to the number of levels (the depth) the beam search goes. We remark that $|\mathcal{C}|$ is often a small constant, as it only contains relevant constraints derived from NL query $Q_0$ and relevant triples in KG, at query-time; hence in practice the time cost is comparable to $O(BL)$. Note that QBackchase is also training-free, hence UniQGen does not incur additioinal learning overhead.

### *C. Design intuition (line 10 in Alg. 1)*

*– Why are new candidates spawned by removing one constraint at a time in* QChase*?*

Alg. 1 (QChase) performs a top-down beam search over subsets of constraints in the scored CTable $C_0$ (normalized to the monotone core). When a candidate query $Q$ fails the completeness check against the oracle reference set $A$, QChase relaxes $Q$ by dropping constraints. Under the monotone-core normalization, removing a constraint can only enlarge the retrieved answer set, and therefore is a principled way to recover missing reference answers and improve relative completeness. The trade-off is that fewer constraints may also introduce more spurious answers and larger intermediate results.

We drop only *one* constraint at a time to enumerate *minimal* relaxations, which has three benefits: (i) it preserves selectivity as much as possible by avoiding overly aggressive relaxation, (ii) it helps isolate which constraint is responsible for blocking coverage, and (iii) it keeps the branching factor controllable under a fixed beam budget. Descendants are prioritized by Eq. 1 and only the top-$b$ descendants are expanded to the next beam step. This favors relaxing less reliable and more permissive constraints via uncertainty $u$, while also accounting for the parent query's quality score.

For example, in the demonstration in Appendix III, the initial candidate $Q_0$ includes two season constraints (e.g., `e.type='Winter'` and `e.type='Summer'`) and is over-restrictive and fails the completeness check against $A$. Dropping the season constraint yields a season-agnostic query $Q_U$, which no longer relies on the uncertain enum grounding and therefore recovers coverage of $A$. This illustrates why single-constraint relaxation is effective: it makes progress toward coverage with the smallest possible relaxation at each step, while avoiding unnecessary over-relaxation (and cost) that would occur if multiple constraints were dropped simultaneously. Finally, QBackchase can re-introduce a *KG-grounded* season constraint to remove spurious answers while preserving coverage, producing the minimal query $Q_M$.

## Appendix III: Demonstrations

**Example 2:** Considering the NL query *"Which cities in the USA have hosted the Olympics in February?"*, With

```
A = {Lake Placid; Salt Lake City; Squaw Valley}
```

, QChase produces $Q_U$ retrieving both Summer/Winter hosts, ensuring $G(A) \subseteq Q_U(G)$:

```
MATCH (c:City)-[:hosted]->(e:OlympicGames)
WHERE c.country = 'USA' AND e.year > 1896
AND (e.type = 'Summer' OR e.type ='Winter')
RETURN c.name
```

QBackchase then adds back only the necessary constraints. As February is a winter month, and the first Olympics hosted in the USA was in 1904, the minimized $Q_M$ might be:

```
MATCH (c:City)-[:hosted]->(e:OlympicGames)
WHERE e.type = 'Winter' AND c.country = 'USA'
RETURN c.name
```

which excludes summer-only hosts like `St.Louis, Los Angeles, and Atlanta`, ensuring soundness as $Q_M(G) \subseteq G(A)$, while retaining only the minimal necessary constraints.

As an in-depth illustration of the above example, we provide a step-by-step illustration of the UniQGen pipeline, as a real-world knowledge search use case.

**M1: Query-centric Factual Extraction:**

**Entity Linking**: Initially, entities explicitly mentioned in the query are identified and linked to corresponding nodes in the knowledge graph ($KG$). For our example query, the identified entities are:

```
{OlympicGames, USA}
```

**Subgraph Extraction**: Using the linked entities, a relevant subgraph $G'$ is extracted from the main knowledge graph $G$. Typically, neighbors within $k$-hop (usually $k = 1$ or $k = 2$) from linked entities are included. For simplicity, we consider 1-hop neighbors, resulting in factual constraints (triples) for the initial CTable:

| | | |
|---|---|---|
| ?city | hosted | ?e:OlympicGames |
| ?city | country | USA |
| ?city | name | ?city_name |
| ?e:OlympicGames | type | ?season |
| ?e:OlympicGames | year | ?host_year |

Here, variables act as placeholders for entities or attributes that will become nodes or properties in the generated Cypher query:

- `?city` represents nodes of type *City*.
- `?city_name` denotes the name attribute of a city.
- `?season` indicates the type of Olympic Games (e.g., Summer or Winter).
- `?host_year` represents the year in which the Olympics were hosted.

**M2: CTable Construction** (Agent $a_1$)**:**

Agent $a_1$ systematically extracts relevant triples from the subgraph and enriches these factual constraints with implicit conditions derived from its internal knowledge base. Specifically, $a_1$ extends the constraints in CTable based on contextual relevance to the query:

| ID | Constraint in CTable |
|---|---|
| $c_1$ | `(c:City)-[:hosted]->(e:OlympicGames)` |
| $c_2$ | `c.country = 'USA'` |
| $c_3$ | `e.year > 1896` |
| $c_4$ | `e.type = 'Winter'` |
| $c_5$ | `e.type = 'Summer'` |

Here, additional constraints include:

- `e.year > 1896` ensures relevant modern Olympic events.
- `e.type within [Summer, Winter]` are temporal constraints, restricting the type of Olympic events to major recognized categories. For this query, the temporal cue *"in February"* should ideally map to a winter-specific constraint, but in practice, the LLM may *fail to ground* the correct season predicate/value under a given KG schema.

We compute an uncertainty score $u(c)$ based on capped match counts in $G$ (see Sec. III-B). Intuitively, constraints that are (i) very broad or (ii) unreliable to ground tend to be deprioritized during search.

**M3: Chase & Backchase** (Agent $a_2$)**:**

Agent $a_2$ first retrieves the reference answer set $A$ from an LLM oracle, which provides a reference answer set $A$:

```
A = {Lake Placid; Salt Lake City; Squaw Valley}
```

QChase **(Universal query generation).** QChase aims to produce a universal query $Q_U$ such that $KG(A) \subseteq Q_U(G)$. It starts from an initially tight constraint set and iteratively relaxes it by dropping *one* constraint at a time (Alg. 1, ln. 10), keeping only the top-$b$ candidates per iteration.

Starting from $\mathcal{C}_0 = \{c_1, c_2, c_3, c_4, c_5\}$, the generated query enforces both seasonal filters (`Winter` and `Summer`), which is over-restrictive and may return empty results:

```
Q_0:
MATCH (c:City)-[:hosted]->(e:OlympicGames)
WHERE c.country='USA' AND e.year>1896
  AND e.type='Winter' AND e.type='Summer'
RETURN c.name
```

$Q_0$ fails the completeness check against $A$. Assuming beam width = 1, then **`e.type='Winter'`** will be dropped in round 1 because it is more specific, given there are fewer winter Olympics, while $Q_1$ still fails the completeness check against $A$. **`e.type='Summer'`** dropped in round 2, then the $Q_2$ is independent of sessions, which can be written as:

```
Q_U:
MATCH (c:City)-[:hosted]->(e:OlympicGames)
WHERE c.country = 'USA'
AND e.year > 1896
RETURN c.name
```

This query retrieves all US cities hosting the Olympics after 1896, covering both Summer and Winter games, thus ensuring $KG(A) \subseteq Q_U(KG)$:

```
{St.Louis;Los Angeles;Lake Placid;Atlanta;Salt Lake City;Squaw Valley}
```

QBackchase **(Minimal Query Refinement)**: $a_2$ then construct a minimal query $Q_M$, ensuring soundness and precision while eliminating irrelevant results:

```
MATCH (c:City)-[:hosted]->(e:OlympicGames)
WHERE e.type = 'Winter'
AND c.country = 'USA'
RETURN c.name
```

This minimal query specifically targets cities hosting Winter Olympics (as it queried Feb sessions), thereby excluding irrelevant summer-hosting cities like `St.Louis, Los Angeles, and Atlanta`, satisfying the soundness criterion $Q_M(KG) \subseteq KG(A)$.

**Error propagation under an incorrect oracle reference set.** UniQGen 's completeness/soundness guarantees are *relative* to the oracle-provided reference set $A$. To illustrate how Oracle errors propagate, consider the same NL query in this demonstration. Suppose the correct reference answers are winter hosts $A = \{\text{Lake Placid}, \text{Salt Lake City}, \text{Squaw Valley}\}$, but the oracle mistakenly adds a summer host (false positive), e.g., $A^{\text{err}} = A \cup \{\text{Los Angeles}\}$. In this case, QChase must construct a universal query $Q_U$ whose answers cover $G(A^{\text{err}})$, which forces $Q_U$ to relax or drop the winter-only constraint (otherwise it cannot cover Los Angeles). Then QBackchase tightens the query while preserving coverage of $A^{\text{err}}$; as a result, the returned minimal query may retain a summer-permitting season filter (or become season-agnostic), producing answers that deviate from the intended "February/Winter" semantics. This demonstrates that if $A$ is wrong, UniQGen may overfit to $A$ even though its runtime checks remain valid w.r.t. $A$; we therefore recommend obtaining $A$ via higher-trust sources such as gold queries, and possible optimization like multi-prompt consensus and external fact checking.

## Appendix IV: Prompt Templates

### A. CTable Constructor

I am generating Cypher queries for natural language (NL) question based on {KB}. The first step is to generate a constraint table for a given natural language question and extract entities with their k-hop relations. The table should include basic triples in the format (subject, predicate, object) or (instance, filter, value), using {KB} entity IDs where applicable. Please provide as many details as possible, listing all possible triples we might use. Include any relevant relationships, properties, and filters that could be useful in constructing the query. Alao feel free to add any relevant triples or constraints based on your knowledge of KB that might improve the generated Cypher query.

Requirements:

1. Please only output the table in markdown format, no other context.

2. Double-check that all essential properties for connecting entities are included; don't omit them.

...

Input:

{NL Query},

{Extracted Entities},

{Factual Constraints}

Output:

{constraint table}

### *B.* Query Generator

Given a natural language query, extracted entities, and a list of constraints, generate a Cypher query that uses all provided constraints without adding additional ones. The query should be optimized for performance testing with the specified constraint set. Add operators or query modifiers as needed to accurately answer the NL query.

**Please be very strict with the syntax rules and requirements mentioned below!!!**

Syntax Rules:

1. For node IDs, use the format: (‘ id‘: ”¡url¿”)

2. For constrains/relationships(predicates that connect two nodes) use the format: [:‘¡url¿‘] 3. For properties/attributes, use the format m.‘¡url¿‘

...

Requirements:

1. Incorporates all given constraints exactly as provided

2. Uses the known entities appropriately

3. Does not introduce any additional constraints

4. Includes necessary operators or query modifiers to answer the NL query

...

Input:

{NL Query},

{Extracted Entities},

{Constraints}

Output:

{Cypher query}

### *C.* Evaluator

You are an expert system for evaluating answers to knowledge graph queries. Your task is to score a list of potential answers to a given natural language query. Also, output a reference answer set, and then we can assert the quality with self-defined hard constraints.

Please follow these guidelines:

1. Consider the relevance and accuracy of each answer in relation to the query.

2. Assign a score from 0 to 20 for each answer, where:

* 0 means completely irrelevant or incorrect

* 20 means highly relevant and likely correct

* Use the full range of scores to differentiate between answers

3. For empty or null answers, assign a score of 0.

4. If multiple answers are identical, give them the same score.

5. Consider both the content and the format of the answer.

...

Input format:

{Natural language query}

{List of answers}

Output:

{Reference Answer Set}

{List of scores}

### *D. Prompt variance of the oracle reference set $A$*

UniQGen uses an LLM oracle to obtain a reference answer set $A$ for an NL query, which guides the Chase/Backchase loop.

**P0: Minimal answer-only prompt (fast, higher variance).**

> **Prompt:** Given the question: *"{NL query}"*, output the answer entities as a semicolon-separated list. Do not provide any explanation.

Note: This prompt is lightweight and cost-efficient, but often exhibits higher variance and may include both false positives and false negatives.

**P1: Structured, precision-oriented prompt (safer grounding).**

> **Prompt:** Given the question: *"{NL query}"*, output a semicolon-separated list of answer entities. **Rules:** (i) only output entities you are confident exist in the KG; (ii) if uncertain, output `UNKNOWN`; (iii) do not include explanations or extra entities.

Note: This strategy aims to reduce hallucinated entities (false positives), which helps avoid forcing QChase to relax constraints to cover incorrect answers.

**P2: Self-consistency oracle (variance reduction via sampling).** We run the structured prompt (P1) $K$ times with different random seeds (e.g., $K = 3$) to obtain $\{A^{(1)}, \ldots, A^{(K)}\}$, and then aggregate:

- **Majority vote:** include an entity if it appears in at least $\lceil K/2 \rceil$ runs (balanced precision/recall).
- **Intersection:** include only entities appearing in all runs (high precision, potentially lower recall).

Note: Self-consistency often stabilizes $A$ and reduces prompt sensitivity, at the cost of additional oracle calls.

**P3: MoE-style aggregation (recall expert + precision expert + verifier).** We implement a lightweight mixture-of-experts (MoE) style oracle using two complementary prompts and an optional verifier:

- **Recall expert ($E_R$):** "List all plausible answer entities, even if you are not fully sure."
- **Precision expert ($E_P$):** "List only high-confidence answer entities; if unsure, output `UNKNOWN`."
- **Aggregator:** start from $A_\cap = A_R \cap A_P$; if $A_\cap$ is empty, fall back to $A_\cup = A_R \cup A_P$ and optionally apply a verifier prompt:

  > **Verifier:** Given the question and candidate answers, remove any answer you are not confident is correct. Output the remaining answers only (semicolon-separated).

Note: This MoE-style strategy explicitly trades off recall and precision, mitigating both hallucinated additions and excessive omissions in $A$, while incurring higher token consumption and end-to-end latency.

## Appendix V: Complementary Experimental Study

**Ablation Studies.** We further analyze critical design choices:

*Prompt-only is insufficient.* The Prompt-Only baseline underperforms on all datasets (e.g., GrailQA-overall F1=0.306; GraphQ F1=0.223), confirming that prompt engineering alone may not be sufficient to achieve desired performance, while UniQGen benefits from both prompts and LLM-guided constraint-based query generation process.

*Evaluator strength matters.* To explore the effect of LLM Oracle, and potential upper bounds of our evaluation process, we conducted experiments replacing the LLM-generated reference set used in Chase & Backchase with groundtruth on 100 random GraphQ questions. The direct use of ground truth notably increased F1 from 0.804 to 0.835, showing that a stronger oracle improves UniQGen 's acceptance decisions and tightens guarantees without changing the logic of the planner. This highlights UniQGen's ability to offer strong and reliable quality guarantees when guided by a trustworthy evaluator. Tests were executed using Claude 3.5:

| Setting | P | R | F1 | EM |
|---|---|---|---|---|
| w. LLM-based Evaluator | 0.813 | 0.857 | 0.804 | 0.740 |
| w. Groundtruth Reference Set | 0.833 | 0.898 | 0.835 | 0.800 |

On the other hand, we evaluate oracle stability by measuring model–ground-truth agreement on a random sample of 200 questions. The oracle achieves an average overlap rate above 80%, indicating that, given the current capability of LLMs, using an LLM oracle provides a sufficiently reliable signal to support our framework in practice.

**GraphQ IR v.s. UniQGen.** GraphQ IR [1] that outputs a unified IR and compiles it to multiple query languages, which is different from UniQGen 's rendering-based, runtime-validated approach. We evaluate GraphQ IR on the overlapping benchmark it supports (GrailQA), and report the accuracy in the following table:

| Methods | I.I.D | Compositional | Zero-shot | Overall |
|---|---|---|---|---|
| UniQGen | 0.989 | 0.74 | 0.889 | 0.871 |
| GraphQ IR | 0.874 | 0.495 | 0.096 | 0.369 |

| Methods | Category | Train | Query | WebQSP Performance |
|---|---|---|---|---|
| CBR-KBQA (CBR) [31] | Case-based reasoning (non-parametric) | Y | Y | F1=72.8, EM=70.0 |
| QGG [32] | Semantic parsing (step-wise) | Y | Y | F1=74.0 |
| RNG-KBQA [33] | Semantic parsing (seq2seq) | Y | Y | F1=75.6, EM=71.1 |
| NSM [34] | Neural symbolic / GNN | Y | N | F1=74.3 |
| UniKGQA [35] | GNN-based retrieval | Y | N | F1=72.2, Hit@1=77.2 |
| StructGPT [36] | LLM+KG (structured prompting) | N | N | Hit@1=72.6 |
| RoG-7B [37] | KG+LLM (7B, fine-tuned) | Y | N | F1=70.8, Hit@1=85.7 |
| G-Retriever [38] | GNN+LLM | Y | N | Hit@1=73.79 |

TABLE III: Representative KGQA methods and reported performance on WebQSP. "Train" indicates task-specific training; "Query" indicates whether the method outputs an executable query (e.g., SPARQL/logical form) rather than only answers. Reported metrics vary across papers (F1/EM vs Hit@1).

It is clear that UniQGen outperforms GraphQ IR across all splits of the GrailQA benchmark. For GraphQ and WebQSP, GraphQ IR does not include an official setting in its release. Adapting it to these benchmarks would require creating IR annotations (or an equivalent supervision signal) and training a new parser based on their self-defined IR formalism. Therefore, we do not present GraphQ IR results for GraphQ/WebQSP.

**Case Studies.** Benchmarks like WebQSP often reward literal truth-matched retrieval, overlooking user intent. For example, WebQTest-1572 asks: "*what should I do **today** in San Francisco?*" Its "golden" SPARQL simply lists all tourist attractions linked to the city. When we tested this on a Wednesday, Agent $a_1$ automatically added constraints that removed SFMOMA and the Asian Art Museum - both closed on Wednesdays; while the reference set generated by Agent $a_2$ filtered out the mis-typed business Travefy, a travel-software company mistakenly included in the KG. If a customer asked Amazon Alexa this question on a Wednesday, they would expect precisely such context-aware, operational answers. This demonstrates how UniQGen 's LLM-assisted, constraint-based design moves beyond static benchmark evaluation toward deployable, intent-aligned query generation.

**Representative KGQA methods.** Table III summarizes representative KGQA approaches on WebQSP spanning non-parametric case-based reasoning (CBR-KBQA), classical semantic parsing (QGG, RNG-KBQA), neural-symbolic / GNN-based reasoning (NSM, UniKGQA), and LLM-era KGQA that couples LLMs with graph grounding (StructGPT, RoG-7B, and G-Retriever). We highlight two practical dimensions that often lead to different deployment trade-offs: **Train** indicates whether a method relies on task-specific training (or a training-backed memory, as in CBR-KBQA), and **Query** indicates whether the method outputs an executable structured query (e.g., SPARQL or a logical form) instead of producing answers directly. Semantic parsing methods (QGG, RNG-KBQA) and CBR-KBQA explicitly generate or retrieve executable queries and report strong F1/EM, but typically depend on supervised training and/or candidate generation and schema alignment. In contrast, GNN/neural-symbolic methods (NSM, UniKGQA) and many LLM-era approaches (StructGPT, RoG-7B, G-Retriever) often answer via traversal/retrieval/reasoning over subgraphs rather than emitting a final query, leading to different robustness and efficiency characteristics.