ICDE 2026
MONTRÉAL, CANADA

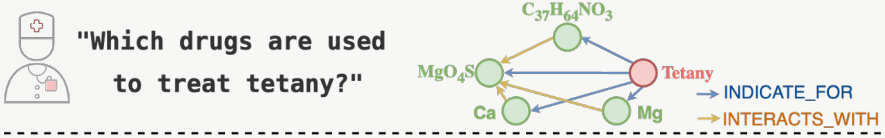
Graph Query Generation with Constraint-guided Large Language Agents

Mengying Wang[1*], Nicolaas 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]TU Dresden

Work done while at Amazon AGI

Support Knowledge Graph QA w. Structured Graph Queries



Rule-based → None

```
MATCH (n0:Drug)-[:TREAT_ON]->(n1 {name: "Tetany"})
RETURN n0
```

→ *Extracts surface relations directly from the question. Looks for a literal TREAT_ON edge.*

Learning-based → $\{C_{37}H_{64}NO_3, Ca, Mg, MgO_4S\}$

```
MATCH (n0:Drug)-[:INDICATE_FOR]->(n1 {name: "Tetany"})
RETURN n0
```

→ *Schema-aware fine-tuning finds INDICATE_FOR, but returns unapproved drugs.*

LLM-Assisted, Constraints-based → $\{C_{37}H_{64}NO_3, Mg, MgO_4S\}$

```
MATCH (n0:Drug)-[:INDICATE_FOR]->(n1 {name: "Tetany"})
WHERE n0.Approved = True
RETURN n0
```

→ *Captures implicit intent (Approved=True) and respects the ontology.*

The goal is not only syntactic validity ——

It is an executable query that respects both KG topology and pragmatic intent.

NL to Structured Graph Queries: SOTA & Challenges

Hallucination (LLM-based)

Generated queries don't match the KG's schema or topology

Missing Constraints (Rule-based)

Implicit user intent lost
(e.g., "Approved=True" filter dropped)

High Maintenance

Schema-specific fine-tuning is brittle and costly on large and evolving graphs

Language Lock-in

A separate model per query language is expensive

***UniQGen**: a training-free, schemaless generator that preserves user intent and renders executable queries for multiple graph query classes.*

Notations & Problem Statement

Key Concepts

`Knowledge Graph G = (V, E)`

Entities V, typed edges E as triples ⟨s, p, o⟩

`Constraint Table C (CTable)`

Typed triple patterns and value/intent constraints, each with uncertainty score u ∈ [0,1]

`Reference Set G(A)`

KG entities corresponding to oracle-provided answer set A

`Query Result Q(G)`

Induced answer set for graph query Q over G

Quality Measures

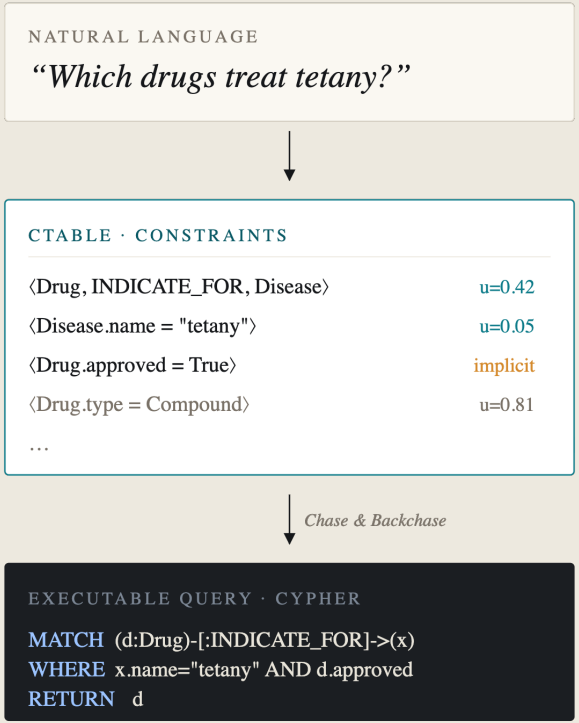
LLM-Sound *No spurious answers*	`Q(G) ⊆ G(A)`
LLM-Complete *All answers covered*	`G(A) ⊆ Q(G)`
Consistent *Constraints respected*	`Every binding in Q(G) satisfies all c ∈ C`
Minimal *Concise queries*	`No redundant constraints`

Goal: Given an NL query Q_n, reference answer set A, KG G, and CTable C extracted from Q_n , generate a query Q that is (i) consistent with C, (ii) jointly maximizes LLM-complete & LLM-sound, and (iii) minimal.

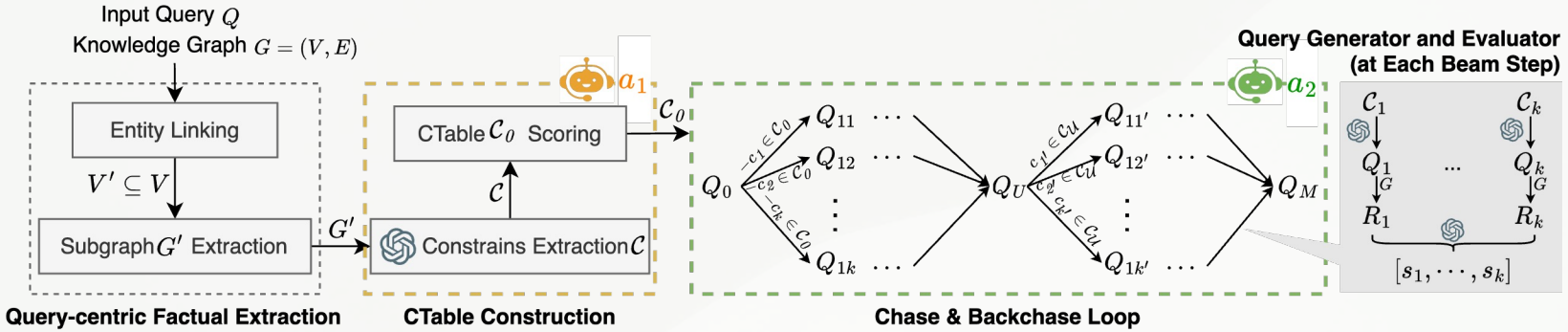
UniQGen

A unified graph-query generation framework.

- → LLM agents extract and refine constraints from the knowledge base interactively;
- → A budgeted "Chase & Backchase" loop with relative quality guarantees;
- → Supports multiple query languages with an LLM-assisted pluggable renderer.



UniQGen Framework Overview



1. Factual extraction
Entity linking + k-hop subgraph

→

2. CTable construction
Agent a_1 extracts and scores constraints

→

3. Chase & Backchase
Agent a_2 searches over constraint subsets

→

4. Rendering
Pluggable renderer → Cypher / SPARQL

Module 1: CTable Construction

Language-agnostic intermediate representation for graph queries

Example: “Which cities in the USA hosted the Olympics in February?”

ID	Constraint candidate	Role
c_1	(c:City)-[:hosted]->(e:OlympicGames)	topology
c_2	c.country = "USA"	value filter
c_3	e.year > 1896	temporal filter
c_4	e.type = "Winter"	implicit intent
c_5	e.type = "Summer"	value filter

Uncertainty Scoring

For each constraint c_i:

$$u_i = n_i / m$$

n_i = match count of c_i in G
m = max match count

Higher u → less specific
u = 0 → pruned

👉*Normalized to a monotone core, dropping a constraint can only enlarge the answer set.*

Module 2: Constraint-guided Query Generation

QChase (Top-down) — Completeness-Guided Beam Search

Goal: Derive universal query Q_U ensuring $G(A) \subseteq Q_U(G)$ — maximize LLM-complete

1. Start with Q_o enforcing all constraints in CTable C_o
2. Spawn children by removing one constraint at a time
3. Score: $s'(c,Q) = \alpha(1-c.u) + (1-\alpha)Q.p$
4. Keep top-b candidates; repeat until $p \geq \tau$

QBackchase (Bottom-up) — Soundness & Minimality

Goal: Derive minimal query Q_M ensuring $Q_M(G) \subseteq G(A)$ — maximize LLM-sound

1. Decompose Q_U into single-constraint subqueries
2. Iteratively combine subqueries via beam search
3. Evaluator verifies soundness against reference set A
4. Output minimal query Q_M

Deployment on Amazon Neptune

Architecture

- Freebase (2.9B triples, 116M entities) on Neptune
- openCypher-over-RDF: dual-language endpoint
- AWS EC2 c5.4xlarge with autoscaling
- LangChain agents via Amazon Bedrock
- Model-agnostic across LLM providers

Cost Control & Guardrails

- Batched evaluator (1 LLM call/beam step)
- Parallel candidate generation + execution
- Bounded beam width/depth + early stopping
- Cached entity links & subgraphs
- Auto-tunable beam parameters

Portability: (i) Training-free → adapt to new KG by updating synonym/hint lists. (ii) Pluggable renderer emits Cypher or SPARQL from the same plan. (iii) IaC recipe + benchmark harness provided for rapid deployment.

Experimental Setup

Benchmarks

Dataset	Test Size	Train:Test	Characteristics
GraphQ	2,395	≈ 0.99	Characteristic-rich, low-resource
GrailQA	6,763	≈ 6.56	I.I.D. / Compositional / Zero-shot
WebQSP	1,639	≈ 1.89	Diverse, amenable to ICL

** All methods query the same Neptune-hosted Freebase. Same entity linking for fair comparison.*

Methods Compared

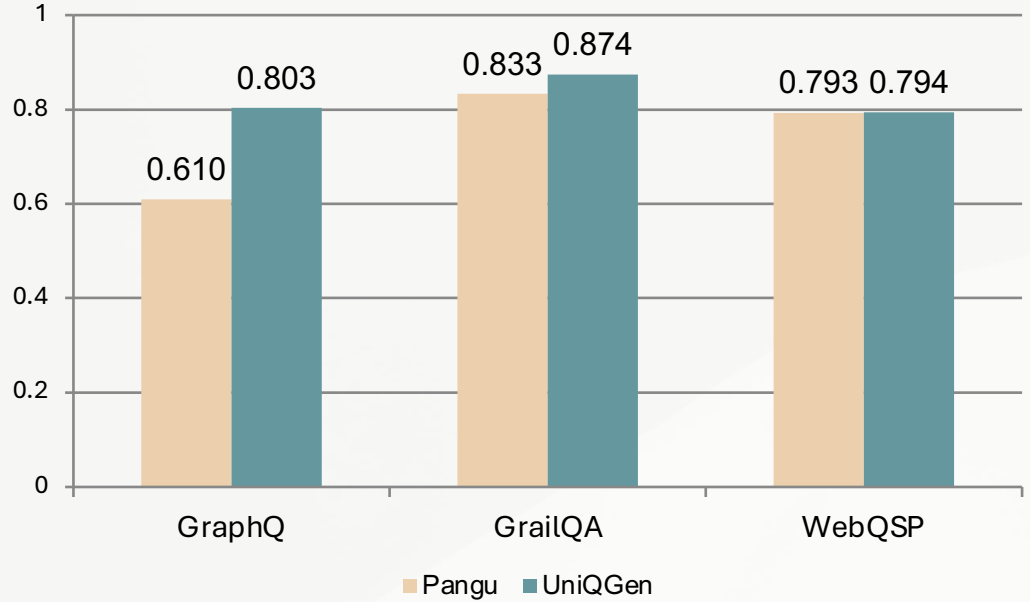
UniQGen (Ours) [Training-free] LLM-based generator + CaB refinement; cross-language (SPARQL / Cypher);

Pangu [Trained] SOTA hybrid of symbolic search + schema-fine-tuned LM; SPARQL only;

ArcaneQA [Trained] Step-wise SPARQL decoder; re-encodes relevant KG schema at each step; SPARQL only;

Prompt-Only [Training-free] Same prompt as UniQGen's generator, no CaB refinement.

Effectiveness



vs. Pangu (fine-tuned SOTA)

+31.6%

F1 GAIN ON GRAPHQ

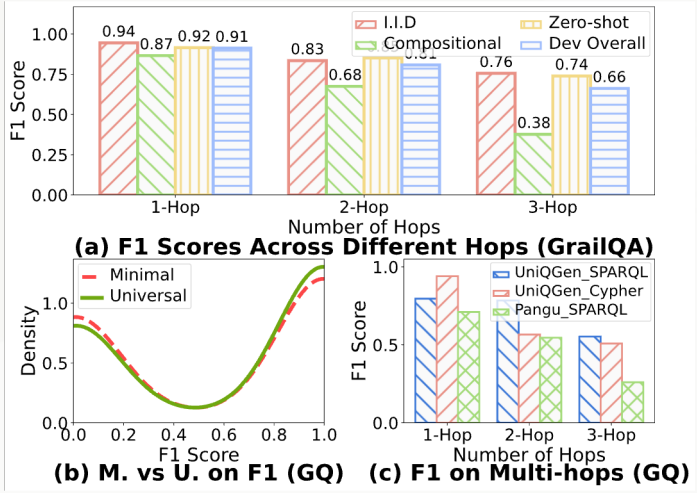
+4.9%

F1 GAIN ON GRAILQ

- Largest gains on GraphQ, where train:test ≈ 0.99 — the lowest among the three;
- On WebQSP, where in-context learning baselines benefit most, UniQGen remains on par (0.794 vs 0.793).
- UniQGen is training-free; Pangu and ArcaneQA require task-specific fine-tuning.

In-depth Analysis

- **Robust on multi-hop:** all methods degrade as hop count grows, but UniQGen degrades the slowest on GraphQ.
- **Bimodal F1:** failures are concentrated — UniQGen either nails the plan or misses one critical constraint.
- **Universal favors recall, Minimal favors precision:** neither always wins, so picking the better one per query helps.



Runtime validation absorbs oracle noise: oracle agrees with ground-truth ~80% of the time, yet F1 drops only 3.9%.

Reference Set	P	R	F1	EM
LLM oracle	0.813	0.857	0.804	0.740
Ground-truth	0.833	0.898	0.835	0.800

Oracle ablation (GraphQ): swap LLM oracle for ground-truth.

3.9%

F1 loss vs. ground-truth oracle

Efficiency & Scalability

Zero

task-specific
training
for new benchmarks

~40s

Avg. query
generation time

Including CaB loop

0.25s

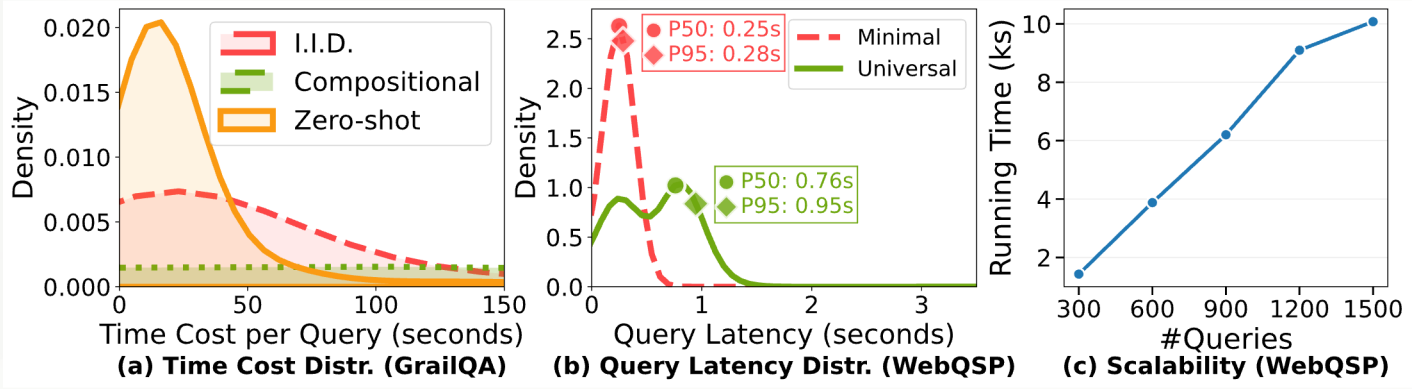
P50 KG execution
latency (Minimal)

P95: 0.28s

1500+

Concurrent requests
with stable latency

No degradation observed



Summary

i.

UniQGen pipeline

- Constraint extraction → CTable → Chase & Backchase refinement;
- Schema-less, unified KGQA at low latency; and
- No fine-tuning required.

ii.

Constraint-based reformulation

- LLM agents explore and rank candidate constraints via beam search;
- Chase & Backchase enforces LLM-complete, LLM-sound, and minimality; and
- Extends to other query languages with modest effort.

iii.

Open resources

- A Neptune-ready Freebase snapshot over SPARQL/Cypher;
- A one-step deployment recipe; and
- Gold Cypher pairs aligned with SPARQL benchmarks.

ICDE 2026
MONTRÉAL, CANADA

Thank You!

Contact: mxw767@case.edu (Mengying Wang)

amazon CWRU