

# Graph Database Backends para AI Retrieval

Comparativa de rendimiento + evaluacion de query generation por LLM

fn\_registry: 393 nodos, 395 aristas

## RESULTADOS CLAVE

Benchmark de rendimiento (393 nodos, 395 aristas):

Mas rapido en queries: igrph (0.7ms para 8 queries)  
Mas rapido en insert: igrph (0.5ms)  
Menor disco: igrph (0.04MB)  
Mejor cold start: SQLite (0.2ms)

LLM Query Generation (claude -p haiku, 40 queries):

SQL (SQLite): 8/8 ejecutan sin error (100%)  
SPARQL (RDFLib): 7/8 ejecutan sin error (87.5%)  
Cypher (Kuzu): 6/8 ejecutan sin error (75%)  
Cypher (Memgraph): 6/8 ejecutan sin error (75%)  
Python (NetworkX): evaluacion manual

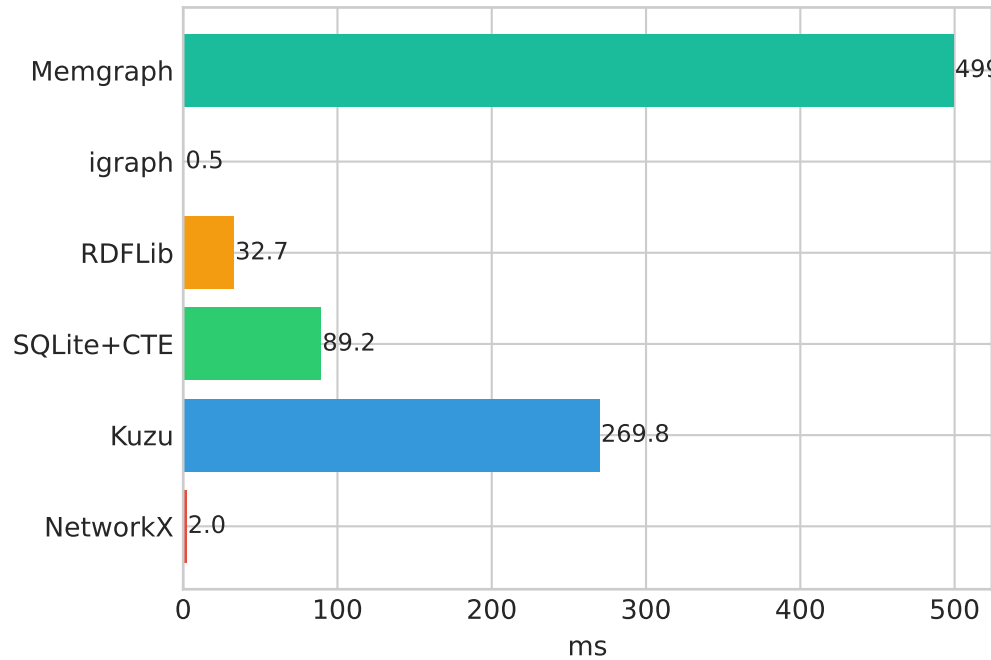
## RECOMENDACION:

Para AI retrieval: SQLite + CTEs recursivos

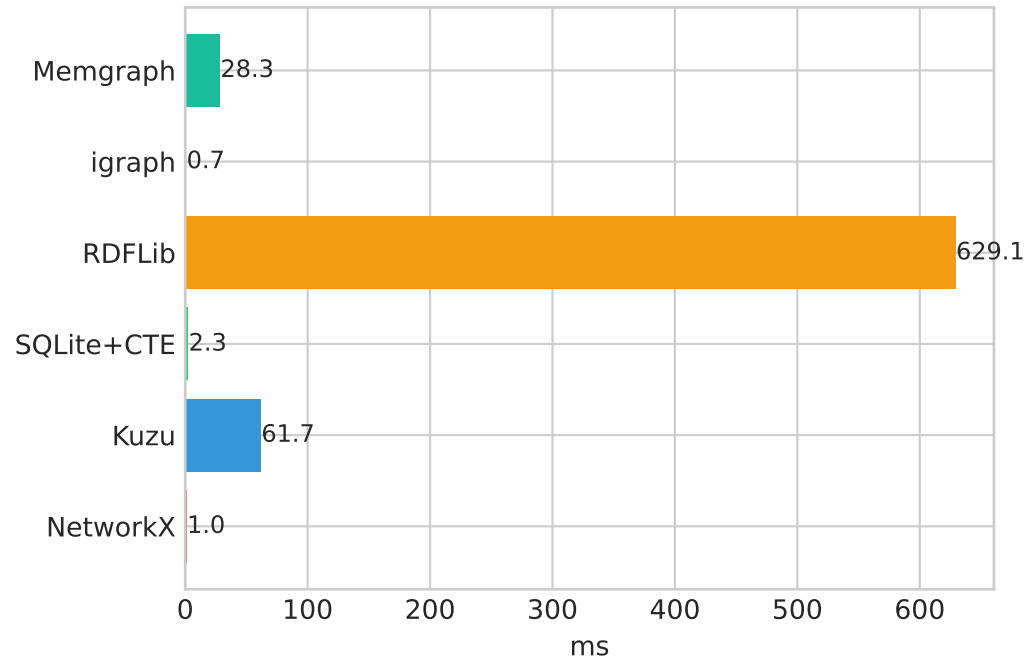
- 100% tasa de queries ejecutables por LLM
- Cold start mas rapido (0.2ms)
- Ya integrado en fn\_registry stack
- Query language mas conocido por LLMs

# Benchmark de Graph Backends

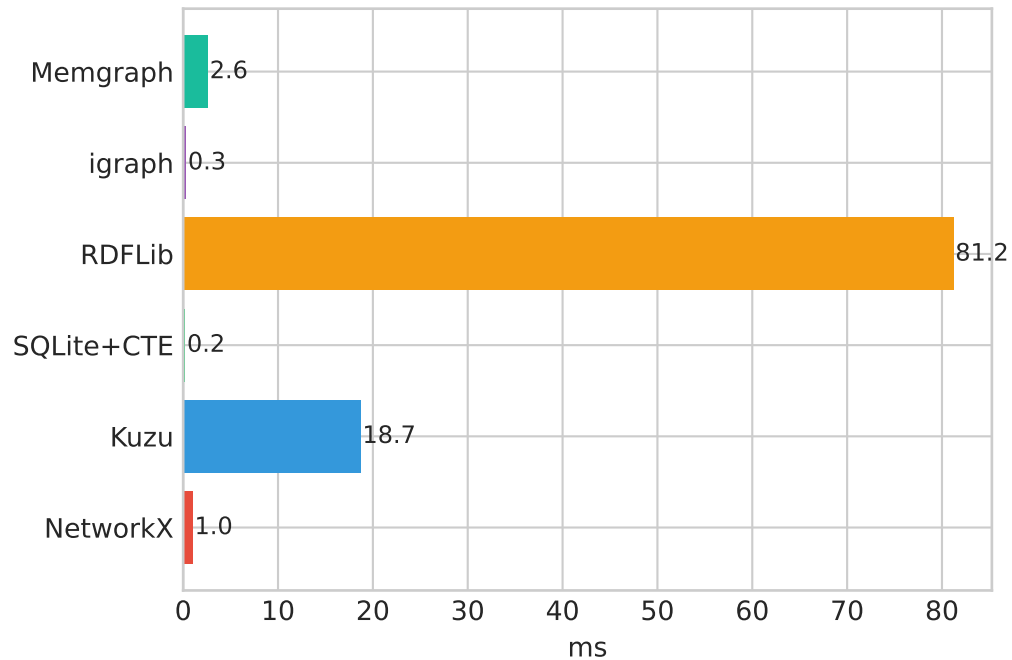
## Insert (nodos + aristas)



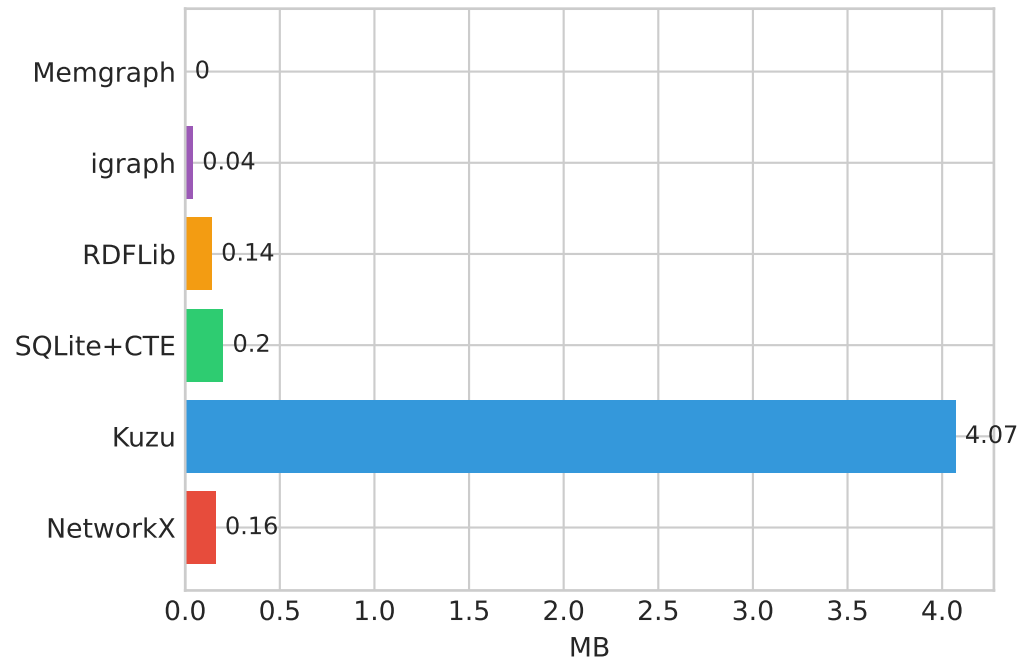
## 8 queries de traversal



## Cold start: load + 1 query

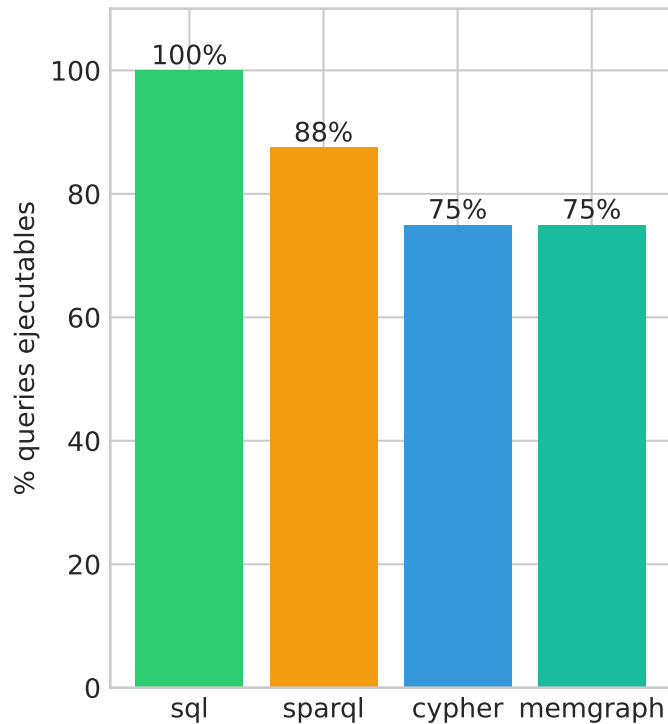


## Tamano en disco

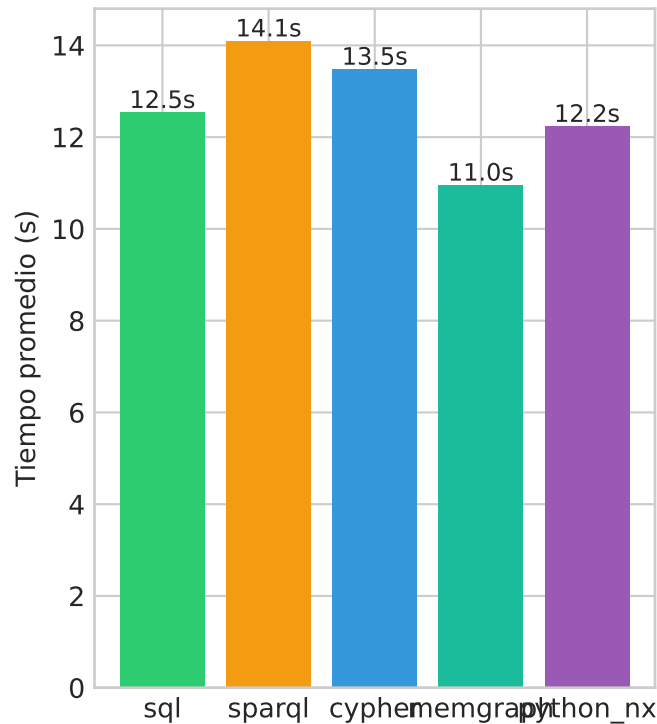


# LLM Query Generation (claude -p haiku)

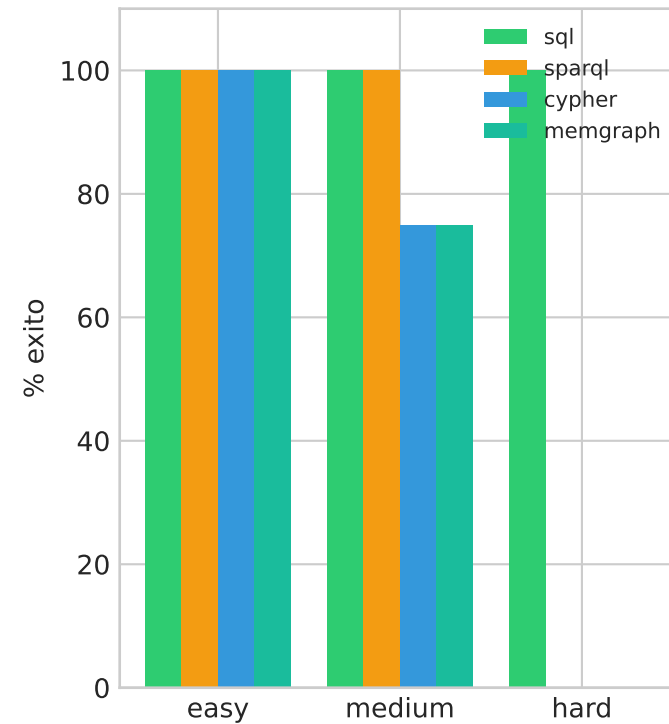
## Tasa de éxito por backend



## Tiempo de generacion



## Exito por dificultad



Detalle de queries generadas por LLM				
Question	Query	Difficulty	Query Time	Execution
q1_direct	cypher	easy	13.99s	OK (0 rows)
q1_direct	sql	easy	13.3s	OK (0 rows)
q1_direct	sparql	easy	13.69s	OK (0 rows)
q1_direct	python_nx	easy	11.82s	MANUAL
q1_direct	memgraph	easy	10.47s	OK (0 rows)
q2_reverse	cypher	easy	13.48s	OK (176 rows)
q2_reverse	sql	easy	12.37s	OK (166 rows)
q2_reverse	sparql	easy	12.51s	OK (0 rows)
q2_reverse	python_nx	easy	12.54s	MANUAL
q2_reverse	memgraph	easy	9.01s	OK (176 rows)
q3_twohop	cypher	medium	15.42s	OK (0 rows)
q3_twohop	sql	medium	14.18s	OK (0 rows)
q3_twohop	sparql	medium	12.49s	OK (0 rows)
q3_twohop	python_nx	medium	10.32s	MANUAL
q3_twohop	memgraph	medium	9.8s	OK (0 rows)
q4_domain	cypher	medium	11.86s	OK (10 rows)
q4_domain	sql	medium	10.18s	OK (14 rows)
q4_domain	sparql	medium	12.97s	OK (0 rows)
q4_domain	python_nx	medium	12.42s	MANUAL
q4_domain	memgraph	medium	9.95s	OK (14 rows)
q5_degree	cypher	medium	16.3s	FAIL
q5_degree	sql	medium	15.58s	OK (5 rows)
q5_degree	sparql	medium	19.48s	OK (5 rows)
q5_degree	python_nx	medium	15.17s	MANUAL
q5_degree	memgraph	medium	11.23s	FAIL
q6_path	cypher	hard	12.06s	FAIL
q6_path	sql	hard	15.02s	OK (1 rows)
q6_path	sparql	hard	14.06s	FAIL
q6_path	python_nx	hard	14.55s	MANUAL
q6_path	memgraph	hard	16.63s	FAIL
q7_isolated	cypher	easy	9.08s	OK (122 rows)
q7_isolated	sql	easy	9.21s	OK (122 rows)
q7_isolated	sparql	easy	17.87s	OK (122 rows)
q7_isolated	python_nx	easy	9.99s	MANUAL
q7_isolated	memgraph	easy	10.03s	OK (122 rows)
q8_typed	cypher	medium	15.64s	OK (0 rows)
q8_typed	sql	medium	10.45s	OK (0 rows)
q8_typed	sparql	medium	9.69s	OK (0 rows)
q8_typed	python_nx	medium	11.05s	MANUAL
q8_typed	memgraph	medium	10.5s	OK (0 rows)

# Validacion cruzada + Queries fallidas

VALIDACION CRUZADA (todos los backends dan el mismo resultado)

```
=====
direct_deps      : ALL OK
reverse_deps     : ALL OK
two_hop          : ALL OK
isolated         : ALL OK
type_users       : ALL OK
path_exists      : ALL OK
most_connected   : DIFF: ['Kuzu', 'RDFLib', 'Memgraph']
domain_subgraph  : ALL OK
```

QUERIES FALLIDAS DEL LLM

=====

[q5\_degree] cypher:

```
Error: Binder exception: Variable m is not in scope.
Query: MATCH (n:FnNode)
RETURN n.id, n.name,
       size([(n)-[:DEPENDS_ON]->(m) | m]) as outDegree,
       size([(m)-[:DEPENDS_ON]->(n) | m]) as inDegree,
...
```

[q5\_degree] memgraph:

```
Error: {neo4j_code: Memgraph.TransientError.MemgraphError.MemgraphError} {message: Not yet implemented: Exi
Query: MATCH (n:FnNode)
RETURN n.id, n.name, size((n)-[]->()) + size(((n)-[]->(n))) as total_degree
ORDER BY total_degree DESC
LIMIT 5...
```

[q6\_path] cypher:

```
Error: Parser exception: Invalid input <MATCH (start:FnNode {domain: "finance"})-[:DEPENDS_ON*1..5]->(end>:
Query: MATCH (start:FnNode {domain: "finance"})-[:DEPENDS_ON*1..5]->(end:FnNode {id: "error_go_core"})
RETURN COUNT(*) > 0 AS exists_path...
```

[q6\_path] sparql:

```
Error: Expected AskQuery, found '?' (at char 262), (line:9, col:3)
Query: PREFIX fn: <http://fn-registry.local/>
PREFIX fnrel: <http://fn-registry.local/rel/>
PREFIX fnprop: <http://fn-registry.local/prop/>
PREFIX rdf: <http...
```

[q6\_path] memgraph:

```
Error: {neo4j_code: Memgraph.ClientError.MemgraphError.MemgraphError} {message: Unbound variable: path.} {g
Query: MATCH (f:FnNode {domain: 'finance'})-[*1..5]-(e:FnNode {id: 'error_go_core'})
RETURN f.id as finance_function, e.id as error_target, length(path) as h...
```

# Recomendaciones para AI Graph Retrieval

## RANKING PARA USO CON LLMs (AI RETRIEVAL)

1. SQLite + CTEs recursivos [RECOMENDADO]
  - + 100% tasa de queries ejecutables por LLM
  - + Cold start mas rapido (0.2ms) – ideal para agentes efimeros
  - + Query time competitivo (2.3ms para 8 queries)
  - + Ya integrado en fn\_registry (registry.db usa SQLite)
  - + SQL es el lenguaje de query mas conocido por LLMs
  - CTEs recursivos son verbosos para traversal profundo
2. Cypher (Kuzu embebido)
  - + Expresivo para patrones de grafo complejos
  - + Variable-length paths nativos
  - + Persistencia en disco (4.07MB)
  - 75% tasa de exito – falla en degree counting y paths complejos
  - Insert lento (270ms) vs igraph/NetworkX
3. Cypher (Memgraph via Docker)
  - + Misma expresividad Cypher + full graph DB features
  - + Reconnect rapido (2.6ms)
  - Requiere Docker – overhead operativo
  - 75% tasa de exito (mismos problemas que Kuzu)
  - 174MB RAM para 393 nodos
4. SPARQL (RDFLib)
  - + 87.5% tasa de exito – mejor de lo esperado
  - + Estandar W3C, buen soporte en LLMs
  - Queries muy lentas (629ms para 8 queries)
  - Sintaxis verbose para operaciones simples
5. Python API (NetworkX/igraph)
  - + Mas rapido (igraph: 0.7ms, NetworkX: 1ms)
  - + Evaluacion manual necesaria – no hay lenguaje de query
  - Requiere que el agente ejecute codigo Python arbitrario
  - No apto para agentes con acceso limitado

## CONCLUSION:

Para fn\_registry, SQLite + CTEs es la opcion optima:

- El agente ya tiene acceso a registry.db
- SQL es el lenguaje mas fiable para LLM query generation
- No requiere infraestructura adicional
- Las queries recursivas cubren el 100% de los patrones de grafo necesarios