cypher_dart Technical Report

Jaichang Park
Google Developer Expert (GDE)*

February 22, 2026

Abstract

This report describes the currently implemented architecture and scope of `cypher_dart`, a pure Dart Cypher toolkit with parser diagnostics, canonical formatting, and an in-memory execution engine. The implementation is intentionally clause-oriented and pragmatic: it provides strong tooling support for common OpenCypher query flows while keeping strict boundaries around unsupported features. We document the parser and diagnostics pipeline, language coverage, execution semantics, and a reproducible local evaluation plan based on tests, TCK smoke coverage scripts, and a built-in microbenchmark driver.

1 Introduction

Cypher tooling in Dart and Flutter commonly needs four capabilities: syntactic validation during editing, actionable diagnostics with source locations, deterministic query normalization, and lightweight execution for local experimentation. `cypher_dart` targets this combination in a single package, with strict OpenCypher behavior by default and explicit feature gating for selected Neo4j-oriented syntax.

The current release (package version `0.1.1`) is best characterized as a production-ready tooling core plus an MVP execution runtime. It is not a full spec-complete database runtime. This report focuses on implemented behavior in the repository rather than projected roadmap items [7].

*Dart-Flutter

2 Background

openCypher defines a vendor-neutral language reference, grammar artifacts, and a Technology Compatibility Kit (TCK) used by implementations to assess behavioral compatibility [18, 16, 17]. `cypher_dart` vendors openCypher resources in-tree and uses them as the primary reference baseline.

The package also includes an ANTLR generation utility and generated parser artifacts in `lib/src/generated`; however, the active parse pipeline described below is a lightweight clause lexer plus AST builder. The ANTLR path is currently auxiliary infrastructure and not the default execution path [20, 7].

3 Related Work

We reviewed fifteen arXiv papers for language semantics, path-query semantics, property-graph integrity, and system execution. We also cross-checked implementation context using current web sources for standardization, query manuals, and benchmark tooling.

3.1 Cypher, language semantics, and expressiveness

Formal foundations for Cypher were established by a denotational account of query semantics and value domains [9]. Recent work extends this direction toward proof-oriented equivalence checking for Cypher queries [24]. Expressiveness boundaries for property-graph querying over relational backends are studied in [21], which is directly relevant when building portability layers between graph-native and relational runtimes.

3.2 Path semantics and recursive query execution

Regular path query (RPQ) semantics continue to evolve, including run-based semantics [8] and comparative analysis of RPQ semantics [12]. Property constraints over paths are formalized in [19], while executable evaluation systems such as PathDB focus on operational treatment of regular path queries [10]. On the engine side, recursive execution and compilation are addressed through cross-paradigm recursive query compilation [22] and robust recursive parallelism in graph DBMSs [5]. Fast dual simulation remains a reference point for pattern-query execution tradeoffs [13].

3.3 Property-graph constraints, schema, and transformations

Property-graph data quality and maintainability are covered by schema validation and schema evolution [2], trigger mechanisms for property-graph updates [4], graph transformation frameworks [3], and explicit repair under property-graph constraints [23]. Together, these works frame the design space for moving from parser-level correctness toward state-consistent execution.

3.4 System and ecosystem references from web sources

A recent survey of graph databases provides high-level system taxonomy and capability framing [6]. For practical language behavior and user-facing semantics, the current Neo4j Cypher manual is an implementation reference [14]. The broader ecosystem includes Apache AGE as another open property-graph implementation target [1]. Standardization context is commonly summarized around ISO/IEC 39075 GQL [25]. For benchmark-style workload thinking, LDBC SNB data-generation assets remain a useful operational reference [11].

4 System Architecture

4.1 Parser and Diagnostics Pipeline

The parser front end is organized as:

1. Source mapping: input text is wrapped by a `SourceMapper` to produce stable spans (line, column, offset).
2. Extension gate pre-scan: strict mode checks for selected Neo4j syntax using feature gates (`CYP201`–`CYP204`).
3. Clause lexing: statements are split on top-level semicolons, then clause keywords are extracted at top level (quote and bracket aware).
4. AST building: lexed clauses map to typed clause nodes (`MatchClause`, `ReturnClause`, etc.) with raw clause bodies.
5. Semantic validation: current checks include clause ordering constraints (`CYP300`), duplicate projection aliases (`CYP301`), and duplicate `RETURN` in one statement (`CYP302`).

Parse options expose:

- Dialect mode (`openCypher9` or `neo4j5`),
- Explicit feature enablement in strict mode,
- Recovery mode (`recoverErrors`) for partial documents.

In fail-fast mode, any error yields `document = null`. In recovery mode, partial AST output is retained when possible.

4.2 AST Representation and Formatter

The AST is clause-level. Each clause stores:

- A normalized keyword,
- Raw body text,
- Source span.

The formatter (`CypherPrinter`) applies canonical uppercase keywords, stable clause ordering in output, statement separation by semicolons, and whitespace normalization inside clause bodies. Because clause bodies remain string-based, formatting is deterministic but not token-preserving.

4.3 In-Memory Execution Engine

`CypherEngine.execute` performs parse-then-run. If parse errors exist, execution is skipped and parse diagnostics are returned. Runtime failures are reported separately as execution errors.

Execution is row-pipeline based:

1. Seed with one empty row,
2. Apply clauses sequentially to transform row sets,
3. Project final records and columns.

The runtime uses an `InMemoryGraphStore` with immutable node, relationship, and path value objects returned to the caller.

5 Language Coverage

Table 1 summarizes implemented scope.

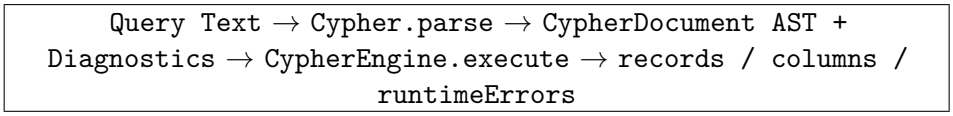

Figure 1: High-level processing pipeline in `cypher_dart`.

6 Execution Model

The engine is designed for deterministic local semantics, not distributed or persistent execution.

Data model. Nodes and relationships are identified by integer IDs and store label/type and property maps. Paths are ordered node/relationship sequences.

Clause semantics. The runtime supports read and write clause chains, including filtering, projection, ordering, aggregation, updates, deletion, and unions. `CALL` is restricted to an in-memory procedure set. `WHERE` uses strict boolean coercion with three-valued logic for `null` propagation.

Expression evaluation. The evaluator supports arithmetic, comparison chains, map/list literals, list/pattern comprehensions, selected temporal constructors, and common scalar and aggregate functions. Error conditions (invalid operators, wrong argument types, unsupported forms) raise explicit runtime errors instead of silent fallback behavior.

Error boundary. The parse phase and runtime phase are explicitly separated: parse errors stop execution; runtime errors return with parse success metadata.

Simple runtime cost model. For a query Q decomposed into k clause-level operators over graph G, we use the following approximation to reason about scaling trends:

$$T(Q, G) \approx \sum_{i=1}^{k} \left(\alpha_i |R_i| + \beta_i |R_i| \log(1 + |R_i|)\right),$$

where R_i is the intermediate row set after operator i, and α_i, β_i are operator-specific constants. This is a practical model for performance discussion, not a strict optimizer guarantee.

7 Evaluation Plan

7.1 Current Local Validation Assets

Repository validation already includes:

- Parser, diagnostics, formatter, and engine unit tests,
- TCK parse coverage script (`tool/tck_parse_coverage.dart`),
- TCK execution smoke script (`tool/tck_execute_coverage.dart`),
- Engine benchmark driver (`tool/benchmark_engine.dart`),
- Release check script chaining format/analyze/test/doc generation.

7.2 Benchmark Method Available Locally

The benchmark driver reports average microseconds per operation and operations per second for parse and execute scenarios (simple parse, complex parse, filter/projection execution, relationship matching, aggregation, and `MERGE` update path). A standard local run is:

```
dart run tool/benchmark_engine.dart \
  --iterations 300 --warmup 60 --nodes 2000
```

The script is intentionally launcher-dependent and should be used for relative regression tracking on a fixed machine profile, not for cross-machine absolute claims.

7.3 Coverage Snapshot (2026-02-22)

The current repository includes `coverage/lcov.info`. We report line coverage using:

$$\text{LineCoverage} = \frac{\text{LH}}{\text{LF}} \times 100.$$

From the latest local snapshot: LF = 2548, LH = 2328, which gives **91.37%** line coverage.

7.4 Benchmark Snapshot (2026-02-22)

For benchmark runs with iteration count N and elapsed wall-clock time t_{total}, we compute:

$$\text{ops/s} = \frac{N}{t_{\text{total}}}, \qquad \text{avg\_ms/op} = 1000 \cdot \frac{t_{\text{total}}}{N}.$$

Using:

```
dart run tool/benchmark_engine.dart \
  --iterations 300 --warmup 60 --nodes 2000
```

we obtained the following local snapshot.

7.5 Reproducibility Checklist

8 Discussion and Limitations

Current limitations are explicit and material:

- The parser is clause-oriented, not a full grammar-derived parser with a complete expression AST.
- Some syntactic and semantic issues are detected at runtime instead of parse time because clause bodies are string-based.
- Neo4j extension detection in strict mode is pattern-based and can miss edge forms outside the implemented detectors.
- Execution scope is MVP by design: no transactions, no persistent storage, no planner/index subsystem, and restricted procedure ecosystem.
- Not all openCypher TCK scenarios are expected to pass; provided scripts are coverage/smoke tools rather than a full conformance certificate.
- The repository ships ANTLR generation tooling, but generated artifacts are not the default active parser path at runtime in the current build.

These constraints are acceptable for editor tooling, validation workflows, and small in-memory experimentation, but they bound applicability for full database-engine use cases.

9 Conclusion

`cypher_dart` currently delivers a cohesive Dart-native Cypher toolchain: clause-level parsing with source-aware diagnostics, stable formatting, and an in-memory execution runtime that supports substantial read/write query workflows. The implementation is intentionally pragmatic and transparent about unsupported scope. The near-term technical leverage is clear: retain deterministic tooling behavior while progressively expanding grammar depth, semantic analysis breadth, and conformance coverage against openCypher artifacts [15, 17].

References

[1] Apache Software Foundation. Apache age documentation. `https://age.apache.org/`. Accessed 2026-02-22.

[2] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. Schema validation and evolution for graph databases, 2019.

[3] Angela Bonifati, Filip Murlak, and Yann Ramusat. Transforming property graphs, 2024.

[4] Stefano Ceri, Anna Bernasconi, Alessia Gagliardi, Davide Martinenghi, Luigi Bellomarini, and Davide Magnanimi. Pg-triggers: Triggers for property graphs, 2023.

[5] Anurag Chakraborty and Semih Salihoglu. Robust recursive query parallelism in graph database management systems, 2025.

[6] Miguel E. Coimbra, Lucie Svitakova, Alexandre P. Francisco, and Luis Veiga. Survey: Graph databases, 2025.

[7] cypher_dart contributors. cypher_dart. `https://github.com/jaichang/cypher_dart`. GitHub repository, accessed 2026-02-22.

[8] Claire David, Victor Marsault, and Nadime Francis. Run-based semantics for rpqs, 2022.

[9] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andres Taylor. Formal semantics of the language cypher, 2018.

[10] Roberto Garcia, Renzo Angles, Vicente Rojas, and Sebastian Ferrada. Pathdb: A system for evaluating regular path queries, 2025.

[11] Linked Data Benchmark Council. Ldbc snb datagen. `https://github.com/ldbc/ldbc_snb_datagen_hadoop`. Accessed 2026-02-22.

[12] Victor Marsault and Antoine Meyer. Designing and comparing rpq semantics, 2026.

[13] Stephan Mennicke, Jan-Christoph Kalo, Denis Nagel, Hermann Kroll, and Wolf-Tilo Balke. Fast dual simulation processing of graph database queries (supplement), 2018.

[14] Neo4j. Cypher manual. `https://neo4j.com/docs/cypher-manual/current/`. Accessed 2026-02-22.

[15] openCypher contributors. opencypher. `https://github.com/opencypher/openCypher`. Repository containing specification, grammar, and TCK; accessed 2026-02-22.

[16] openCypher contributors. opencypher grammar (bnf). `https://github.com/opencypher/openCypher/blob/master/grammar/openCypher.bnf`. Accessed 2026-02-22.

[17] openCypher contributors. opencypher technology compatibility kit (tck). `https://github.com/opencypher/openCypher/tree/master/tck`. Accessed 2026-02-22.

[18] openCypher contributors. opencypher 9. `https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf`, 2018. Language reference.

[19] Fernando Orejas, Elvira Pino, Renzo Angles, Edelmira Pasarella, and Nikos Milonakis. Properties for paths in graph databases, 2025.

[20] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[21] Hadar Rotschield and Liat Peterfreund. On the expressiveness of languages for querying property graphs in relational databases, 2025.

[22] Amir Shaikhha, Youning Xia, Meisam Tarabkhah, Jazal Saleem, and Anna Herlihy. Raqlet: Cross-paradigm compilation for recursive queries, 2025.

[23] Christopher Spinrath, Angela Bonifati, and Rachid Echahed. Repairing property graphs under pg-constraints, 2026.

[24] Lei Tang, Wensheng Dou, Yingying Zheng, Lijie Xu, Wei Wang, Jun Wei, and Tao Huang. Proving cypher query equivalence, 2025.

[25] Wikipedia contributors. Graph query language. `https://en.wikipedia.org/wiki/Graph_Query_Language`. Summary of ISO/IEC 39075 (GQL), accessed 2026-02-22.

Table 1: Implementation support matrix (current repository state).

Capability	Status	Notes
Clause AST + formatter for core clauses: `MATCH`, `WHERE`, `WITH`, `RETURN`, `ORDER BY`, `SKIP`, `LIMIT`, `CREATE`, `MERGE`, `SET`, `REMOVE`, `DELETE`, `DETACH DELETE`, `UNWIND`, `CALL`, `UNION`, `UNION ALL`	Supported	Typed clause nodes with source spans.
Clause-order and alias diagnostics (`CYP300`–`CYP302`)	Supported	Includes duplicate `RETURN` and duplicate projection aliases.
Strict-mode feature gates for `EXISTS {...}`, `CALL {...} IN TRANSACTIONS`, pattern comprehension, `USE`	Supported	Enforced unless enabled by feature flags or `neo4j5` dialect.
Expression-level parse tree in AST	Not yet	Clause bodies are stored as raw strings.
MATCH relationship patterns (direction, alternation, variable length)	Supported	Includes path binding and variable-length matching in `MATCH`.
Variable-length relationships in `CREATE`/`MERGE`	Not yet	Explicit runtime errors for these shapes.
`MERGE ... ON CREATE SET ... ON MATCH SET ...`	Supported	Clause-local `SET` chains implemented.
`CALL` procedures	Partial	Built-ins: `db.labels()`, `db.relationshipTypes()`, and `db.propertyKeys()`; test procedures are also implemented for harnessing.
Pattern/list comprehensions and scalar/aggregate expressions in engine	Supported (MVP)	Broad expression support, with runtime validation and explicit unsupported cases.
Transactions, persistence, indexes, cost-based planning, storage engine	Not yet	Runtime is intentionally in-memory and single-process.

Table 2: Coverage snapshot from `coverage/lcov.info` (2026-02-22).

Scope	LH / LF	Line coverage
Overall (`lib/src`)	2328 / 2548	91.37%
`lib/src/engine/engine.dart`	1778 / 1992	89.26%
`lib/src/engine/graph.dart`	78 / 78	100.00%

Table 3: Local benchmark snapshot (2026-02-22).

Scenario	avg_ms/op	ops/s
`parse_simple`	0.0627	15945.6
`parse_complex`	0.0453	22070.2
`execute_filter_projection`	14.0994	70.9
`execute_relationship_match`	11.2954	88.5
`execute_aggregation`	8.1554	122.6
`execute_merge_on_match`	0.0404	24756.6

Table 4: Reproducibility checklist for this report.

Item	Status	How to verify
SDK bounds and dependency lockfile tracked	Yes	`pubspec.yaml`, `pubspec.lock`
Deterministic test entry points committed	Yes	Run `dart test` in `test/`.
TCK parse coverage harness available	Yes	Run `dart run` on `tool/tck_parse_coverage.dart`.
TCK execution smoke harness available	Yes	Run `dart run` on `tool/tck_execute_coverage.dart`.
Engine microbenchmark harness available	Yes	Run `dart run` on `tool/benchmark_engine.dart`.
Document build command documented	Yes	`docs/paper/README.md`
Machine/OS/VM metadata capture automated	No	Record manually when reporting benchmark results