openCypher: New Directions in Property Graph Querying

Alastair Green
Neo4j
alastair.green@neo4j.com

Martin Junghanns
Neo4j & University of Leipzig
martin.junghanns@neo4j.com

Max Kiessling
Neo4j
max.kiessling@neo4j.com

Tobias Lindaaker
Neo4j
tobias.lindaaker@neo4j.com

Stefan Plantikow
Neo4j
stefan.plantikow@neo4j.com

Petra Selmer
Neo4j
petra.selmer@neo4j.com

ABSTRACT

Cypher is a property graph query language that provides expressive and efficient querying of graph data. Originally designed and implemented within the Neo4j graph database, it is now being used by several industrial database products, as well as open-source and research projects. Since 2015, Cypher has been an open, evolving language, with the aim of becoming a fully-specified standard with many independent implementations.

We introduce Cypher and the property graph model, and then describe extensions – either actively being developed or under discussion – which will be incorporated into Cypher in the near future. These include (i) making Cypher into a fully compositional language by supporting multiple graphs and allowing graphs to be returned from queries; (ii) allowing for more complex patterns (based on regular path queries) to be expressed; and (iii) allowing for different pattern matching semantics – homomorphism, relationship isomorphism (the current default) or node isomorphism – to be configured at a query-by-query level.

A subset of the proposed Cypher language extensions has already been implemented on top of Apache Spark. In the tutorial, we will present our approach including an in-depth analysis of the challenges we faced. This includes mapping the property graph model to the Spark DataFrame abstraction and the translation of Cypher query operators into relational transformations. The tutorial will conclude with a demonstration based on a real-world graph analytical use case.

1 INTRODUCTION

The past few years have seen a marked increase of property graph databases [12] – such as Neo4j [20], Sparksee and JanusGraph – in both the industrial and research arenas. Property graphs have become the model of choice for next-generation graph applications[1]. Their use increasingly replaces older approaches to graph data processing such as cross-linked document stores or object-oriented database management systems.

Across both research and industry, property graphs have been used in a wide variety of domains, spanning areas as diverse as fraud detection, recommendations, geospatial data, master data management, network and data centre management, authorisation and access control [23], the analysis of social networks [5], bioinformatics [1, 14, 28] and pharmaceuticals [18], software system analysis [9], and investigative journalism [3].

This trend of increased usage of property graphs is grounded in: (i) their ability to operate on multiple large and highly-connected data sets as one graph that enables novel pattern matching and graph analytical queries; (ii) their natural ability to cleanly map onto object-oriented or document-centric data models in programming languages; (iii) their visual nature that helps communication between business, application domain, and technical experts; and (iv) their historical development based on the pragmatic needs of real world application developers.

This trend is evidenced by two major factors. The first is the emergence of Cypher as the de-facto standard declarative query language for property graphs, and the second is the growing number of both industrial and academic software products for property graphs.

Since 2015, as part of the openCypher project [22], Cypher has been an open language, and is evolving under the auspices of the openCypher Implementers Group (oCIG), with the aim of becoming a fully-specified standard that can be independently implemented. The recently released Cypher 9 reference [21] along with accompanying formal grammar definitions (EBNF and ANTLR4) and conformance test suite (TCK) – published under the Apache 2.0 license – already provide implementers with a solid basis for adopting Cypher. At the time of writing, Cypher is supported by several commercial systems including SAP HANA Graph [24], Agens Graph, Redis Graph, and Memgraph, along with research frameworks including – in varying degrees of completeness – Gradoop [11], inGraph [15], Cytosm [25], Cypher for Apache Spark [19] and Cypher over Gremlin.

Current developments that are under way include the ability to pass multiple graphs and a table as input to a Cypher query. Moreover, queries will also be able to project and save multiple graphs, and this, coupled with the ability to chain queries together, will render Cypher as the first graph compositional query language. Following on from this work, complex pattern matching and configurable pattern matching semantics will further increase the utility of Cypher in the very near future.

2 SCOPE OF THE TUTORIAL

2.1 Intended audience

This tutorial is aimed at a wide scope of audience, including researchers, students, developers, and industrial practitioners who are interested in the emerging and quickly-evolving area of graph data, databases and languages. All attendees will gain a comprehensive idea of what this field comprises, as well as the future features and challenges that lie ahead for Cypher, the most-used property graph query language.

It is our hope that owing to the many challenges that exist in this area, researchers and students will be motivated to consider this area as a future topic of research.

There are no preliminary requirements for this tutorial, as it will be self-contained and commence with the property graph data model and Cypher, thus assuming no prior knowledge of these.

[1] https://db-engines.com/en/ranking/graph+dbms

2.2 Goals of the tutorial

The main outcomes of the tutorial comprise:

- A comprehensive understanding of the property graph data model, and how it compares against some of the other graph data models.
- A good understanding of the Cypher property graph query language and its main constructs and features.
- An in-depth treatment of how Cypher will become a fully compositional language through the introduction of multiple graph support and query chaining.
- An overview of Cypher's version of *regular path queries* in the form of path pattern queries, which additionally include node and relationship property tests to increase the expressivity of Cypher to manage emerging industrial use-cases and requirements.
- An understanding of node and relationship isomorphism and homomorphism, the characteristics of each, the benefits and drawbacks of each (from an industrial point of view) and how these are envisioned to be incorporated into Cypher.
- A good understanding of the Cypher implementation on top of Apache Spark and how to map a schema-free graph data model to a schema-based relational abstraction.
- An understanding of real-world use-cases which can be better solved by using graphs and the proposed language extensions.

3 TUTORIAL OUTLINE

We will begin the tutorial with a brief history of Cypher and the property graph model, and provide an overview of the openCypher project and how this is helping to drive forward the design of the language, before proceeding onto the main topics.

3.1 The Cypher property graph query language

Property graph data model. The property graph data model will be described, along with how it originated historically from application use cases. We will compare and contrast property graphs with other graph data models. The tutorial will also contain a discussion of ongoing work on potential extensions to the property graph data model.

Cypher query language. Cypher as it stands today will be presented, focusing on its core elements: pattern matching, path functionality and how updates to the data are performed. We will also cover how Cypher queries are structured, as this will lead into the topic of query composition further on in the tutorial. To set the scene and lay the foundation for the later topics, we will walk through an example query in detail, describing the syntax and semantics at each stage of the query.

Challenge: language evolution. Evolving a language with active users is not a trivial undertaking. Every language change needs to be understood in terms of a plethora of interlocking concerns such as usability, relevant use cases, consistency, ergonomics, syntax, tractability, implementability and performance, as well as aesthetics. Every design decision may have hidden consequences in terms of constraining the design in the future. We will talk about some of these concerns, how we design language changes, and the formal openCypher process for evolving the language.

3.2 Multiple graphs and composition

Multiple graphs. Having set the scene, we will proceed with describing current developments in Cypher. The first of these is the notion of supporting multiple graphs. We will describe how graphs can be referenced, created, updated, saved as a named graph, and projected. We also define a series of set operations on graphs. Throughout, we will use a running example and describe the syntax and semantics at each stage.

Query composition. Having support for multiple graphs, and being able to return one or more graphs from a query paves the way for true graph query composition. Each query can be considered a function, taking as input a table and multiple named graphs, and returning as output a table and multiple named graphs. Thus, a Cypher query can be thought of as a chain of functions, composed of a series of constituent, elementary queries to form a *query chain* or pipeline. The addition of subqueries – a well-known construct from SQL – may be used to transform a query chain into a tree. Named queries – allowing for queries to be re-used in different contexts – will also be presented. We will illustrate these concepts with examples, and show the power and expressivity conferred through graph query composition; we note that, to our knowledge, no other declarative, widely-used query language allows for this.

Challenge: language revolution. Ideal language additions do not interfere with existing features. Sometimes languages need to be changed so substantially that it is impossible to avoid conflicting with pre-existing semantics. In the history of Cypher, various breaking changes have occurred. We will discuss our experiences with breaking changes, language versioning, and planning and executing large scale additions to the language such that the concerns of all relevant stakeholders are incorporated.

3.3 Powerful pattern matching

Path Pattern Queries (PPQs). Regular path queries were first proposed by Cruz, Mendelzon and Wood [4] in 1987, and now, thirty years later, we have turned our attention to this topic and how it may be included in Cypher in the form of *Path Pattern Queries*, or PPQs.

PPQs, inspired by recent work by Libkin, Martens and Vrgoč [13], extend RPQs with notions of node property tests, and are an extremely powerful and expressive mechanism for graph querying. PPQs have been designed to allow for the composition of paths into more complex ones, incorporating both node and relationship property tests, along with the consideration of path costing. We see this as an integral part of Cypher, particularly as the need for users to express ever more complex patterns becomes more pressing in the near future. Using a running example, we will describe the syntax and semantics in detail.

Configurable pattern-matching semantics. The default pattern-matching semantics in Cypher uses relationship (or edge) isomorphism (referred to informally as 'Cyphermorphism'). Although it has been stated that it is a useful default in most real-world queries [26], there are some cases where a different semantics would be more appropriate. To this end, Cypher will allow the writer of a query to configure the type of pattern-matching semantics the query is to use: either homomorphism, relationship isomorphism, or node isomorphism. We will discuss how this is envisaged to function, and the benefits and drawbacks conferred by each approach.

Challenge: Tractability. Providing more powerful and flexible pattern matching is grounded in ever-growing application requirements. This needs to be balanced against what can be implemented efficiently and what is theoretically tractable. However, in certain cases, these equally valid theoretical and practical viewpoints may be at variance with each other. We will discuss this and provide a perspective on the tensions that exist between theoretical complexity analysis and industrial requirements of graph query languages.

3.4 Cypher for Apache Spark

Graph query languages are currently most prominent in graph database systems such as Neo4j [20]. However, it is our opinion that many systems can benefit from having such a language as part of their feature set. One of these systems is Apache Spark [6], which is one of the most popular open source frameworks in the context of distributed processing of large data volumes within complex analytical workloads.

Apache Spark. Apache Spark is a distributed dataflow framework supporting the declarative definition and execution of distributed dataflow programs sourced from batch data. The basic abstractions of such programs are so-called Resilient Distributed Datasets (RDDs) [29] and transformations between those. A Spark RDD is an immutable, distributed collection of arbitrarily-structured data; transformations are higher-order functions (e.g. map and reduce) that describe the construction of new RDDs either from existing ones or from data sources (e.g., HDFS or RDBMS). To describe an analytical task, a Spark program may include multiple chained transformations. During execution, Spark manages data distribution, parallel execution, load balancing and failover across a cluster of machines.

In addition to the RDD abstraction, Apache Spark includes libraries which offer a higher level of abstraction tailored to specific analytical tasks such as machine learning (SparkML), graph processing (GraphX [8, 27]) and relational operations (SparkSQL). In SparkSQL [2], the abstraction is a so-called DataFrame, which handles structured data according to a fixed schema. Available transformations are well known from relational algebra, comprising, for example, selection, projection, join and grouping. Furthermore, SparkSQL includes Catalyst [2], a rule-based query optimizer that transforms a relational query into an optimized dataflow program by undertaking well-known techniques such as predicate pushdown, column projection and code generation [7].

To incorporate the benefits of Cypher from the graph database domain into the world of distributed dataflow processing, we began developing Cypher for Apache Spark (CAPS) [19]. CAPS is an additional library built on top of SparkSQL and can be integrated into a regular Spark analytical program. CAPS is primarily focused on graph-powered data integration and graph analytical query workloads within the Spark ecosystem. In addition, CAPS is our testbed for Cypher language extensions as specified in the previous sections; for example, query composition, graph transformation and multiple graphs.

Challenge: Schema-flexible mapping. In order to benefit from the query optimization capabilities of Catalyst, we decided to implement CAPS on top of SparkSQL.[2] This however introduces the problem of mapping the schema-flexible property graph model to a schema-fixed DataFrame representation. We solve this problem by defining a graph schema, which includes information about node labels, relationship types and associated properties potentially having conflicting data types. For structured data sources, such as CSV files or RDBMSs, the schema can be derived directly from meta data supplied by the data source. However, unstructured or semi-structured data sources – exemplified by native graph databases such as Neo4j or document databases such as CosmosDB [17] – will require a full scan of the source data to compute the schema, should it not exist in the first instance. Once a schema is available, it is used to split node and relationship data into multiple column entries, i.e., a row inside a structured DataFrame ("flatten out nodes and relationships") resulting in potentially sparse tables. In the tutorial, we will discuss the process of schema computation and node / relationship flattening in detail including more information about our type system.

Challenge: Multi-phase planning. A second challenge we faced when building CAPS was the translation of a Cypher graph query to a sequence of relational operations on the Spark DataFrame API. Our implementation approach is based on our experiences from building the Neo4j query planner as well as several existing publications discussing the formal aspects of that topic [10, 11, 16]. CAPS uses multiple compilation phases to produce an executable Spark program, including: building a canonical query representation from an abstract syntax tree; translating the canonical form into graph-specific query operators (logical planning); computing the schema for intermediate results (flat planning); and translating logical operatos into Spark DataFrame transformations (physical planning). A physical CAPS plan is optimized by Catalyst and translated into an executable Spark program. In the tutorial, we give an introduction into the CAPS compilation phases as well as optimization techniques like reuse of intermediate results, tree rewriting and Catalyst optimization.

3.5 Demonstration

To highlight the analytical benefits of the graph data model as well as the Cypher query language and its proposed extensions, we will end the tutorial with a live demonstration of CAPS. The demonstration will illustrate a hypothetical analytical workflow including graph data integration from multiple data sources, graph transformation and graph analytical queries. We also demonstrate the integration of CAPS within the Spark ecosystem by using Apache Zeppelin, a tool for browser-based interactive data analytics.

4 PRESENTER BIOGRAPHIES

Alastair Green is a Director of Product Management at Neo4j and is a member of the openCypher Language Group, supporting the openCypher project.

Martin Junghanns is part of the Cypher for Apache Spark Engineering team at Neo4j. Apart from that, he is finishing his PhD in Computer Science at the University of Leipzig. Martin is working on the Gradoop project with a focus on distributed graph analytics, graph data models and analytical DSLs.

Max Kiessling is part of the Cypher for Apache Spark Engineering team at Neo4j. He recently finished his Master's thesis at the University of Leipzig, in which he researched distributed pattern matching as part of the Gradoop project.

Tobias Lindaaker is one of the first engineers to have worked at Neo4j. He has been and continues to be a key influencer in

[2] The CAPS architecture is backend-agnostic and can be ported to alternative backends / systems.

the evolution of the property graph model, as well as the Cypher query language.

Stefan Plantikow is a member of the Engineering team at Neo4j and is leading the openCypher Language Group. He has worked on the Neo4j kernel, the original implementation of Cypher and the current Neo4j query planner, as well as the BOLT protocol. Stefan has a Master's degree in Computer Science from Humboldt University, Berlin, in the area of distributed systems.

Petra Selmer is a member of Engineering team at Neo4j and is a member of the openCypher Language Group, supporting the openCypher project. For many years, she worked as a consultant and developer in a variety of different domains and roles and has a PhD in Computer Science from Birkbeck, University of London, where she researched the flexible querying of graph-structured data.

REFERENCES

[1] Davide Alocci, Julien Mariethoz, Oliver Horlacher, Jerven T. Bolleman, Matthew P. Campbell, and Frederique Lisacek. 2015. Property Graph vs RDF Triple Store: A Comparison on Glycan Substructure Search. *PLOS ONE* 10, 12 (12 2015), 1–17.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*.

[3] Mar Cabra. 2016. How the ICIJ used Neo4j to unravel the Panama Papers. Neo4j Blog. (May 2016). https://neo4j.com/blog/icij-neo4j-unravel-panama-papers/.

[4] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *SIGMOD Conference*. ACM Press, 323–330.

[5] Georgios Drakopoulos, Andreas Kanavos, and Athanasios K. Tsakalidis. 2016. Evaluating Twitter Influence Ranking with System Theory. In *Proceedings of the 12th International Conference on Web Information Systems and Technologies, WEBIST 2016, Volume 1, Rome, Italy, April 23-25, 2016*. 113–120.

[6] Apache Software Foundation. 2017. Apache Spark. https://spark.apache.org/. (accessed: Dec. 2017).

[7] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book*.

[8] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.

[9] Nathan Hawes, Ben Barham, and Cristina Cifuentes. 2015. FrappÉ: Querying the Linux Kernel Dependency Graph. In *Proceedings of the GRADES'15 (GRADES'15)*. ACM, 4:1–4:6.

[10] Jürgen Hölsch and Michael Grossniklaus. 2016. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016.*

[11] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. 2017. Cypher-based Graph Pattern Matching in Gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES '17)*.

[12] Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, and David Domínguez-Sal. 2014. Introduction to Graph Databases. In *Reasoning Web (Lecture Notes in Computer Science)*, Vol. 8714. Springer, 171–194.

[13] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying graphs with data. *Journal of the ACM* 63, 2 (2016), 14:1–14:53.

[14] Artem Lysenko, Irina A. Roznovat, Mansoor Saqi, Alexander Mazein, Christopher J. Rawlings, and Charles Auffray. 2016. Representing and querying disease networks using graph databases. *BioData Mining* 9, 1 (25 Jul 2016), 23.

[15] József Marton, Gábor Szárnyas, and Márton Búr. 2017. Model-Driven Engineering of an OpenCypher Engine: Using Graph Queries to Compile Graph Queries. In *SDL Forum (Lecture Notes in Computer Science)*, Vol. 10567. Springer, 80–98.

[16] József Marton, Gábor Szárnyas, and Dániel Varró. 2017. *Formalising openCypher Graph Queries in Relational Algebra.*

[17] Microsoft. 2017. CosmosDB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. (accessed: Dec. 2017).

[18] Joseph Mullen, Simon J. Cockell, Peter Woollard, and Anil Wipat. 2016. An Integrated Data Driven Approach to Drug Repositioning Using Gene-Disease Associations. *PLOS ONE* 11, 5 (05 2016), 1–24.

[19] Neo4j. 2017. Cypher-for-Apache-Spark. https://github.com/opencypher/cypher-for-apache-spark. (accessed: Dec. 2017).

[20] Neo4j. 2017. Neo4j Database. http://www.neo4j.com/. (accessed: Dec. 2017).

[21] openCypher. 2017. Cypher Query Lang. Ref. https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf. (accessed: Dec. 2017).

[22] openCypher. 2017. opemCypher. http://www.opencypher.org/. (accessed: Dec. 2017).

[23] Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph databases.* O'Reilly Media.

[24] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings.* 403–420.

[25] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Felix Cuadrado, Luis M. Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative Property Graph Queries Without Data Migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES'17)*. 4:1–4:6.

[26] Oskar van Rest. 2017. Graph pattern matching semantics. https://tinyurl.com/oCIM1-patternmatching. (accessed: Dec. 2017).

[27] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*.

[28] Byoung-Ha Yoon, Seon-Kyu Kim, and Seon-Young Kim. 2017. Use of Graph Database for the Integration of Heterogeneous Biological Data. *Genomics & informatics* (03 2017), 19–27.

[29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*.