# Updating Graph Databases with Cypher

Alastair Green
Neo4j
alastair.green@neo4j.com

Paolo Guagliardo
University of Edinburgh
paolo.guagliardo@ed.ac.uk

Leonid Libkin
University of Edinburgh
libkin@inf.ed.ac.uk

Tobias Lindaaker
Neo4j
tobias.lindaaker@neo4j.com

Victor Marsault*
LIGM, UPEM/ESIEE-Paris/ENPC/CNRS
victor.marsault@u-pem.fr

Stefan Plantikow
Neo4j
stefan.plantikow@neo4j.com

Martin Schuster*
Abbott Informatics

Petra Selmer
Neo4j
petra.selmer@neo4j.com

Hannes Voigt
Neo4j
hannes.voigt@neo4j.com

## ABSTRACT

The paper describes the present and the future of graph updates in Cypher, the language of the Neo4j property graph database and several other products. Update features include those with clear analogs in relational databases, as well as those that do not correspond to any relational operators. Moreover, unlike SQL, Cypher updates can be arbitrarily intertwined with querying clauses. After presenting the current state of update features, we point out their shortcomings, most notably violations of atomicity and nondeterministic behavior of updates. These have not been previously known in the Cypher community. We then describe the industry-academia collaboration on designing a revised set of Cypher update operations. Based on discovered shortcomings of update features, a number of possible solutions were devised. They were presented to key Cypher users, who were given the opportunity to comment on how update features are used in real life, and on their preferences for proposed fixes. As the result of the consultation, a new set of update operations for Cypher were designed. Those led to a streamlined syntax, and eliminated the unexpected and problematic behavior that original Cypher updates exhibited.

**PVLDB Reference Format:**
Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stephan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. Updating Graph Databases with Cypher. *PVLDB*, 12(12): 2242-2253, 2019.
DOI: https://doi.org/10.14778/3352063.3352139

*Affiliated with the School of Informatics at the University of Edinburgh during the time of contributing to this work.


*Proceedings of the VLDB Endowment,* Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352139

## 1. INTRODUCTION

Updating relational databases is a well understood subject. Tuples can be inserted into a table, deleted from a table, or values of some attributes of existing tuples can be modified. Update statements that insert, delete, and modify tuples are part of the SQL Standard. These statements are separate from SQL queries; once a database is updated and is in a consistent state, new queries can be asked about it.

Our understanding of updates in the popular graph database model, however, is still very rudimentary. The complexity of the graph model that combines stored data with the graph topology allows for rather complex update operations that can be intertwined with graph queries. Here we present the story of update features in Neo4j [21], one of the most popular property graph databases[1]. The query language of Neo4j is called Cypher. It was inspired by the developments of SQL, XPath and SPARQL. After its invention at Neo4j in 2011, it has evolved to support a comprehensive feature set based on extensive graph pattern matching capabilities. Its implementations are not limited to the Neo4j product, and include other products such as SAP HANA Graph[19], RedisGraph[20], Agens Graph (over PostgreSQL)[8] and Memgraph[4], as well as open-source projects, such as Cypher for Apache Spark [1] and Cypher over Gremlin [2]. Cypher is used in hundreds of production applications across many industry vertical domains, such as financial services, telecommunications, manufacturing and retail, logistics, government and healthcare.

While the design of Cypher was influenced by SQL, the way in which the language operates is rather different from SQL. A statement of Cypher is a sequence of clauses, with next clause taking as its input the result of the previous one. Moreover, static and dynamic features are mixed freely: one can start with a pattern matching clause, then update a database based on the matched patterns, and then proceed with another pattern matching clause on the updated database – all in one single statement.

When it comes to dynamic aspects of the language, the facilities it provides are of two kinds. First, there are analogs of the usual insertion, deletion, and modification clauses of

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

SQL, though more complex ones, to reflect the fact that both graph topology and stored data can be updated. Then there are facilities that reflect more recent additions to the SQL Standard, that mix updates and insertions in a single statement, but again they do it in a more complicated fashion reflecting the datamodel. Specifically, some pattern matching clauses are designed in a way that they never fail: if a match is not found, the database is updated in a way that creates a match. One of many possible uses of such a facility is to populate a database with nodes and relationships: those can be read from a file, and added to the database, but only if they do not already exist.

The goal of this paper is to tell the story of dynamic aspects of Cypher and Neo4j. However, we do not only describe graph database updates. Instead, we explain what they are, what the main issues with the current state of affairs are, how to fix problems, and what future update features will be. In more detail, we do the following.

- We first explain the current state of affairs. Update features were added to Cypher in a rather ad hoc manner, to address specific users' needs. We survey them, and explain a number of issues caused by the update facilities of Cypher 9, the current release of the language.
- We then describe how these issues were resolved in a process that involved a collaboration between Neo4j, its academic partner (the database group at the University of Edinburgh), and users of the Neo4j product. A number of alternative solutions were proposed by the academic team. They were presented to many users of the Neo4j product (including participants of the openCypher Implementers Meeting, as well as Field Engineering and Developer Relations Teams of Neo4j), who were invited to comment on the proposals and the real-life usage of update operators. Their answers were then used to choose possible solutions for solving problems with current update facilities.
- We present the outcome of this consultation, describing new update features of that will be available in the future releases of Cypher. These could be useful for other existing property graph query languages [23, 7], and could inform the design of new ones.

*Organization.* In Section 2 we introduce the property graph model and Cypher by example. In Section 3 we describe updates as they currently are in Neo4j's implementation of Cypher. Section 4 explains the main issues with the current state of Cypher's dynamic features. In Section 6 we outline various proposals for fixing these problems. Section 7 presents the decisions about new update features based on interactions with Cypher's user base. In Section 8 we outline the formal semantics of new Cypher updates. Section 9 provides a final summary and outline of the implementation plans in the Neo4j product. Full details of the formal semantics of updates can be found in [12].

## 2. QUERYING GRAPHS WITH CYPHER

We give a brief introduction into the property graph model and the Cypher query language, by means of examples.

Unlike the relational model, the graph data model stores information in the form of a *graph* that consists of *nodes* and *relationships* between those nodes. Nodes usually model real-world entities, while relationships model connections between those entities. Additional information about the entities and the connections between them is stored in both nodes and relationships in the form of *node labels*, *relationship types* and (node or relationship) *properties*.

As a running example, consider the back-end of an online marketplace where users can buy products that are offered either by professional vendors or by other users. In this example, nodes of the graph database represent users, vendors and products, and relationships detail which products were ordered by which users, and which users or vendors offer which products. A very small property graph of this form is given by the solid lines in Figure 1.

The basic properties of the property graph model are as follows.

- Each node may have an arbitrary number of *labels*, denoted by a leading colon; for example, nodes $p_1$, $p_2$ and $p_3$ all have the label `:Product`. Node labels are optional, i.e. there may be unlabeled nodes.
- Each relationship has precisely one *source node*, *target node*, and *type*, with the latter denoted by a colon; for instance, the relationship between nodes $u_1$ and $p_1$ is of type `:ORDERED`. The uniqueness of source and target nodes means that relationships are always *directed* from source to target, and there may never be any "dangling relationships" (i.e. relationships with a missing source or target). There may, however, be multiple relationships with identical types and properties between the same nodes.
- Both nodes and relationships may have an optional *property map*, given as a set of key-value pairs. In the example, all nodes have `id` and `name` properties, while relationships do not have any properties; in general, property keys can also vary between nodes (or relationships, respectively).

We next describe the basic features of the Cypher language, the query language of the Neo4j graph database product. The querying fragment of the Cypher language for property graphs uses an "ASCII art" representation of nodes and relationships. For instance, the query

```
MATCH (p:Product)<-[:OFFERS]-(v:Vendor)
        -[:OFFERS]->(q:Product)
WHERE p.name = "laptop"
RETURN v
```

can be used to find all vendors offering two products, one of which has the name `"laptop"`. We use this query for a very brief introduction to the read-only fragment of Cypher; for a more comprehensive overview, see e.g. [21, 13].

At their very core, Cypher queries consist of a sequence of *clauses* that are evaluated on a property graph and manipulate a *driving table*. In the context of Cypher, tables are bags, or multisets, of consistent *records*, i.e., of key-value maps with the same set of keys.

In the example, we start with an empty table, which the first `MATCH` clause populates with two records (`p`:$p_1$, `v`:$v_1$, `q`:$p_2$) and (`p`:$p_2$, `v`:$v_1$, `q`:$p_1$). Indeed, replacing variables `p`, `v`, and `q` by the nodes given for them in each of the records gives a match for the input pattern in the graph of Figure 1.

Readers experienced in SQL may wonder why the variables `p` and `q` cannot be matched to the *same* node (which

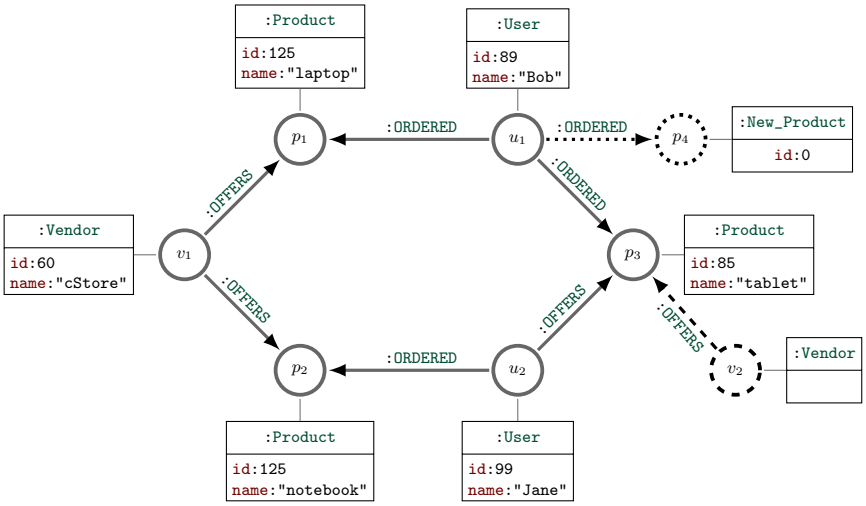



**Figure 1:** Property graph for the running examples of Sections 2 and 3. Solid lines indicate the initial graph; dotted lines – the additions from Query 1; dashed lines – the additions from Query 4.

would yield two additional records); this is due to a feature in the pattern matching of Cypher requiring that distinct relationship patterns in a graph pattern have to be mapped to distinct relationships in the graph. This ensures that outputs of queries are finite. For example, consider a graph containing a single loop on node $v$. If the same edge can be traversed multiple times, it is not clear how many times the node $v$ will be returned by the query `MATCH (v) -[*]-> (v) RETURN v` that allows an arbitrary number of edges on the path from $v$ to $v$. Cypher eliminates this ambiguity by disallowing multiple edge traversals in pattern matching; this, however, is not without complexity costs, see [13].

Next, the `WHERE` clause acts as a filter on the driving table according to the condition given in the clause. In the example, this would remove the record (`p`:$p_2$, `v`:$v_1$, `q`:$p_1$) from the driving table, as the `name` property of node $p_2$ is different from `"laptop"`.

Finally, the `RETURN` clause projects all records of the driving table to the variable(s) given in the clause; in the example, we would end up with a final table containing only the record (`v`:$v_1$). Note that tables are bags, i.e. without the `WHERE` clause removing one record from the intermediary driving table, the final table would have contained *two* copies of the record (`v`:$v_1$).

Following this intuition, previous work in [13] formalized the semantics of a Cypher clause $C$, given a property graph $G$, as a mapping $[\![C]\!]_G$ from tables to tables. That is, each clause takes as input a driving table and produces from it an output driving table. The semantics of a Cypher query can then simply be derived from the semantics of its clauses by composing their corresponding mappings from left to right, and the output of a query is computed by feeding the empty table to its semantic mapping as an input. Based on this idea, a full formal semantics for the read-only fragment of Cypher was given in [13]. This was part of an ongoing effort to provide a formal language specification as part of creating a standard for Cypher. Similar efforts for SQL have a long history [9, 18] and are still ongoing [11, 14].

While the read-only fragment of Cypher, along with its formalization, are relatively straightforward, it turns out that updates to property graphs present additional challenges compared to the way that updates are handled in the relational model. In particular, Cypher's modular nature allows for chaining arbitrary sequences of reading and writing clauses in one single query, which raises questions regarding atomicity of statements and transaction boundaries. In the next section we present a brief overview of Cypher updates, before outlining some of the most significant issues that the current state of the art leads to.

## 3. UPDATES IN CYPHER 9

We now describe the state of update facilities in Cypher 9, the current release of the language in Neo4j. The main syntactic constructs are shown in Figures 2–5; we do not go into the details, as most of the syntax is self-explanatory and in line with the syntax of read-only queries (see [12, 13]). What is worth noting are the different types of restricted patterns for updates and the somewhat involved structure of queries containing both reading and update clauses; we shall discuss these points in more detail in Section 4.4. Here, in the same spirit of the previous section, we introduce Cypher's update facilities by means of a few easy to follow examples.

The `CREATE` clause in Cypher (similar to the `INSERT` clause in SQL) is used to insert new nodes or relationships into the property graph. Similar to the `MATCH` clause, it uses an "ASCII-art" style of representing nodes and relationships and can insert entire paths. As a simple example, the following would insert a new product ordered by user 89:

Query 1

```
MATCH (u:User{id:89})
CREATE (u)-[:ORDERED]->(:New_Product{id:0})
```

In the example in Figure 1, this query would insert the dotted node $p_4$ and attached relationship.

Specifying node labels as well as properties for nodes or relationships in newly created patterns is optional; relationship types, on the other hand, have to be specified in order to ensure that every relationship has a unique type.

Similar to the `UPDATE` clause in SQL, the clauses `SET` and `REMOVE` in Cypher 9 are used to manipulate node labels and

⟨query⟩ ::= ⟨clause sequence⟩ [ UNION [ALL]? ⟨query⟩ ]?

⟨clause sequence⟩ ::= [ ⟨reading clause⟩ ]* ⟨return⟩ | ⟨reading clause⟩* ⟨update clause⟩+ [ ⟨with⟩ ⟨clause sequence⟩ ]?

**Figure 2:** Syntax of queries. The derivation rules for ⟨*reading clause*⟩, ⟨*with*⟩ and ⟨*return*⟩ are omitted; see [12, 13] for details.

⟨update clause⟩ ::= ⟨set⟩ | ⟨remove⟩ | ⟨create⟩ | ⟨delete⟩ | ⟨merge⟩ | ⟨for each⟩

⟨set⟩ ::= SET ⟨set item⟩ [ , ⟨set item⟩ ]*

⟨remove⟩ ::= REMOVE ⟨rem. item⟩ [ , ⟨rem. item⟩ ]*

⟨create⟩ ::= CREATE ⟨dir. upd. pat.⟩ [ , ⟨dir. upd. pat.⟩ ]*

⟨delete⟩ ::= DELETE ⟨expr⟩ [ , ⟨expr⟩ ]*

⟨merge⟩ ::= MERGE ⟨upd. pat.⟩

⟨for each⟩ ::= FOREACH ( ⟨name⟩ IN ⟨expr⟩ | ⟨update clause⟩ )

**Figure 3:** Syntax of update clauses. The derivation rules for ⟨*expr*⟩ and ⟨*type*⟩ are omitted.

⟨set item⟩ ::= ⟨expr⟩ = ⟨expr⟩ | ⟨expr⟩ += ⟨expr⟩ | ⟨expr⟩ ⟨label list⟩

⟨rem. item⟩ ::= ⟨expr⟩ . ⟨key⟩ | ⟨expr⟩ ⟨label list⟩ ; ⟨label list⟩ ::= : ⟨label⟩ [ : ⟨label⟩ ]*

**Figure 4:** Syntax of clause items used by REMOVE and CREATE. The derivation rules for ⟨*label*⟩ and ⟨*key*⟩ are omitted.

⟨upd. pat.⟩ ::= [ ⟨name⟩ = ]? ⟨node pat.⟩ [ ⟨rel. upd. pat.⟩ ⟨node pat.⟩ ]*

⟨dir. upd. pat.⟩ ::= [ ⟨name⟩ = ]? ⟨node pat.⟩ [ ⟨dir. rel. upd. pat.⟩ ⟨node pat.⟩ ]*

⟨node pat.⟩ ::= ( ⟨name⟩? ⟨label list⟩? ⟨map⟩? )

⟨rel. upd. pat.⟩ ::= [ < ]? -[ ⟨name⟩? :⟨type⟩ ⟨map⟩? ]-[ > ]?

⟨dir. rel. upd. pat.⟩ ::= -[ ⟨name⟩? :⟨type⟩ ⟨map⟩? ]-> | <-[ ⟨name⟩? :⟨type⟩ ⟨map⟩? ]-

**Figure 5:** Syntax of the patterns that appear in the MERGE and CREATE clauses. The derivation rules for ⟨*map*⟩ and ⟨*type*⟩ are omitted.

property maps on both nodes and relationships. For instance, the following query would change the id property, add a new name property and replace the label :New_Product with :Product on the newly created node from Query 1:

Query 2

```
MATCH (p:New_Product{id:0})
SET p:Product, p.id=120,p.name="smartphone"
REMOVE p:New_Product
```

The DELETE clause is used to remove nodes and relationships from a property graph, with one subtlety: As property graphs may contain no "dangling relationships" (i.e. each relationship must have a source node and target node), any DELETE clause supposed to delete nodes must also ensure that those nodes do not have any relationships attached to them. For instance, in the running example, after execution of Queries 1 and 2, the query

```
MATCH (p:Product{id:120})
DELETE p
```

would fail, because the :Product node with id 120 (node $p_4$) is the source node of an :ORDERED relationship. In order to delete this newly inserted node, we would also have to match and delete its attached relationship:

```
MATCH ()-[r]->(p:Product{id:120})
DELETE r,p
```

Alternatively, the DETACH DELETE clause can be used to delete nodes along with all relationships attached to them:

Query 3

```
MATCH (p:Product{id:120})
DETACH DELETE p
```

Due to the compositional nature of Cypher, Queries 1, 2 and 3 can be intertwined as in the following example (given for purely illustrative purposes) that creates a new node, changes its properties and then deletes it again:

```
MATCH (u:User{id:89})
CREATE (u)-[:ORDERED]->(p:New_Product{id:0})
SET p:Product,p.id=120,p.name="phone"
REMOVE p:New_Product
DETACH DELETE p
```

Finally, Cypher 9 offers a hybrid query/update clause MERGE, that borrows the idea from the merge statement present in recent versions of the SQL Standard, but expands it with the full power of Cypher pattern matching. On specification of an input pattern, MERGE tries to find a match of that pattern (like the MATCH clause), and where this match fails, MERGE works as an update and creates a new instance of the pattern (like the CREATE clause).

As an example, consider the following query, which specifies that each product in the graph should be offered by

some vendor and returns the (matched or newly created) product/vendor pairs:

QUERY 4

```
MATCH (p:Product)
MERGE (p)<-[:OFFERS]-(v:Vendor)
RETURN p,v
```

In the example graph of Figure 1, the first line would match the three :Product nodes $p_1$, $p_2$ and $p_3$. For the former two, the MERGE clause finds a match (in the :Vendor node $v_1$), while for the latter, it does not find a match and therefore creates a new :Vendor node $v_2$, and an :OFFERS relationship shown by dashed lines. Finally, the RETURN clause returns the two :Product $p_1$, $p_2$, each paired with the :Vendor node $v_1$, and the :Product node $p_3$, paired with the newly created :Vendor node $v_2$.

# 4. PROBLEMS WITH UPDATES

While the update operations described in the previous section may seem clear and simple on the surface, they do in fact have some problems that appear when one takes a closer look. In this section, we discuss how the more complex structure of the graphs and tables that Cypher operates on (compared to the relational tables for SQL) leads to violations of some very basic properties one would expect from update operations in databases. In particular, we will see that updates violate the basic principle of *atomicity*. Intuitively, the reason behind this is that updates appear in a statement where they are preceded by other clauses, and thus they operate over the driving table, going over it tuple by tuple (in a way similar to `for each row` triggers). While doing so, such updates can see intermediate results, i.e., those that were the result of update operations looking at earlier rows in the driving table. In addition to violating atomicity, some updates may be *nondeterministic*. Indeed, the order in which they process the driving table, and the lack of atomicity, may affect the end result. We now illustrate these issues by means of examples.

## 4.1 Problems with SET

In this section, we show how the behavior of SET in the previous Neo4j implementation may be both non-atomic and nondeterministic.

*Example 1.* For this example, we assume that the property graph has already been populated, but due to an error in data entry, the product ID numbers for the products "laptop" and "tablet" have been switched.

In order to avoid manual fetching and setting of product ID numbers, experienced SQL programmers might expect the following query to work:

```
MATCH (p1:Product{name:"laptop"}),
      (p2:Product{name:"tablet"})
SET p1.id = p2.id, p2.id = p1.id
```

In the current implementation, however, this query behaves the same as the following:

```
MATCH (p1:Product{name:"laptop"}),
      (p2:Product{name:"tablet"})
SET p1.id = p2.id
SET p2.id = p1.id
```

In other words, the current semantics would *first* set the ID of "laptop" to be the same as that of "tablet", after which both products bear the same ID number, and *then* perform a "no-operation" by setting the ID of "tablet" to that of "laptop", at a point where both IDs are identical. Enforcing the originally desired switch of the two ID properties, on the other hand, is impossible without using auxiliary variables in the current implementation.

*Example 2.* This example is very similar to Example 1; here, we want to set the name of a product with ID number 85 to be the same as the name of a product with ID 125. This is easily done by the following query:

```
MATCH (p1:Product{id:85}),(p2:Product{id:125})
SET p1.name = p2.name
```

Let us assume, however, that due to some error in data entry, or dirty input data, the graph actually contains more than one node with label :Product and ID number 125; this situation is exemplified in Figure 1, where nodes $p_1$ and $p_2$ both have ID number 125, but different name properties. In this case, it is unclear what result we would even expect from the execution of the above query, and depending on the order in which nodes are matched, node $p_3$ might end up with name set to either "notebook" or "laptop".

## 4.2 Problems with DELETE

In the current implementation of Cypher, the DELETE clause violates atomicity. It is possible, for instance, to set properties of deleted nodes. The following query (rather artificial, given for illustration purposes only) goes through without an error and returns an empty node without any labels or properties.

```
MATCH (user)-[order:ORDERED]->(product)
DELETE user
SET user.id = 999
DELETE order
RETURN user
```

As a minor issue, this raises the question of how deleted entities may be manipulated after they have been deleted. In particular, one wonders what happens if a deleted entity is to be returned. Much more seriously, however, allowing this query to go through without an error creates an intermediate state (after the first DELETE clause) where the current working graph is in an illegal state – there are "dangling relationships" without a start node. Complex data querying may actually be executed on this illegal graph (e.g., MATCH clauses), with an unclear meaning.

## 4.3 Problems with MERGE

In this section, we demonstrate issues with the current behavior of MERGE. As a high-level overview, the current implementation of MERGE works (as intended) in a "match-or-create" fashion on a per-record base, that is, for each record in the driving table, it tries to match an instance of the given pattern in the current graph, and, if unsuccessful, creates an instance in the graph.

While processing an update record-by-record is similar to what SQL does, the main issue is that the current implementation of Cypher allows MERGE to read its own writes. That is, if processing a record in the driving table creates some nodes and relationships, these may be matched during the processing of subsequent records of the driving table

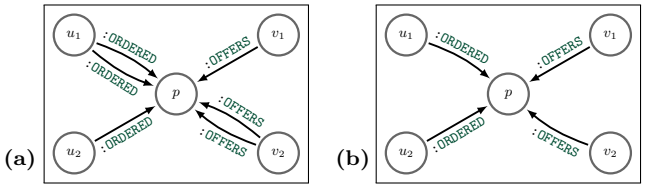



**Figure 6:** Two possible output graphs in Example 3.

within the *same* MERGE clause. Not only does this behavior violate the desired atomicity of clauses, but it may also lead to cases where the output graph of a MERGE clause depends on the order in which records in the driving table are processed. Since tables in Cypher are assumed to be unordered and may be reordered at will by the query-processing engine, this means that the behavior of a MERGE clause may be nondeterministic. For an illustration, consider the following example.

*Example 3.* In this example, we demonstrate how the behavior of MERGE may be nondeterministic in the current Neo4j implementation. For this example, we assume that the input graph contains only nodes and no relationships, and that the input table is already populated. This assumption is reasonable, as the feedback from the user base tells us that MERGE is often used to populate a graph based on a table that has been produced by importing from a relational database or a CSV file, and that it is a common practice to input nodes first and relationships later.

The query contains the following MERGE clause:

QUERY 5

```
MERGE (user)-[:ORDERED]->(product)
            <-[:OFFERS]-(vendor)
```

The associated driving table is the following:

| user | product | vendor |
|---|---|---|
| $u_1$ | $p$ | $v_1$ |
| $u_2$ | $p$ | $v_2$ |
| $u_1$ | $p$ | $v_2$ |

Here, $u_1$, $u_2$, $p$, $v_1$ and $v_2$ stand for nodes that are already present in the graph, and we assume that these nodes do not have any relationships yet.

The execution of Query 5 may produce two different graphs depending on the evaluation order. Indeed, going through the driving table bottom-up yields the graph in Figures 6a, since none of the expected paths of length 2 can be matched. On the other hand, evaluating the driving table top-down yields the graph in Figure 6b, since the path $u_1 \to p \to v_2$ can be matched after creating the paths $u_1 \to p \to v_1$ and $u_2 \to p \to v_2$ .

## 4.4 Problems with the syntax

The full syntax for updates in Cypher 9 is somewhat lengthy and cumbersome. This is due to two major factors.

First, the current implementation in Neo4j allows for reading clauses (such as MATCH and WHERE) to be followed by a sequence of update clauses. However, if a sequence of update clauses is followed by reading clauses, a WITH clause is required in between. While this rule is not strictly necessary and exist for historic reasons, it turns WITH clause into a clear demarcation line marking when effects of update clauses become visible to reading clauses of the same query.

Second, the CREATE and MERGE clauses, though similar in behavior, allow for different types of patterns to be specified in their input. For CREATE, tuples of path patterns may be given, but relationship patterns must carry a specified direction, whereas MERGE only allows for a single path pattern, where relationship patterns may also be undirected.

# 5. INPUT FROM NEO4J USERS

In order to come up with the redesign of update features that would not only eliminate problems outlined in the previous section but would also satisfy the needs of Neo4j users, it was decided to consult the user base to understand how update features are used, and what is expected of them in practice. Although there was a recent survey of users of graph databases [22] that revealed much useful information, it was not specific enough to address the usage of a particular language features, as we need here.

We have consulted three groups of users.

- The first group consisted of participants of the openCypher Implementers Meeting (oCIM). This is the meeting of the openCypher project [5] that aims to deliver a full and open specification of the language; it is attended by implementers of the language from several companies and research projects.
- The second group consisted of members of the Neo4j Field Engineering Team. They are responsible for helping customers deliver enterprise-level solutions using Neo4j, troubleshooting issues with applications in production environments, and exploring proof-of-concept ideas.
- The third group involved the Neo4j Developer Relations Team. They are responsible for interacting with and growing the developer community, and user training and support. Their members oversaw multiple training sessions that involved about 500 Cypher programmers (thus significantly exceeding the number of users who took the survey of [22]).

The consultation with openCypher implementers took place during the 4th oCIM in May 2018. It concentrated primarily on enforcing atomicity and determinism of updates in Cypher updates. Upon the presentation of issues detailed in the previous section, the proposal to enforce atomicity of SET and DELETE received overwhelming support of participants.

However, Cypher implementers do not necessarily see how updates are used in practice. While changes to SET and DELETE are fairly straightforward, it is not so clear how MERGE needs to be modified; indeed we shall see soon that there are many different options. Thus, to choose them, it was important to know how MERGE is actually used, and what difficulties Cypher programmers experience with it. For this, we undertook a survey within Neo4j in which we solicited feedback from members of the Field Engineering the Developer Relations Teams.

Most importantly, the survey revealed that users intuitively think that the behavior of MERGE is to create the

missing parts of a specified pattern, after first matching as much of the pattern as possible. The notion that it operates on the *entire* pattern is counter-intuitive. Thus, by far the most prevalent error caused by misunderstanding MERGE is the unintended creation of duplicate nodes and relationships. The majority of the survey participants observed on multiple occasions errors caused by this misunderstanding of MERGE.

The prevailing initial response to MERGE by new users in training sessions is one of surprise when it comes to merging patterns, and following that, some concern about how to determine when they have made an error. They tend to expect that the missing part of the pattern will be merged into an existing pattern. On the other hand, the idea of merging nodes – as opposed to patterns – does not cause problems. Over half of the survey respondents said that at least 60% of users at training sessions struggle with understanding MERGE when it is presented to them for the first time.

A clear majority of the survey respondents were of the opinion that the MERGE clause ought to be revised in the following ways.

(i) Its semantics must be deterministic and atomic, and it should be clear what precisely is being merged on, and what is expected to be unique.

(ii) The term itself and the syntax of the clause ought to be amended, to be closer in line with users' understanding of the semantics.

In the next section, we explain how to turn this wish list into concrete proposals for new update features of Cypher.

# 6. PROPOSALS FOR NEW CYPHER

Given the problems presented in Section 4 and Cypher users' wishes outlined in Section 5, we now describe proposals for changing syntax and semantics of Cypher updates.

To start with, it is fairly clear how the problems with the SET and DELETE clauses can be solved. Those statements must enforce atomicity, i.e., they cannot see partial results of deletions or changing of values. Indeed, these clauses have their direct SQL counterparts (although they add more complex pattern matching machinery), and thus we can adopt well known solutions from the relational world.

The situation with the MERGE clause, on the other hand, is more complicated. While it is clear that the users have two primary wishes – atomicity and clarity with respect to what is being merged on – there could be multiple ways of achieving these.

As already pointed out in Example 3, the cause of the nondeterministic behavior of the current implementation of MERGE is its ability to read its own writes. Any proposal to fix this would therefore have to separate the reading (MATCH-like) parts of MERGE from its writing (CREATE-like) parts. The easiest way of doing so would be the following.

*Atomic* MERGE. It creates a copy of the input pattern for each record in the driving table for which the input pattern could not be matched in the original input graph. Once all these input patterns are created, the graph is modified in a single atomic step, by adding all of them.

This version is very natural but at the same time is does not address the intuition that MERGE should transform the input graph with "minimal changes" into one where the input pattern can be matched for each record in the driving table – and this is indeed something that Neo4j users felt strongly about. To address this, the next logical step is to still to find all the matches in the original graph, but perform all the writing in a temporary *change graph*, which then gets minimized by collapsing similar nodes and relationships, and afterwards is inserted into the input graph.

However, there are multiple ways in which similarity of nodes and relationships can be defined, for the purpose of collapsing them. For Atomic MERGE, no minimization is performed at all. We now describe four ways of meaningfully collapsing nodes and edges, and then illustrate them by means of examples.

*Grouping* MERGE. The minimization is done by grouping records in the driving table by the expressions appearing in the pattern and then creating only one copy for each group. That is, not only do duplicates get eliminated, but irrelevant entries are disregarded.

*Weak Collapse* MERGE. In addition to the grouping performed as above, all newly created nodes that have the same labels and properties and are matched to the same position of the input pattern in MERGE get collapsed into one; similarly, all relationships that have the same types, properties, source and target nodes (after node collapsing) and occur in the same position of the input pattern get collapsed into one.

*Collapse* MERGE. It behaves like Weak Collapse MERGE, except that it allows to collapse nodes that match *different* positions of the input pattern.

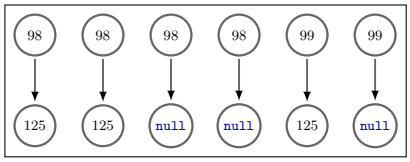
*Strong Collapse* MERGE. It lifts the restriction from Collapse MERGE in that it also collapses relationships with identical types, properties, source and target nodes (after collapsing) that match *different* positions in the input pattern. This behavior lifts the restriction that patterns are matched by traversing each edge at most once. Indeed, after collapsing relationships, it is possible that the pattern in MERGE is satisfied by traversing a relationship more than once.

We now give several examples illustrating the different possible semantics proposed for MERGE. Throughout these examples, we often assume that the input graph is empty, while the input table is already populated. As explained in the previous section, this assumption is not artificial: it reflects the way in which a graph database may be initially populated by importing data from a relational database or a CSV file.

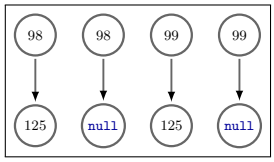
First, one may see that these new approaches resolve the issue of nondeterministic behavior in Example 3.

*Example 4.* We consider the setting of Example 3 and describe how the different proposed MERGE variants behave.

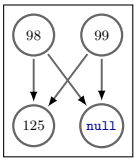
All of them avoid nondeterminism. Atomic or Grouping semantics always yield the graph of Figure 6a, that is, create the path of the input pattern for all three records in the input table. All three variants of collapse MERGE create the minimal graph (Figure 6b), as the relationships between $u_1$ and $p$, and between $p$ and $v_2$ have identical properties and are matched by the same positions of the pattern in MERGE.




**(a)** Atomic




**(b)** Grouping




**(c)** (Weak/Strong) Collapse

**Figure 7:** Resulting graphs for different semantics of `MERGE` in Example 5. Top-row nodes are labeled `:User`, bottom-row nodes are labeled `:Product`, relationships are all of type `:ORDERED`, and values inside nodes denote values of their `id` attribute.

*Example 5.* In this example, we contrast the different proposed semantics options for `MERGE` in some more detail. In particular, we demonstrate how they handle duplicates, node collapsing, and `null` values. The query in this scenario is as follows:

```
MERGE (:User{id:cid})
      -[:ORDERED]->(:Product{id:pid})
```

Assume that the input graph is empty, and consider the following driving table:

| cid | pid | date |
|---|---|---|
| 98 | 125 | 2018-06-23 |
| 98 | 125 | 2018-07-06 |
| 98 | null | null |
| 98 | null | null |
| 99 | 125 | 2018-03-11 |
| 99 | null | null |

Here, both atomic and grouping `MERGE` may create multiple nodes for the same user ID if a user has multiple orders. The main difference is that, if the same user and product ID appear together in several records (possibly with different order dates), atomic `MERGE` will create the corresponding nodes multiple times while grouping `MERGE` only creates one pair of nodes for each unique pair of cid/pid (regardless of additional columns in the driving table). If the driving table contains any `null` values, these are treated the same way as regular IDs, i.e. a new node is created for each order with a `null` ID (unless, in the case of grouping `MERGE`, another `null` order associated with the same customer already exists).

Concretely, for the example table, Atomic `MERGE` will create the graph with twelve nodes and six relationships in Figure 7a, while Grouping `MERGE` eliminates duplicate cid/pid pairs and creates only the eight-node graph in Figure 7b.

All three versions of collapse `MERGE` show identical behavior in this example, creating only a single product node for each pid, a single customer node for each cid and a single relationship for each unique cid/pid pair. If the driving table contains any `null`s, e.g. for pid, only a *single* corresponding node with `null` product ID is created, and *all* orders associated with `null` product IDs in the driving table are linked to this "non-product" by an `:ORDERED`-relationship. With the given table, this yields the graph shown in Figure 7c.

*Example 6.* This example highlights the difference between Weak Collapse `MERGE` and (Strong) Collapse `MERGE`. The scenario is like the one given for Example 3, except for the fact that this time we want to insert information about sales between two users instead of sales between a user and a vendor. Consider the query

```
MERGE (:User{id:bid})
      -[:ORDERED]->(:Product{id:pid})
      <-[:OFFERS]-(:User{id:sid})
```

with the following driving table:

| bid | pid | sid |
|---|---|---|
| 98 | 125 | 97 |
| 99 | 85 | 98 |

This query, executed on an empty graph, yields the results displayed in Figure 8, depending on the semantics chosen.

As can be seen in that figure, Collapse and Strong Collapse `MERGE` actually allow for combining the two copies of the `:User` node with ID 98, as one would expect.

*Example 7.* We finally highlight the difference between the behavior of the Collapse and Strong Collapse semantics. Let us consider the following driving table.

| a | b | c | d | e | tgt |
|---|---|---|---|---|---|
| $p_1$ | $p_2$ | $p_3$ | $p_1$ | $p_2$ | $p_4$ |

All values in the table are node ids that represent products previously looked up in the graph. It tracks the last product pages visited by customer (`a`–`e`) before making a purchase (`tgt`). Here, the customer visited, in that order, the pages of the products $p_1$, $p_2$, $p_3$, then once again the pages of the products $p_1$ and $p_2$, and finally added product $p_4$ to the cart.

The Cypher statement below incorporates this search-and-purchase query into the graph:

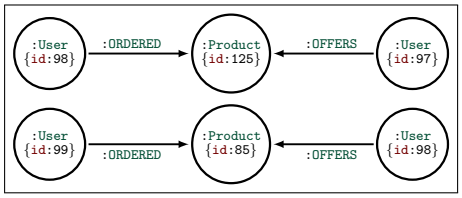
```
MERGE (a)-[:TO]->(b)-[:TO]->(c)-[:TO]->(d)
      -[:TO]->(e)-[:BOUGHT]->(tgt)
```

The different resulting graphs, depending on the chosen semantics, are shown in Figure 9.

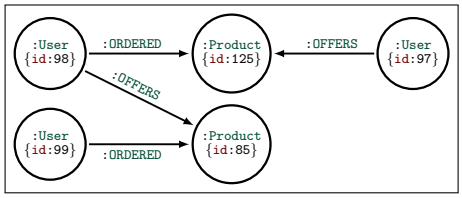
Recall that the Strong Collapse semantics differs from the semantics of `MATCH` in that separate relationship patterns in the input pattern do *not* have to be matched to separate relationships in the graph. This is illustrated by our example. In fact, if after executing the above `MERGE`, one tries to match the added pattern with

```
MATCH (a)-[:TO]->(b)-[:TO]->(c)-[:TO]->(d)
      -[:TO]->(e)-[:BOUGHT]->(tgt)
```

the query would return *no* matches in the resulting graph of the above `MERGE` with Strong Collapse semantics. This is due to collapsing the two `:TO` edges from $p_1$ to $p_2$, making it impossible to match the above pattern under the single-edge-traversal semantics of Cypher pattern matching. However, if instead of the current Cypher matching semantics




**(a)** Atomic, Grouping, or Weak Collapse




**(b)** Collapse or Strong Collapse

**Figure 8:** Resulting graphs for different semantics of `MERGE` in Example 6.

one would use matching based on graph homomorphisms, then for each of the above versions of merge, first merging a pattern and then matching it will result in a positive match. It is planned that subsequent versions of Cypher will offer more flexibility in terms of pattern matching semantics, including homomorphism-based (with suitable restrictions to guarantee finite outputs). For them, Strong Collapse will be a very natural choice.

# 7. DECISIONS ON NEW CYPHER

We now explain the decisions made by Neo4j in response to the consultation with the users, and various proposals on fixing problems with update facilities in Cypher 9. These decisions have or will be proposed to the openCypher Implementers Meeting where the design for future versions of Cypher is ultimately decided. It will also be used as input for the design of GQL [3].

*Semantics for* `SET`. The decision is that `SET` should be atomic, i.e. all the changes in a single `SET` clause should occur "at the same time", so the query given in Example 1 should actually switch IDs as expected.

In Example 2 on the other hand, there is clearly no "right" output, so any ambiguous `SET` clause like the one given there should abort with an error.

To achieve both of these ends, the `SET` clause is evaluated as follows. First, all the expressions within a `SET` clause are evaluated on the input graph for all the records in the input driving table, to accumulate all the changes to the input graph that would be induced. If these changes are well-defined (i.e. there are no attempts to set some property to two conflicting values as in Example 2), they are then applied to the input graph in order to produce an output graph.

*Semantics for* `DELETE`. Likewise, `DELETE` should be atomic, i.e. that "dangling relationships" should never occur at any time during the processing of a query. Attempting to delete any nodes without deleting, in the same clause, all relationships still attached to them should return an error.

Cypher will follow a *strict semantics* which assumes that all entities deleted within a `DELETE` clause are removed from the input graph as soon as the clause is completely processed, and any reference to a deleted entity in the driving table is replaced by a `null` instead.

*Semantics for* `MERGE`. It was agreed that `MERGE` should be made both deterministic and atomic. Given the options described in the previous section, Neo4j decided to implement the Atomic semantics from the previous section in Cypher for Apache Spark [1] in the form of a new `MERGE ALL` clause and the Stronge Collapse semantics in the form of a new `MERGE SAME` clause, thereby providing the user with a choice between semantics. These two semantics seem to strike a good balance of solving use case requirements in Cypher for Apache Spark, being semantically well defined, and straightforward to implement. The experience of using this implementation will be informative for the evolution of openCypher and GQL.

To illustrate this, using Example 5 with its driving table, the statement

```
MERGE ALL (:User{id:cid})
          -[:ORDERED]->(:Product{id:pid})
```

would produce the graph in Figure 7a, while

```
MERGE SAME (:User{id:cid})
          -[:ORDERED]->(:Product{id:pid})
```

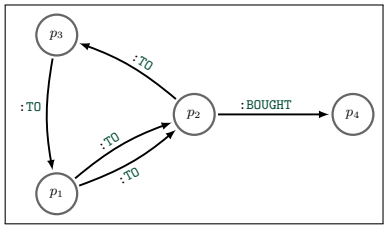
would result in the graph in Figure 7c. The query used in Example 5 (without `ALL` or `SAME`) will no longer be allowed.

*Syntax.* To streamline the syntax, it was decided to drop the requirement of `WITH` clauses between writing and reading clauses, and to treat both simply as clauses. The syntax of `CREATE` and `MERGE` was unified in that both clauses should allow for *tuples* of path patterns with *directed* relationships. The latter change eliminates some of the "`MATCH`-like" capability of `MERGE` to match relationships regardless of direction, but also removes an additional source of non-determinism in the direction of relations to be created. This led to the updated syntax for Cypher displayed in Figure 10; all tokens whose derivation rules are not detailed here refer to those given in Figures 2–5 (or [12, 13], respectively). We note that the syntax tokens ⟨*upd. pat.*⟩ and ⟨*rel. upd. pat.*⟩ are no longer required in the new syntax.

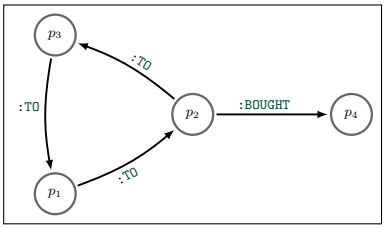
# 8. SPECIFICATION OF UPDATES

Unlike many other languages that are defined by means of a specification in natural language, the read-only core of Cypher has been fully formalized [13, 12], in the same spirit of some programming languages [17, 6, 15, 16], as well as partial specifications of SQL's semantics [10, 24, 14]. In this section we explain how to extend the formal semantics of the read-only core of Cypher to the full language, which includes querying and update features.

Specifically, we provide the formal semantics of the update constructs specified by the `SET`, `DELETE`, `CREATE`, `REMOVE` and `MERGE` clauses. Giving full details of the semantics of all constructs entails significant space requirements; thus, we only




**(a)** Atomic, Grouping, Weak Collapse, or Collapse




**(b)** Strong Collapse

**Figure 9:** Resulting graphs for different semantics of `MERGE` in Example 7.

⟨*clause sequence*⟩ ::= [ ⟨*clause*⟩ ]* [ ⟨*return*⟩ | ⟨*update clause*⟩ ]

⟨*clause*⟩ ::= ⟨*reading clause*⟩ | ⟨*update clause*⟩

⟨*merge*⟩ ::= `MERGE ALL` ⟨*dir. upd. pat.*⟩ [ `,` ⟨*dir. upd. pat.*⟩ ]*

| `MERGE SAME` ⟨*dir. upd. pat.*⟩ [ `,` ⟨*dir. upd. pat.*⟩ ]*

**Figure 10:** Modification of Cypher syntax.

explain the general principles behind the semantics of updates, and concentrate on the most interesting `MERGE` clause. A complete version of the semantics is available in [12].

## 8.1 Principles of the semantics

The formal semantics of the core fragment of Cypher relies on the following relation and function.

- The *pattern matching relation* checks if a path $p$ in a graph $G$ satisfies a pattern $\pi$, under an assignment $u$ of values to the free variables of the pattern. This is written as $(p, G, u) \models \pi$.
- The *semantics of expressions* associating a value $[\![e]\!]_{G,u}$ with an expression $e$, a graph $G$ and an assignment $u$.

While the exact definitions of these can be found in [12], previous examples of Cypher queries should provide the reader with sufficient intuition about pattern matching to follow the rest.

Given a graph $G$, the semantics of a read-only clause $C$ is defined in [12] as a function $[\![C]\!]^{\text{ro}}_G$ that takes a driving table and returns a modified driving table (and similarly for queries consisting of read-only clauses). In order to take into account the dynamical aspects considered in this article, we extend this definition as follows.

- The *semantics of queries* (resp., *clauses*) associates a query $Q$ (resp., clause $C$) with a function $[\![Q]\!]$ (resp., $[\![C]\!]$) that takes as input a graph-table pair $(G, T)$, and returns a graph-table pair $(G', T')$.

Here $G'$ is the updated graph, and $T'$ is the modified driving table.

The semantics of read-only clauses/query is cast in this framework in the natural way: given a read-only clause $C$, a graph $G$ and a table $T$, then we set: $[\![C]\!](G, T) = (G, [\![C]\!]^{\text{ro}}_G(T))$. Semantics of read-only queries is done similarly. In other words, the semantics of read-only clauses and queries does not modify the graph database, as expected.

Note that the semantics of a query $Q$ should not be confused with the *output* of $Q$. Indeed, the evaluation of a query starts with the graph in the database and the table containing one empty tuple. This graph-table pair is then progressively changed by applying functions that provide the semantics of $Q$'s clauses. The composition of such functions, i.e., the semantics of $Q$, is a function again that defines as follows the graph $G'$ to be committed to memory and the table $T'$ output to the user:

$$\mathsf{output}(Q, G) = (G', T') = [\![Q]\!](G, T_{()})$$

where $T_{()}$ is the table containing the single empty tuple ().

## 8.2 Semantics of update clauses

We now give an overview of the formal semantics for update clauses in Cypher. For most clauses, this semantics is quite straightforward (if technical), with the notable exception of the `MERGE` clauses, which requires a detailed definition of when nodes and relationships may be collapsed in `MERGE SAME`. For this reason, here we only give the semantics of the `MERGE` clause and provide high-level sketches of the semantics of other clauses; for the formal details please refer to [12].

*Composition of clauses.* As already sketched above, sequences of clauses are evaluated from left to right in a compositional manner; i.e., if $C$ is a clause and $\bar{S}$ is a sequence of clauses, we have $[\![C\ \bar{S}]\!](G, T) = [\![\bar{S}]\!]\big([\![C]\!](G, T)\big)$. For unions of queries containing updates, output tables are unioned as for read-only queries, while updates are treated as side-effects in a left-to-right fashion, i.e. for queries $Q$, $Q'$, we apply query $Q$ to the input graph-table pair, then apply $Q'$ to the *resulting* graph and *input* table, and finally combine the graph resulting from $Q'$ with the union of tables derived from $Q$ and $Q'$.

*The `SET` clause.* The semantics of `SET` is defined by means of a two-step process. In the first step, the list $\bar{s}$ of all set items given with the clause are evaluated on the input graph $G$ and table $T$, and all changes to the graph resulting from this are collected in two relations. The relation $\mathsf{propchanges}(T, \bar{s})$ contains all changes to property maps and the relation $\mathsf{labchanges}(T, \bar{s}, n)$ contains label changes to all nodes $n$ in $G$. The latter relation is unproblematic (because `SET` can only *add* node labels and no conflicts may appear here), while the former may contain conflicting changes to properties, as in Example 2. If there are conflicts, the semantics is undefined and the execution of the clause results in error. Otherwise, in the second step, the changes collected in the two relations are applied to the input graph to obtain an updated graph.

*The* `REMOVE` *clause.* The semantics of `REMOVE` is straightforward, as label or property removals may not incur any conflicts; changes induced by given removal items are simply evaluated and applied inductively from left to right.

*The* `CREATE` *clause.* The semantics of the `CREATE` clause proceeds in three steps. First, by a process of *saturation*, all entities (nodes, relationships, paths) in the given pattern that do not carry a variable addressing them are decorated with a temporary variable. Then, nodes are created inductively, followed by the creation of relationships; during this creation, the driving table is updated to bind variables to the created entities. Finally, as the updated graph is returned, all temporary variables assigned during saturation are projected out of the driving table, and the updated graph and table are returned.

*The (*`DETACH`*)* `DELETE` *clause.* The semantics of deleting clauses are again quite straightforward. First, the input expressions are evaluated, and all nodes and relationships to be deleted are collected. For the `DELETE` clause, if deleting the collected nodes and relationships would leave any dangling relationships, the deletion fails; for `DETACH DELETE`, all relationships attached to collected nodes are added to the collection instead. Finally, all collected entities are removed from the input graph, and all references to these entities within the driving table are replaced by `null` values.

### *The* `MERGE` *clause*

We now define formally the semantics of the `MERGE ALL` and `MERGE SAME` clauses. Even though we reference the semantics of the `MATCH` and `CREATE` clauses, which can be found in [12], an intuitive understanding of pattern matching (as given in Section 2) and `CREATE` (explained above) will suffice.

Let $\overline{\pi}$ be a tuple of update patterns. We start with the simpler case of `MERGE ALL` $\overline{\pi}$. For a graph $G$ and a driving table $T$, the semantics of `MERGE ALL` is as follows.

$$[\![\texttt{MERGE ALL } \overline{\pi}]\!](G,T) = (G_{\text{create}}, T_{\text{match}} \uplus T_{\text{create}})$$

$$\text{where} \quad \begin{cases} (G, T_{\text{match}}) = [\![\texttt{MATCH } \overline{\pi}]\!](G,T) \\ T_{\text{fail}} = \{\!\{u \in T \mid [\![\texttt{MATCH } \overline{\pi}]\!](G, \{\!\{u\}\!\}) = \emptyset\}\!\} \\ (G_{\text{create}}, T_{\text{create}}) = [\![\texttt{CREATE } \overline{\pi}]\!](G, T_{\text{fail}}) \end{cases}$$

Here $\{\!\{\}\!\}$ brackets indicate bag semantics, and $\uplus$ is bag union, that adds up duplicates. In particular, $u$ occurs as many times in $T_{\text{fail}}$ as in $T$ if the condition is satisfied.

The semantics of `MERGE SAME` is defined by collapsing nodes and relations from the graph (and table) resulting from the semantics of `MERGE ALL`. Recall [12] that a property graph is formalized as $G = \langle N, R, \mathsf{src}, \mathsf{tgt}, \iota, \lambda, \tau \rangle$ where (1) $N$ is a set of nodes, (2) $R$ is a set of relationships of $G$, (3) $\mathsf{src}, \mathsf{tgt} \colon R \to N$ are functions that map each relationship to its source and target nodes, (4) $\lambda$ is a function that maps each node to a (possibly empty) set of labels, (5) $\tau$ is a function that assigns to each relationship in $R$ its type, and (6) $\iota$ maps each node or relationship to a set of key-value pairs. That is, we have a set $\mathcal{K}$ of keys, and $\iota$ is a function from $(N \cup R) \times \mathcal{K}$ to values; $\iota(n,k) = v$ means that the value of key $k \in \mathcal{K}$ associated with a node $n$ is $v$ (and likewise for relationships). If no value is defined for key $k$, then $\iota(n,k) = \texttt{null}$.

Now assume that $[\![\texttt{MERGE ALL } \overline{\pi}]\!](G,T) = (G',T')$, where $G' = \langle N', R', \mathsf{src}', \mathsf{tgt}', \iota', \lambda', \tau' \rangle$.

*Definition 1.* Two nodes $n_1, n_2 \in N'$ are *collapsible*, denoted by $n_1 \sim n_2$, if all of the following conditions hold:

(i) $\lambda'(n_1) = \lambda'(n_2)$;

(ii) $\iota'(n_1,k) = \iota'(n_2,k)$ for each $k \in \mathcal{K}$; and

(iii) either $n_1$ and $n_2$ are not part of $N$, or $n_1 = n_2$.

The latter condition is to ensure that any nodes that were already present in the original input graph are only collapsible with themselves, and not with any other node, whether already present or newly created.

Being collapsible is an equivalence relation over $N'$. Thus, we can partition $N'$ into non-empty subsets $N_1, \cdots N_k$ such that 1) two elements taken from the same subset are collapsible, and 2) two elements taken from different subsets are not collapsible. From each subset $N_i$, we choose an arbitrary representative that we denote by $n_i$. Note that these choices do not make the semantics nondeterministic: the output graph-table pairs are the same up to id renaming. Then, for each node $n$ in $\mathcal{N}'$, we set $[n] = n_i$, where $i$ is the unique index such that $n \in N_i$.

*Definition 2.* Two relationships $r_1, r_2 \in R'$ are *collapsible*, denoted by $r_1 \sim r_2$, if all of the following conditions hold:

(i) $\tau'(r_1) = \tau'(r_2)$;

(ii) $\iota'(r_1,k) = \iota'(r_2,k)$ for each $k \in \mathcal{K}$;

(iii) $\mathsf{src}'(r_1) \sim \mathsf{src}'(r_2)$;

(iv) $\mathsf{tgt}'(r_1) \sim \mathsf{tgt}'(r_2)$; and

(v) either $r_1$ and $r_2$ are not part of $R$, or $r_1 = r_2$.

Collapsibility is also an equivalence relation over $R'$, and we may then define $[r]$ for each relationship id $r$ just like we did for node ids.

Finally, the semantics of `MERGE SAME` is as follows.

$$[\![\texttt{MERGE SAME } \overline{\pi}]\!] = (G'', T'')$$

where $G'' = \langle N'', R'', \mathsf{src}'', \mathsf{tgt}'', \iota'', \lambda'', \tau'' \rangle$, with

- $N'' = \{[n] \mid n \in N'\}$;
- $R'' = \{[r] \mid r \in R'\}$;
- $\mathsf{src}''(r) = [\mathsf{src}'(r)]$ for every $r \in R''$;
- $\mathsf{tgt}''(r) = [\mathsf{tgt}'(r)]$ for every $r \in R''$;
- $\iota''(v,k) = \iota'(v,k)$ for every $v \in (N'' \cup R'')$ and every $k \in \mathcal{K}$;
- $\lambda''(n) = \lambda'(n)$ for every $n \in N''$;
- $\tau''(r) = \tau'(r)$ for every $r \in R''$;

and where $T''$ is obtained from $T'$ by replacing every occurrence of an element $x$ of $N' \cup R''$ with $[x]$.

## 9. CONCLUSIONS

We analyzed update features of Cypher, a widely used graph query language implemented in the Neo4j database product, and several others. We identified a number of issues and shortcomings in the current implementation and proposed a number of fixes, with which we then consulted the user base, to decide on the final modifications to Cypher updates. These restored atomicity and determinism of update clauses in the language.

The issues of the existing semantics of Cypher 9 update clauses revealed in this study as well as the proposed refinements and corrections are very valuable input to the Neo4j engineering teams. Neo4j plans to integrate the refinements and corrections into its Cypher implementation in future version of the Neo4j database system under the existing deprecation regime to avoid or minimize query breakage for customers. In the *Cypher for Apache Spark™* project [1] and its related upcoming product Neo4j Morpheus, `MERGE ALL` and `MERGE SAME` clauses as part of multiple graphs and graph construction capabilities of Cypher 10.

## Acknowledgments

Authors at the University of Edinburgh were supported by a grant from Neo4j Inc. and EPSRC grant M025268.

## 10. REFERENCES

[1] Cypher for Apache Spark, Oct. 2018. `https://github.com/opencypher/cypher-for-apache-spark/`.

[2] Cypher for Gremlin, Oct. 2018. `https://github.com/opencypher/cypher-for-gremlin/`.

[3] Graph Query Language GQL, 2018. `https://www.gqlstandards.org/`.

[4] MemGraph, Oct. 2018. `https://memgraph.com/`.

[5] openCypher Project, 2018. `http://www.opencypher.org/`.

[6] H. Abelson et al. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[7] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutiérrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18. ACM, 2018.

[8] Bitnine. AgensGraph, Oct. 2018. `https://github.com/bitnine-oss/agensgraph/`.

[9] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985.

[10] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[11] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 510–524. ACM, 2017.

[12] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer, and A. Taylor. Formal semantics of the language Cypher. *CoRR*, abs/1802.09984, 2018. `https://arxiv.org/abs/1802.09984`.

[13] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445. ACM, 2018.

[14] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1):27–39, 2017.

[15] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, pages 274–308, 1992.

[16] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[17] R. Milner, M. Tofte, and R. Harper. *Definition of Standard ML*. MIT Press, 1990.

[18] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 16(3):513–534, 1991.

[19] M. Paradies. Graph pattern matching in SAP HANA. First openCypher Implementers Meeting, Feb. 2017. `https://tinyurl.com/ycxu54pr`.

[20] Redis Labs. RedisGraph, Oct. 2018. `https://oss.redislabs.com/redisgraph/`.

[21] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. O'Reilly Media, 2013.

[22] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[23] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16. ACM, 2016.

[24] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *LNCS*, pages 425–446. Springer, 2010.