OAEP-2023-03 **DOI: 10.54285/ldbc.KKHM1756**

Published on LDBC's website April 2023

Cypher schema constraints proposal

CIP2016-12-16 **"Constraints syntax"**, Mats Rydberg

♦

A presentation summarizing this proposal is appended

"Schema and constraints", Mats Rydberg

First openCypher Implementers Meeting (oCIM 1) - 8 February 2017
SAP Walldorf, Germany

LDBC Open Access to External Papers

In this series, Linked Data Benchmark Council makes papers published originally for a restricted audience available for open access.

Such papers are of interest to our members and the public, and are concerned with topics that relate to the work of LDBC. They are published with the permission of their copyright holders, which may have been given by a licence grant.

This article and accompanying presentation are relevant for the work of the LEX (LDBC Extended GQL Schema) Working Group.

These documents have the character of technical reports: they have not been submitted to or accepted via peer review by an established scholarly publication.

CIP2016-12-16 Constraints syntax

Author: Mats Rydberg <mats@neotechnology.com>

Abstract

This CIP describes syntax and semantics for Cypher constraints. These are language constructs that impose restrictions on the shape of the data graph, and how statements are allowed to change it.

Table of Contents

1. Background

Cypher has a loose notion of a schema, in which nodes and relationships may take very heterogeneous forms, both in terms of properties and in graph patterns. Constraints allow us to mould the heterogeneous nature of the property graph into a more regular form.

2. Proposal

This CIP specifies the general syntax for constraint definition (and constraint removal), and provides several examples of possible use cases for constraints. However, the specification does not otherwise specify or limit the space of expressible constraints that the syntax and semantics allow.

This specification also covers the return structure of constraint commands, see Return record.

2.1. Syntax

The constraint syntax is defined as follows:

Grammar definition for constraint syntax.

```
<ConstraintCommand> ::=
    <CreateConstraint>
  | <DropConstraint> ;

<DropConstraint> ::=
    "DROP", "CONSTRAINT", <SymbolicName> ;

<CreateConstraint> ::=
    "CREATE", "CONSTRAINT", [ <SymbolicName> ],
    "FOR", <Pattern>,
    "REQUIRE", <ConstraintPredicate>,
    { "REQUIRE", <ConstraintPredicate> } ;

<ConstraintPredicate> ::=
    <Expression>
  | <Unique>
  | <NodeKey> ;

<Unique> ::=
    <GroupedPropertyExpression>, "IS", "UNIQUE" ;

<NodeKey> ::=
    <GroupedPropertyExpression>, "IS", "NODE", "KEY" ;

<GroupedPropertyExpression> ::=
    <PropertyExpression>
  | "(", <PropertyExpression>, { ",", <PropertyExpression> }, ")" ;
```

References to existing grammar parts:

- <SymbolicName>
- <Pattern>
- <Expression>
- <PropertyExpression>

References to new grammar parts:

- <ConstraintCommand>
- <CreateConstraint>
- <DropConstraint>
- <ConstraintPredicate>
- <Unique>
- <NodeKey>
- <GroupedPropertyExpression>

The `REQUIRE` clause works exactly like the `WHERE` clause in a standard Cypher query, with the addition of also supporting the special constraint operators `IS UNIQUE`, `IS NODE KEY`, and the new <GroupedExpression> expression. This allows for complex concrete constraint definitions (using custom predicates) within the specified syntax.

For details on `IS UNIQUE`, `IS NODE KEY`, and <GroupedExpression>, see the dedicated sections below: Uniqueness, Node key, Grouped expression.

The term 'constraint expression' is used in the following to describe the expressions that constitute the body of the constraint predicate.

2.1.1. Constraint names

All constraints provide the user the option to specify a nonempty *name* at constraint creation time. This name is subsequently the handle with which a user may refer to the constraint, for example when dropping it. In the case where a name is not provided, the system will generate a unique name.

2.1.2. Removing constraints

A constraint is removed by referring to its name.

Example of dropping a constraint with name `foo`*:*

```
DROP CONSTRAINT foo
```

2.2. Semantics

The semantics for constraints follow these general rules:

1. The constraint pattern define the constraint *domain*, where all elements that would be returned by a `MATCH` clause with the same pattern constitute the domain, with one notable exception (see 3.).
2. The constraint expressions defined in the `REQUIRE` clauses of the constraint definition must all evaluate to `true`, at all times.
3. Elements for which a constraint expression evaluate to `null` under Cypher's ternary logic are *excluded* from the constraint domain, even if they fit within the constraint pattern.
4. The constraint expression must be deterministic and free of side effects (such as graph mutations).

2.2.1. Errors

The following list describes the situations in which an error will be raised:

- Attempting to add a constraint on a graph where the data does not comply with a constraint predicate.
- Attempting to add a constraint with a name that already exists.
- Attempting to add a constraint that the underlying engine does not support enforcing.
- Attempting to drop a constraint referencing a non-existent name.
- Attempting to modify the graph in such a way that it would violate a constraint.

2.2.2. Mutability

Once a constraint has been added, it may not be amended. Should a user wish to change a constraint definition, the constraint has to be dropped and added anew with an updated structure.

2.2.3. Grouped expression

This CIP introduces the concept of a *grouped expression*, consisting of one or more property expressions. A grouped expression expresses a new value type in Cypher: a tuple type. This type exists only for the purposes of the `IS UNIQUE` and `IS NODE KEY` operators and this CIP does not further extend its applicability.

The tuple type is composed of the types of the property expressions. These rules apply:

1. When one of the contained property expressions is `null`, the tuple type is also `null`.
2. When compared for equality to another tuple type, the comparison is equivalent to comparing the property expressions of the tuples respectively, in a conjunction.

A wider definition is not necessary for this type to satisfy the requirements of the `IS UNIQUE` and `IS NODE KEY` operators.

2.2.4. Uniqueness

The new operator `IS UNIQUE` is only valid as part of a constraint predicate. It takes as argument a grouped expression, and asserts that it is unique across the domain of the constraint. Following on rule 3. above, elements for which the grouped expression is `null` are not part of the constraint domain. In particular, in the case where the grouped expression is a single property expression, this means that the uniqueness constraint does not hinder the existence of multiple elements having a `null` value for the specified property.

Example of a constraint definition using `IS UNIQUE`*, over the domain of nodes labeled with* `:Person`*:*

```
CREATE CONSTRAINT only_one_person_per_name
FOR (p:Person)
REQUIRE p.name IS UNIQUE
```

2.2.5. Node key

The new operator `IS NODE KEY` is only valid as part of a constraint predicate. It takes as argument a grouped expression, and asserts that two conditions hold:

1. Each property expression within the grouped expression cannot be `null`.
2. The grouped expression is unique across the domain of the constraint.

By way of 1. the node key constraint avoids applicability of rule 3. above. The domain of a node key constraint is thus exactly defined as all elements which fit the constraint pattern.

Example of a constraint definition using `IS NODE KEY`*, over the domain of nodes labeled with* `:Person`*:*

```
CREATE CONSTRAINT person_details
FOR (p:Person)
REQUIRE (p.name, p.email, p.address) IS NODE KEY
```

The node key constraint can be equivalently expressed using a combination of the `IS UNIQUE` and `IS NOT NULL` operators. The below example illustrates this.

Example of a constraint definition using `IS UNIQUE` *and* `IS NOT NULL`*, over the domain of nodes labeled with* `:Person`*:*

```
CREATE CONSTRAINT person_details
FOR (p:Person)
REQUIRE (p.name, p.email, p.address) IS UNIQUE
REQUIRE p.name IS NOT NULL
REQUIRE p.email IS NOT NULL
REQUIRE p.address IS NOT NULL
```

2.2.6. Compositionality

It is possible to define multiple `REQUIRE` clauses within the scope of the same constraint. The semantics between these is that of a conjunction (under standard 2-valued boolean logic) between the constraint predicates of the clauses, such that the constraint is upheld if and only if for all `REQUIRE` clauses, the joint predicate evaluates to `true`.

2.2.7. Return record

Since constraints always are named, but user-defined names are optional, the system must sometimes generate a constraint name. In order for a user to be able to drop such a constraint, the system-generated name is therefore returned in a standard Cypher result record. The result record has a fixed structure, with three string fields: `name`, `definition`, and `details`.

A constraint command will always return exactly one record, if successful. Note that also `DROP CONSTRAINT` will return a record.

Name

This field contains the name of the constraint, either user- or system-defined.

Definition

This field contains the constraint definition, which is the contents of the constraint creation command following (and including) the `FOR` clause.

Details

The contents of this field are left unspecified, to be used for implementation-specific messages and/or details.

Return record example

Consider the following constraint:

```
CREATE CONSTRAINT myConstraint
FOR (n:Node)
REQUIRE (n.prop1, n.prop2) IS NODE KEY
```

A correct result record for it could be:

```
name         | definition                                    | details
-----------------------------------------------------------------------
myConstraint | FOR (n:NODE)                                  | n/a
             | REQUIRE (n.prop1, n.prop2) IS NODE KEY        |
```

2.3. Examples

In this section we provide several examples of constraints that are possible to express in the specified syntax.

> NOTE
> The specification in this CIP is limited to the general syntax of constraints, and the following are simply examples of possible uses of the language defined by that syntax. None of the examples provided are to be viewed as mandatory for any Cypher implementation.

Consider the graph added by the statement below. The graph contains nodes labeled with `:Color`. Each color is represented as an integer-type RGB value in a property `rgb`. Users may look up nodes labeled with `:Color` to extract their RGB values for application processing. Users may also add new `:Color`-labeled nodes to the graph.

```
CREATE (:Color {name: 'white', rgb: 0xffffff})
CREATE (:Color {name: 'black', rgb: 0x000000})
CREATE (:Color {name: 'very, very dark grey', rgb: 0x000000}) // rounding error!
```

Owing to the duplication of the `rgb` property, the following attempt at adding a constraint will fail:

```
CREATE CONSTRAINT only_one_color_per_rgb
FOR (c:Color)
REQUIRE c.rgb IS UNIQUE
```

Now, consider the following query which retrieves the RGB value of a color with a given `name`:

```
MATCH (c:Color {name: $name})
WHERE c.rgb IS NOT NULL
RETURN c.rgb
```

The `WHERE` clause is here used to prevent an application from retrieving `null` values for user-defined colors where the RGB values have not been specified correctly. It may, however, be eliminated by the introduction of a constraint asserting the existence of that property:

```
CREATE CONSTRAINT colors_must_have_rgb
FOR (c:Color)
REQUIRE c.rgb IS NOT NULL
```

Any updating statement that would create a `:Color` node without specifying an `rgb` property for it would now fail.

If we instead want to make the *combination* of the properties `name` and `rgb` unique, while simultaneously mandating their existence, we could use a `NODE KEY` operator to capture all these requirements in a single constraint:

```
CREATE CONSTRAINT color_schema
FOR (c:Color)
REQUIRE (c.rgb, c.name) IS NODE KEY
```

This constraint will make sure that all `:Color` nodes has a value for their `rgb` and `name` properties, and that the combination is unique across all the nodes. This would allow several `:Color` nodes named `'grey'`, as long as their `rgb` values are distinct.

More complex constraint definitions are considered below:

Multiple property existence using conjunction

```
CREATE CONSTRAINT person_properties
FOR (p:Person)
REQUIRE p.name IS NOT NULL AND p.email IS NOT NULL
```

Using larger pattern

```
CREATE CONSTRAINT not_rating_own_posts
FOR (u1:User)-[:RATED]->(:Post)<-[:POSTED_BY]-(u2:User)
REQUIRE u.name <> u2.name
```

Property value limitations

```
CREATE CONSTRAINT road_width
FOR ()-[r:ROAD]-()
REQUIRE 5 < r.width < 50
```

Cardinality

```
CREATE CONSTRAINT spread_the_love
FOR (p:Person)
REQUIRE size((p)-[:LOVES]->()) > 3
```

Endpoint requirements

```
CREATE CONSTRAINT can_only_own_things
FOR ()-[:OWNS]->(t)
REQUIRE (t:Vehicle) OR (t:Building) OR (t:Object)
```

Label coexistence

```
CREATE CONSTRAINT programmers_are_people_too
FOR (p:Programmer)
REQUIRE p:Person
```

Assuming a function `acyclic()` that takes a path as argument and returns `true` if and only if the same node does not appear twice in the path, otherwise `false`, we may express:

Constraint example from CIR-2017-172

```
CREATE CONSTRAINT enforce_dag_acyclic_for_R_links
FOR p = ()-[:R*]-()
REQUIRE acyclic(p)
```

2.4. Interaction with existing features

The main interaction between the constraints and the rest of the language occurs during updating statements. Existing constraints will cause some updating statements to fail, thereby fulfilling the main purpose of this feature.

2.5. Alternatives

Alternative syntaxes have been discussed:

- `GIVEN`, `CONSTRAIN`, `ASSERT` instead of `FOR`
- `ASSERT`, `ENFORCE`, `IMPLIES` instead of `REQUIRE`
- `ADD` instead of `CREATE`
  - It is desirable for verb pairs for modifying operations to be consistent in the language, and recent discussions are (so far informally) suggesting `INSERT` / `DELETE` to be used for data modification, thus making `CREATE` and `DROP` available for schema modification such as constraints.
- Using a prefix model for uniqueness and node keys, alike `REQUIRE UNIQUE (n.a, n.b)`
  - This was discarded in favour of the suffix model due to similarity with already existing `IS NOT NULL`. Prefix operators are uncommon in Cypher.

The use of an existing expression to express uniqueness — instead of using the operator `IS UNIQUE` — becomes unwieldy for multiple properties, as exemplified by the following:

```
FOR (p:Person), (q:Person)
REQUIRE p.email <> q.email AND p.name <> q.name AND p <> q
```

3. What others do

In SQL, the following constraints exist (inspired by http://www.w3schools.com/sql/sql_constraints.asp):

- `NOT NULL` - Indicates that a column cannot store a null value.
- `UNIQUE` - Ensures that each row for a column must have a unique value.
- `PRIMARY KEY` - A combination of a `NOT NULL` and `UNIQUE`. Ensures that a column (or a combination of two or more columns) has a unique identity, reducing the resources required to locate a specific record in a table.
- `FOREIGN KEY` - Ensures the referential integrity of the data in one table matches values in another table.
- `CHECK` - Ensures that the value in a column meets a specific condition
- `DEFAULT` - Specifies a default value for a column.

The `NOT NULL` SQL constraint is expressible using an `exists()` constraint predicate. The `UNIQUE` SQL constraint is exactly as Cypher's `IS UNIQUE` constraint predicate. The `PRIMARY KEY` SQL constraint is exactly as Cypher's `IS NODE KEY` constraint predicate.

SQL constraints may be introduced at table creation time in a `CREATE TABLE` statement, or in an `ALTER TABLE` statement:

Creating a `Person` *table in SQL Server / Oracle / MS Access:*

```
CREATE TABLE Person
(
   P_Id int NOT NULL UNIQUE,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255)
)
```

Creating a `Person` *table in MySQL:*

```
CREATE TABLE Person
(
  P_Id int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255)
  UNIQUE (P_Id)
)
```

Adding a named composite `UNIQUE` *constraint in MySQL / SQL Server / Oracle / MS Access:*

```
ALTER TABLE Person
ADD CONSTRAINT uc_PersonID UNIQUE (P_Id,LastName)
```

4. Benefits to this proposal

Constraints make Cypher's notion of schema more well-defined, allowing users to maintain graphs in a more regular, easier-to-manage form.

Additionally, this specification is deliberately defining a constraint *language* within which a great deal of possible concrete constraints are made possible. This allows different implementers of Cypher to independently choose how to limit the scope of supported constraint expressions that fit their model and targeted use cases, while retaining a common and consistent semantic and syntactic model.

5. Caveats to this proposal

Some constraints may prove challenging to enforce in a system seeking to implement the contents of this CIP, as these generally require scanning through large parts of the graph to locate conflicting elements.

Schema and Constraints

Mats Rydberg
mats@neotechnology.com

Schema in Cypher

- Cypher is *schema-optional*
  - Fits well with heterogenous data
  - Makes typing and query planning harder
  - Does not fit well with many existing table-based engines
- Constraints are the only tools to enforce structure in the data

Schema in Cypher is a point where we expect there to be major developments as more actors get involved.

New constraint syntax

oC
Proposal

- Consistent syntax for all types of constraints (CIP)
- Re-use as much as possible from the rest of the language
- Allow for a large set of future constraints, some of which are vendor-specific
  - This allows vendors to use more strict schema when necessary

```
CREATE CONSTRAINT <name>
FOR <simple pattern>
REQUIRE <constraint expression>
```

Constraints, examples

oC
Proposal

- Property uniqueness constraint
  - ```
    CREATE CONSTRAINT unique_person_names
    FOR (p:Person)
    REQUIRE UNIQUE p.firstName, p.lastName
    ```
- Property existence constraint
  - ```
    CREATE CONSTRAINT person_must_have_firstName
    FOR (p:Person)
    REQUIRE exists(p.firstName)
    ```

Constraints, examples

oC
Proposal

- Property value constraint
  - ```
    CREATE CONSTRAINT roads_must_have_positive_finite_length
    FOR ()-[r:ROAD]-()
    REQUIRE 0 < r.distance < infinity
    ```
- Property type constraint
  - ```
    CREATE CONSTRAINT people_schema
    FOR (p:Person)
    REQUIRE p.email IS STRING
    REQUIRE p.name IS STRING?
    REQUIRE p.age IS INTEGER?
    ```

Constraints, examples

oC Proposal

- Cardinality constraints
  - ```
    CREATE CONSTRAINT spread_the_love
    FOR (p:Person)
    REQUIRE size((p)-[:LOVES]->()) > 3
    ```
- Endpoint constraints
  - ```
    CREATE CONSTRAINT can_only_own_things
    FOR ()-[:OWNS]->(t)
    REQUIRE (t:Vehicle) OR (t:Building) OR (t:Object)
    ```
- Label coexistence constraints
  - ```
    CREATE CONSTRAINT programmers_are_people_too
    FOR (p:Programmer)
    REQUIRE p:Person
    ```
    ```

That's it!

Questions ?