Despite the maturity of commercial graph databases, little consensus has been reached so far on the standardization of data definition languages (DDLs) for property graphs (PG). At present, there are only a few examples of PG systems offering schema and DDL, e.g. Neo4j's Cypher for Apache Spark and TigerGraph. Neo4j 3.5 already provides the means to express certain basic aspects of schemas: the use of *unique property* and *property existence constraints*—or, more generally, of n*ode keys*—on node and edge labels enables us to enforce nodes (or edges) to have certain properties that moreover uniquely characterize that node (or edge). However, this does not allow users to express more advanced aspects of schemas such as specifying, for a given node or edge label, the collection of all possible associated properties; or constraining whether or not an edge may exist between nodes with certain labels. On the other hand, relational database schema approaches have been shaped by requirements originating from applications and application development methods that were state-of-the-art a few decades ago.

The schemas that (property) graph database systems typically provide are *descriptive* in the sense that they only reflect the data: the schema can be changed by manipulating the data instance directly with no particular restrictions. The flexibility that this entails is generally perceived as a valuable characteristic, particularly in the earlier stages of application development, and especially in conjunction with the now ubiquitous agile software development method. A system that allows for the structure of graph elements to be manipulated and refactored freely, as the understanding and modelling of an application's universe evolves, greatly simplifies the development process in its early stages. As applications mature, however, a gradual shift in priorities occurs. As a concept becomes more stable, well- established, and central in our data model, we must treat it with increasing caution when considering further modifications. The demand for restrictive schema manipulation policies further increases when an application goes into production since data becomes precious and misshaped data can have large financial consequences. By this stage, traditional *prescriptive* schemas are the more appropriate choice.

However, descriptive and prescriptive schemas are only the two extremes of a spectrum of practical agility needs. Moreover, applications in productive use still continue to evolve and some schema changes are inevitable. As an application and its schema grows, some parts of the schema mature faster than others giving rise to a different trade- off, between modification flexibility and demand for restriction, within a single schema. We have been studying prescriptive and descriptive PG schemas by relying on ReGraph, a mathematical framework for schema validation, allowing us to construct both instances and schemas as PGs and to enforce schema validation through the existence of a homomorphism from instance to schema. Mathematically specified graph rewriting rules and their application to graph update instances and/or graph schemas allow to propagate these changes from schema to instance (or vice versa) while keeping the instance and schema consistent at all times.

Our position is the following:

> *We focus on the requirements for graph refactoring and schema evolution based on the use of graph rewriting to express such operations mathematically. In this framework, we show how modern graph database systems should support both descriptive and prescriptive scenarios and allow users to smoothly change their flexibility/restriction requirements.*

Finally, we will also discuss a prototype implementation demonstrating feasibility and showing the need of offering high-level primitives for schema validation and evolution in a PG query language; such prototypes builds on a Python library, called ReGraph, that allows rewriting and propagation by means of Cypher queries using Neo4j as backend.