POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Building reactive processing rules for knowledge graphs

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Alessia Gagliardi**

Student ID: 970196
Advisor: Prof. Stefano Ceri
Co-advisor: Dr. Anna Bernasconi
Academic Year: 2021-22

Abstract

Knowledge graphs are an essential ingredient of modern information systems; they enable semantic search and are widely used both in the scientific world (along the linked data paradigm) and by large companies, such as Google and Amazon (where they enrich search results). Graph databases are emerging as leading technology for supporting knowledge graphs; however, they currently lack semantic support, as their data model is quite simple, and allows to create nodes and edges equipped with simple properties, called property graphs. Along the broad vision of creating a standard query language for graph databases, the scientific community is also proposing the addition of schemas and keys to property graphs. This thesis makes one further step, and proposed adding reactive rules to property graphs.

This addition must take into account a body of work produced in the world of active databases, and particularly how database triggers have been deployed since decades. In particular, the thesis presents Reactive Knowledge Rules, which follow a classical event-condition-action paradigm. While events are simple (node creations and deletions), this study introduces two new concepts, *Guard* and *Alert*, for expressing the condition and action parts of a reactive knowledge rule. The Guard detects knowledge contexts that require further investigation. The Alert globally considers the context; when it reveals a critical condition that may require further analysis, it produces a new node, also labeled Alert, carrying enough information extracted from the knowledge graph to enable a knowledge modification action. Such action is outside the scope of the thesis.

The thesis also considers how knowledge is currently managed; normally, knowledge is proposed within scientific communities, which however need to interact. To reflect this situation, we assume that knowledge graphs can be partitioned into knowledge hubs, each managed by a different community, but the graph includes edges between knowledge hubs, that allow knowledge bridging and sharing. Similarly, the effect of reactive rules can be the production of Alerts nodes whose content must be considered across scientific communities. Finally, the thesis considers how notions of new and old state, which are classical aspects of database triggers, can be supported in the context of reactive knowledge rules.

The thesis is built bottom-up, from a concrete example, monitoring of the spreading of the COVID-19 virus over a geographical region. It also includes a proof-of-concept, implemented upon the Neo4j system and query language Cypher. Many functions need to be implemented outside of Cypher, as part of the APOC library. The results of this thesis can therefore be considered as a starting point for a new research, with many aspects that deserve further investigation.

Keywords: Knowledge graph, active databases, Reactive Knowledge Rules, Neo4j, Cypher, COVID-19

Abstract in lingua italiana

I grafi di conoscenza sono un ingrediente essenziale dei moderni sistemi informativi; consentono la ricerca semantica e sono ampiamente utilizzati sia nel mondo scientifico (tramite il paradigma dei dati collegati) sia dalle grandi aziende, come Google e Amazon (dove arricchiscono i risultati della ricerca). I database a grafo stanno emergendo come tecnologia leader per supportare i grafi di conoscenza; tuttavia, attualmente mancano di supporto semantico, poiché il loro modello dati è abbastanza semplice e consente di creare nodi e archi dotati di proprietà semplici, chiamati grafi di proprietà. Nell'ampia visione di creare un linguaggio di interrogazione standard per i database a grafo, la comunità scientifica sta anche proponendo l'aggiunta di schemi e chiavi ai grafi di proprietà. Questa tesi fa un ulteriore passo avanti e propone l'aggiunta di regole reattive ai grafi di proprietà.

Questa aggiunta deve tenere conto di un insieme di lavori prodotti nel mondo dei database attivi e, in particolare, di come i trigger del database sono stati utilizzati da decenni. In particolare, la tesi presenta le Regole di Conoscenza Reattiva, che seguono un paradigma classico evento-condizione-azione. Mentre gli eventi sono semplici (creazione e cancellazione di nodi), questo studio introduce due nuovi concetti, *Guard* e *Alert*, per esprimere le parti di condizione e azione di una regola di conoscenza reattiva. La Guard rileva i contesti di conoscenza che richiedono ulteriori indagini. L'Alert considera globalmente il contesto; quando rileva una condizione critica che potrebbe richiedere ulteriori analisi, produce un nuovo nodo, anch'esso etichettato Alert, che porta informazioni sufficienti estratte dal grafo di conoscenza per consentire un'azione di modifica della conoscenza. Tale azione è al di fuori del campo di questa tesi.

La tesi considera anche come la conoscenza viene attualmente gestita; normalmente, la conoscenza è proposta all'interno delle comunità scientifiche, che tuttavia devono interagire. Per riflettere questa situazione, assumiamo che i grafi di conoscenza possano essere suddivisi in hub di conoscenza, ognuno gestito da una comunità diversa, ma il grafo include archi tra gli hub di conoscenza, che consentono il collegamento e la condivisione della conoscenza. Allo stesso modo, l'effetto delle regole reattive può essere la produzione di nodi Alert il cui contenuto deve essere considerato tra le comunità scientifiche. Infine, la tesi considera come le nozioni di stato nuovo e vecchio, che sono aspetti classici dei

trigger del database, possono essere supportate nel contesto delle regole di conoscenza reattive.

La tesi è costruita con un approccio dal basso verso l'alto, partendo da un esempio concreto, il monitoraggio della diffusione del virus COVID-19 su una regione geografica. Include anche una prova di fattibilità, implementata sul sistema Neo4j e sul linguaggio di interrogazione Cypher. Molte funzioni devono essere implementate al di fuori di Cypher, come parte della libreria APOC. I risultati di questa tesi possono quindi essere considerati come un punto di partenza per una nuova ricerca, con molti aspetti che meritano ulteriori indagini.

Parole chiave: Grafi di conoscenza, Database Attivi, Regole di Conoscenza Reattiva, Neo4j, Cypher, COVID-19

Contents

1 | Introduction

1.1. Premises

Active database systems enhance traditional database functionalities with powerful reactive processing capabilities. By means of reactive processing, data changes can be monitored; if they reveal an event of interest, a reaction takes place. In this way, active rules provide a uniform and efficient mechanism for dealing with integrity constraints, views and derived data, statistics gathering, monitoring and alerting, and many other database system features and functionalities. For example, in the context of integrity management, active rules restore integrity after violations, by correcting the errors which caused the violation. Other applications may include decisions about buying or selling stocks in reaction to changes in their market value, or changing an airplane travel plan as a result of mutated meteorological conditions or for avoiding plane collisions.

Most of the work on active databases assumes conventional relations (tables) as the underlying description model and assumes some form of standard run-time execution model, e.g., based on the classical transaction model for guaranteeing ACID properties; reactive processing has been defined and standardized in this context since a long time.

Relational data have evolved into new forms, including semi-structured, document-based, various forms of non-tabular data, and graphs. In particular, graphs have emerged as a leading model for representing complex knowledge, along the direction that was initially undertaken by the semantic web and linked data communities, and later adopted for developing large knowledge graphs both in the open-source world (e.g., Wikidata) and in the industry (including Google, IBM, and Amazon).

Graphs are 'unifying abstractions' that can leverage interconnectedness to represent, explore, predict, and explain real- and digital-world phenomena [1]. Large knowledge graphs are already being employed in academia, start-ups, and big-tech companies such as Google, Facebook, and Microsoft, which introduced various systems for managing and processing the growth of big graphs.

Given these premises, this thesis is concerned with adding to graph databases the concepts about reactive processing that are well-known for active databases. As graph databases are used to represent complex knowledge, this extension allows us to introduce **reactive knowledge processing** as a new and important way of dealing with changes in knowledge graphs.

It is somehow surprising that this topic has not emerged earlier; perhaps this is due to the fact that large knowledge graphs have been so far considered in the context of rather static knowledge changes. For instance, if we consider industrial knowledge graphs, the addition of new nodes typically occurs by adding new entities or connections whose role is well understood; knowledge changes are not affecting the main underlying business process.

On the other hand, if we consider other contexts of use, then the creation of knowledge does not occur in this disciplined way. In the scientific world, knowledge is created within different communities, each with its own scientific background and responding to different conventions about how knowledge is created. The experience gathered in the linked data effort has shown how difficult it is to share terminological knowledge across domains even when they are close, e.g., in biology, since the agreement on common ontological terms requires huge coordination efforts.

The complexity of knowledge creation is exacerbated in crisis scenarios, where knowledge production is pushed by needs and occurs at a very fast speed. The recent COVID-19 pandemic has shown a situation where knowledge has been produced at unprecedented speed and within very different scientific communities, including virologists, clinicians, economists, social scientists, and – ultimately – decision-making for politics. Thus, another unexplored aspect in knowledge evolution concerns dealing with knowledge partitioning that occurs when knowledge is contributed by distinct communities of experts and scientists, each with its own organization.

We address this aspect by discussing the partitioning of a large and heterogeneous knowledge graph into independent portions, which we will call **knowledge hubs** so that each knowledge graph represents items of knowledge that are produced within a group of scientists and are independently updated and managed. Note, however, that knowledge is linked so as to form a unique interconnected knowledge graph, and that reactive processing is conceived as a mechanism that should be able to communicate significant changes in knowledge across hubs.

As we can see in Figure 1.1, knowledge must be shared across hubs; in particular, reactions to knowledge changes should cross the hubs' borders and be shared between hubs

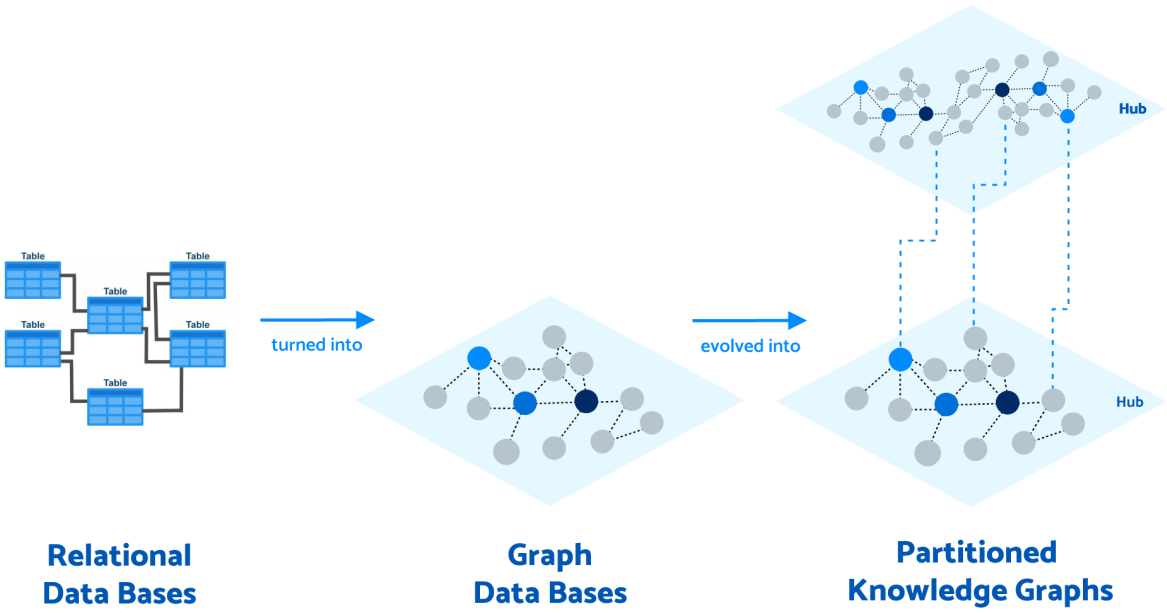


Figure 1.1: Illustration of the evolution from relational data to graphs and to partitioned knowledge graphs

according to agreed paradigms.

As an additional level of complexity, knowledge changes can be observed at different levels of granularity. Reactive processing can capture each individual change, but it is useful to support some form of aggregation so as to communicate changes across hubs only by aggregating substantial amounts of knowledge changes. As each partition is updated according to different dynamics, aggregation is performed along a **temporal dimension**, by aggregating changes after fixed time intervals, e.g. every day. The intuition behind using the time dimension for aggregation is that communication across hubs will take advantage of some form of easily defined coordination.

This analysis of requirements shows that replicating and adapting the active database concepts in the world of partitioned knowledge graphs is challenging from several perspectives. In this thesis, we are producing a first approach to understand and then partially solve some of these problems with a pragmatic, example-driven approach.

1.2. Objective

In this study, our aim is to build a reactive knowledge system using knowledge graphs starting from the modeling of the knowledge to the implementation of reactive behaviors

that act upon changes in the graphs, along three main objectives:

- Model reactive knowledge management as a reactive behavior supported over knowledge graphs partitioned into different hubs. Identify Reactive Knowledge Rules including two components denoted as *Guard* and *Alert* that replace/extend the classical notion of Event-Condition-Action for this new setting.
- Analyse what is offered by current graph database technology to support these concepts.
- Superimpose a temporal dimension to monitor the evolution of knowledge, in particular by allowing knowledge change aggregation. We assume that changes can be periodically consolidated at each hub and that reactive processing should be able to deal with subsequent states of consolidation. This enables us to define notions such as a "statistically significant increase" of a given variable.

1.3. Approach

Although this thesis moves from high-level requirements, its development is grounded within specific technologies and inspired by real-life examples. In particular, after comparative analysis, Neo4j [2] has been chosen as the most representative and widespread graph data model, also thanks to a wide network of open-source supportive initiatives.

The requirements for this study as well as the initial solutions are generated by a concrete example in the COVID-19 pandemic scenario. We will learn from the concrete modeling and implementation issues generated by this example, which are the opportunities and limitations of modeling alternatives that can be supported by Neo4j.

We then progressively introduce aspects that increase the complexity of both the model and the implementation. Eventually, we discuss the possibilities and limitations offered by current Neo4j technology and provide a preview of how the scenario could be improved with semantically enriched graphs, in particular graphs supporting "property graph schemas".

2 State of the art on reactive processing

2.1. Reactive processing concepts

Event-based applications are gaining popularity and becoming ever more important in different application domains. With that, reaction rules and event-processing technologies have been investigated comprehensively over the last decades.

At first, system events were used for interrupt- and exception-handling; then, the focus of event-driven events switched to the implementation of active databases.

Active databases are not the only domain of application for reaction rules. At present times, they have evolved in different ways for a specific domain of application. For instance, another strong demand for event processing functionalities comes from the web community, in particular in the area of Semantic Web and Rule Markup Languages.

Some reaction rules directly react to events as they occur (e.g., active databases), while others focus on the development of axioms to formalize the notions of actions and causality (e.g., KR events/action logics).

Another approach of reactive event processing implies notification and messaging systems that facilitate the communication of events in a distributed environment or monitor external systems by notifying subscribed clients upon detected events.

Finally, we also have event/action logics, which have their origins in the area of knowledge representation (KR) and logic programming (LP); their focus is on the formalization of action/event axioms and on the inferences that can be made from happened or planned events/actions.

In this context, the report written by Paschke and Boley [3] introduces a multi-dimensional classification scheme for the five main families of reaction rules approaches:

- **Event/Action Logics, Transition Logics and Process Calculi**

This family of reaction rules focuses on the development of axioms to formalize the notions of actions as well as events and causality where events/actions have an effect on the actual knowledge states;

- **Dynamic Logics, Update Logics and Tansaction Logics**
  The main aim of these logical update languages is to provide declarative semantics for logic languages with update actions forming a basis also for more general reaction rules such as production rules and ECA rules.

- **Production Rule Systems**
  The production rules system approach specifies an operational semantics or execution semantics formalizing state change. A production rule is a statement of rule programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied, i.e., *if Condition do Action*;

- **Rule-based Complex Event Processing and Event Notification Systems**
  Complex Event Processing (CEP) is an emerging technology to achieve actionable, situational knowledge from distributed systems in real-time. *Complex Event Processing* is defined by the process of event selection, aggregation, hierarchization, and event abstraction;

- **Active Databases and ECA Rule Systems**
  Event-driven applications based on reactive rules and in particular Event-Condition-Action rules (**ECA**), which trigger actions as a response to the detection of events.

Between the listed approaches, for our study, we will focus on the replication of the ECA rules in graph databases. With that intention, we are going to discuss more in detail traditional active databases in the following sections.

2.2. Active databases

Active database systems enhance the functionalities of traditional databases with rule processing rules. Instead, conventional database management systems are indeed considered *passive* systems. By passive we mean that every operation on data (e.g., creation, modification, addition) is performed only after being issued by the user.

The interest in active databases fired between the 1970s and 1980; the first example of Triggering Systems was proposed in 1976 by Kapali P. Eswaran [4], in which the author proposes triggers as extended assertions and as a means to materialize virtual data objects shaping the semantic of triggers that we still use nowadays.

Active databases in traditional database systems, as known at the time of writing, were completed in 1995 by Widom and Ceri [5].

Active database systems differ from passive ones in their ability to automatically respond to certain events occurring in the database if certain conditions are being satisfied. Those mechanisms that respond to events are also called **rules**.

In the course of the years, rules of different types have been developed; we have **internal rules** and **external rules**. The former are rules that are system-generated and not visible to the users. For instance, this type of rule can be applied to check integrity constraints (see [6]), for versioning management, privacy, security, for event recording, and logging. The internal rules are however being surpassed in usage by the external rules. External rules, instead, respond to specific business rules and can be personalized and adapted by the database administrator thanks to context awareness. Active database rules are also known as **triggers**.

Focusing on the SQL3 standard (which was strongly influenced by DBM2), triggers consist of **Event-Action-Condition** rules (ECA). These rules are activated by a database state transition and have an SQL3 predicate as a condition and a list of SQL3 statements as an action. The ECA paradigm states that "*whenever an event* ***e*** *occurs,* ***if*** *a condition* ***c*** *is true,* ***then*** *an action* ***a*** *is executed*":

- `EVENT`
  In ECA rules, events specify what caused the rule to be triggered. Events that trigger rules include *data modification events*, *data retrieval events*, *temporal event*, or *application-defined events*. We notice that the majority of triggers are based on data modification events, i.e., `insert`, `delete`, `update`. However, when translating ECA rules to graph management systems, we will also discuss temporal events. We can say that when the event occurs, the trigger is *activated*;

- `CONDITION`
  In active databases, the condition specifies an additional condition to be checked once the rule is triggered and before the action is executed. When the condition is evaluated, the trigger is *considered*. Useful conditions include: *database predicates*, *database queries*, *restricted predicates*, and *application procedures*;

- `ACTION`
  The action part is executed when the rule is triggered and its condition is true. A trigger can have a variety of action types: *data retrieval operations*, *application procedures*, *data modification operations*. Eventually, when the action is elaborated,

the trigger is *executed*.

In the SQL3 standard, each trigger reacts to a specific data modification operation on a specific table. We here present the syntax for triggers in SQL3:

```
create trigger <TriggerName>
{ before | after }
{ insert | delete | update [of <Column>] } on <Table>
[ referencing { [ old table [as] <OldTableAlias> ]
                [ new table [as] <NewTableAlias> ] |
                [ old [row] [as] <OldTupleName> ]
                [ new [row] [as] <NewTupleName> ] }
]
[ for each { row | statement } ]
[ when <Condition> ]
<SQLProceduralStatement>
```

As shown in the above syntax, an SQL3 is characterized by:

- **Name**
  The name constitutes the identifier of that specific trigger;
- **An execution mode**
  A trigger is specified to execute either `BEFORE`, `AFTER`, or `INSTEAD OF` the triggering operation. In a trigger whose execution mode is `BEFORE`, the trigger is considered before the event, i.e., before the database status change. This type of trigger however needs a safety constraint: a `BEFORE` trigger cannot change the database's status since that could possibly cause a problem for the event happening later, causing an integrity problem for the entire database. Thus, this mode is typically used to check and validate a modification before it takes place, and possibly condition the modification itself.
  An `AFTER` trigger is instead considered (and possibly executed) after the event. It constitutes the most common execution mode, suitable for most applications.
- **A monitored event**
  Typically includes operations like `insert`, `delete`, or `update`.
- **The name of the target (monitored) table**
- **Granularity**
  In SQL3, a trigger can be executed `FOR EACH ROW`, meaning that the condition is evaluated once for each tuple of the specified table (*tuple-level granularity*). Oth-

erwise, when it is executed `FOR EACH STATEMENT`, the trigger is considered (and possibly executed) only once for each activating statement, independently of the number of affected tuples in the target table (*statement-level granularity*). The default setting is `FOR EACH STATEMENT`
References to values before or after the triggering event, respectively called *old* and *new* values, are available in the means of the `REFERENCING` clause. This clause is used to associate names with the new and/or old values of the modified tuples (when the execution is `FOR EACH ROW`) or of the entire table (when the execution is `FOR EACH STATEMENT`). If it is row-level, two transition variables (`old` and `new`) represent the value respectively prior to and following the modification of the row (i.e., tuple) under consideration. Otherwise, if it is statement-level, two transition tables (`old table` and `new table`) contain respectively the old and the new value of all the affected rows (tuples). Transition variables and transition tables enable tracking of the changes that activate triggers, and are crucial to efficiency.

- **Names and aliases for transition values and transition tables**
- **An action**
  The statement that defines the operation to be performed in case the condition is verified.
- **A creation timestamp**
  A very important aspect is the timestamp value given associated with the trigger during its insertion in the database. This value is helpful for operation logging but also in case of conflicts between triggers, since we can assign an order of execution based on the creation timestamp.

The execution of multiple triggers is one of the main concerns for triggers. ECA rules are handled within Trigger Execution Contexts (TECs) and usually, if several triggers are associated with the same event, SQL3 prescribes the following execution policy: `BEFORE` triggers (statement-level first, and then row-level) are considered and possibly executed, then the modification is applied and the integrity constraints defined on the DB are checked. Eventually, the `AFTER` triggers (row-level first, and then statement level) are considered and possibly executed. If there are several triggers in the same category, the order of execution depends on the system implementation (e.g., based on the definition time—older triggers have higher priority).

Moreover, the execution of a trigger action may activate other triggers, that are to be evaluated within new, "inner" TECs. The "outer" TEC is saved, the inner one is built, and its trigger is executed; this is a recursive process. At the end of each inner TEC's

execution, the outer one is restored and its execution is resumed. Any failure during a chain of activation due to a statement causes the rollback of that statement and of all the performed changes up to that point. The execution typically halts when a given recursion depth is reached, rising a "**nontermination**" exception.

It is desirable that a trigger reports the following three properties:

- **Termination**
  For any initial state and any sequence of modifications, a final state is always produced (infinite activation cycles are not possible). Termination is by far the most important property.
- **Confluence**
  Triggers terminate and produce a unique final state, independent of the order in which they are executed (meaningful only if there is non-determinism in the activation, otherwise the trigger will be trivially confluent).
- **Determinism of observable behavior**
  Triggers are confluent and produce the same sequence of messages.

3 | State of art in graph databases

Relational databases have been around for many decades and are the database technology of choice for most traditional data-intensive storage and retrieval applications. Relational database systems are generally efficient unless the data contains many relationships requiring joins of large tables.

With the rise of the Internet as a tool for the general public, data began to increase both in volume and interconnectedness, and SQL-based relational databases were not considered the fittest choice anymore. The NoSQL ("Not only SQL") space brings together many interesting solutions offering different data models and database systems, each more suitable than traditional SQL solutions for certain use cases and shapes of data. In particular, the term “NoSQL” (as a term for modern web data stores), first began to gain popularity in early 2009. Note that, with NoSQL, we refer to a spectrum of technologies that vary largely from each other. The majority of NoSQL systems are aggregate-oriented since they group data based on a specific criterion. Those systems include *key-value stores* (which implement a key to value persistent map for data indexing and retrieval, e.g. BerkeleyDB), *column family stores* (which follow the BigTable model of Google, e.g., Cassandra), *document stores* (which are oriented to store semi-structured data, e.g., MongoDB), and *graph technologies* (which are oriented to store graph-like data).

In this study, we will focus on graphs, a topic that has gained recognition from the IT community but has yet to garner large-scale academic study. Graph database models took off in the eighties and early nineties. Based on the survey by Angles and Gutierrez [7], we report the main concept of graph databases.

- *Data (also called schema)* are represented by data structures that generalize the notion of a graph. Multiple approaches have been studied in the past; specifically, we will focus on the one defined by Hidders and Paredeaens in 1994 [8], which uses labeled graphs to represent schemas and instances.
- *Data manipulation* is expressed by operations whose main primitives are on graph features like paths, subgraphs, graph patterns, connectivity, and graph statistics.

- *Integrity constraints* enforce data consistency. Some examples of these are 1) labels with unique names, 2) typing constraints on nodes' functional dependencies, and 3) domain and range of properties.

The rise in popularity of graph databases is possible thanks to three main properties of these types of data models. First, `Performance`, since traditional relational databases have their own limits as the number and depth of relationships increase, whereas graph databases have no impact on performance, even with large amounts of data. Second, `Flexibility`, because IT and data architect teams move at the speed of business; the structure and schema of a graph model adjust itself as applications and industries change. Rather than exhaustively modeling a domain ahead of time, data teams can add to the existing graph structure without endangering current functionality. Finally, `Agility`, because of the agile approach to the development with graph databases, which allows graph databases to evolve in steps with the rest of the application and any changing business requirements.

Two major categories of graph databases exist: Resource Description Framework (RDF) graphs – also known as triple-stores – and Labeled Property Graphs (LPG).

RDF was developed as a metadata schema for describing things; RDF files consist of a set of logical assertions of the form <`subject, predicate, object`>. Triples are the atom of information and what builds knowledge. When the triples are connected together they form a graph consisting of nodes and edges.

Labeled property graphs (or just Property Graphs) consist of a set of nodes and a set of edges, which corresponds to simple data structures with keys and values. They are defined next; more details on this type of graph database are provided in Section 3.1.

Definition 1. *A property graph is defined as a tuple* $G = (N, E, \rho, \lambda, \phi)$ *where:*

- N *is a finite set of node identifiers;*
- E *is a finite set of edge identifiers, such that* $N \cap E = \emptyset$*;*
- $\rho : E \to (N \times N) \cup \{\{u, v\} | u, v \in N)\}$ *is a total function mapping edges to ordered or unordered pairs of nodes;*
- $\lambda : (N \cup E) \to 2^L$ *is a total function mapping node and edge identifiers to sets of labels (including the empty set);*
- $\pi : (N \cup E) \times P \to Val$ *is a partial function mapping elements and property names to property values.*

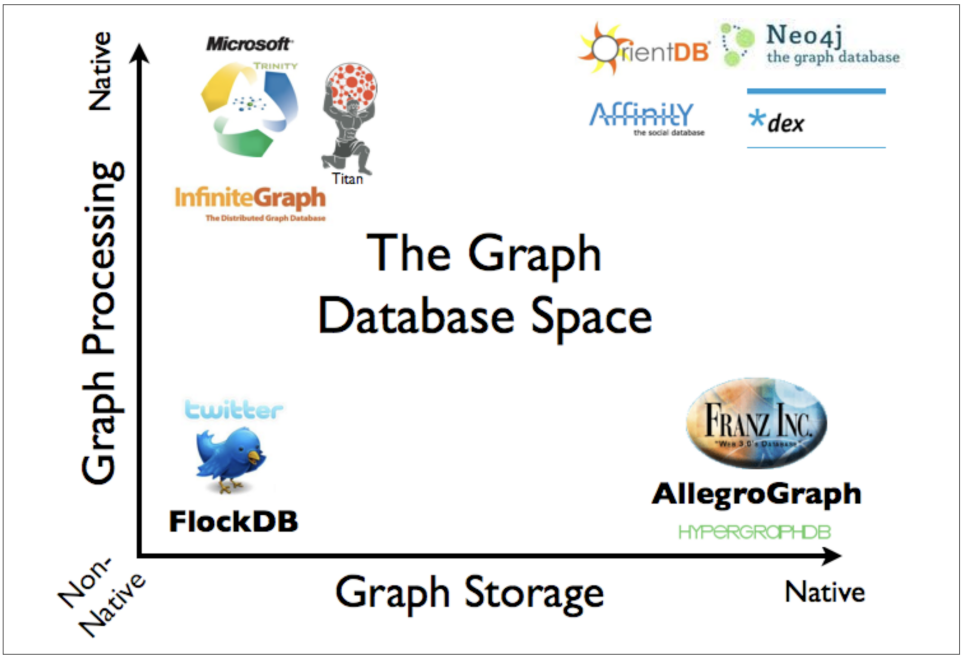


Figure 3.1: An overview of the graph database space

The data model implemented by a database plays an important role in its selection, therefore we need to know its features, advantages, and disadvantages. In the course of the years, numerous graph database technologies have been developed. To choose the most suitable technologies for this work we can exploit some studies that focused on comparing the principal graph database technologies available nowadays. In the article proposed by Fernandes and Bernardino in 2018 [9], they take into consideration five of the most popular graph databases (see Figure 3.1): AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB, comparing them based on some of the most important features a graph database should have:

- `Flexible schema`. Contrarily to relational databases, where the insertion of new data corresponds to the alteration or the addition of new tables, in graph technologies new data can be implemented in the database through vertices and edges without altering any of the already stored data;
- `Query language`. Similar to SQL for relational databases, there exist many different query languages to manipulate specific graph databases such as Cypher on Neo4J, or AQL on ArangoDB;
- `Sharding`. This consists of splitting large datasets by distributing them across a number (typically replicated) of machines. This technique allow NOSQL databases to scale;

	AllegroGraph	ArangoDB	InfiniteGraph	Neo4j	OrientDB
Flexible Schema	1	3	3	4	3
Query Language	3	3	3	4	3
Sharding	3	3	0	0	3
Back-ups	3	2	3	4	3
Multi-model	4	4	2	2	4
Multi-architectures	3	4	3	4	3
Scalability	3	4	3	4	3
Cloud-ready	3	3	4	4	3
Total	23	26	21	26	25

Table 3.1: In this comparison a five Likert scale from 0 to 4 was used. Grade 4 means that the feature is well-implemented, while 0 (zero) is assigned if the feature is not supported by the software. Following, we present the legend for comparison: Great: 4 points; Good: 3 points; Average/Normal: 2 points; Bad: 1 point; Does not support: 0 points.

- `Back-ups`. Graph databases should provide functions for planning, performing, and restoring a database backup. A full backup should allow the restore the database in the exact condition it was at the moment of the backup;
- `Multi-model`. A multi-model graph database provides a database with unstructured data; we can typically visualize relationships with data in the form of graphs, key-value pairs, documents, or tables;
- `Multi-architecture`. When implementing a graph data model there are several structural decisions to make, depending on the graph database product;
- `Scalability`. There are two approaches to scaling a database. Vertical scaling (Scale Up) involves the addition of more physical or virtual resources to the underlying server hosting the database – more CPU, more memory, or more storage. The horizontal scaling (Scale Out) involves adding more instances/nodes of the database to deal with increased workload;
- `Cloud-ready`. To solve scalability problems and provide real-time management of the data, the database should be implemented on the cloud.

Table 3.1 shows a comparison between the five considered graph databases. After this analysis, we concluded that Neo4J and ArangoDB stand out for their functionalities with 26 points that make them the best graph database options nowadays. Neo4j, specifically, is optimized for graph databases and ArangoDB and OrientDB for databases with different data models (e.g., documents, key-value). In our study, we opted for Neo4j, as it also provides well-maintained and documents software, with enhanced implemented features.

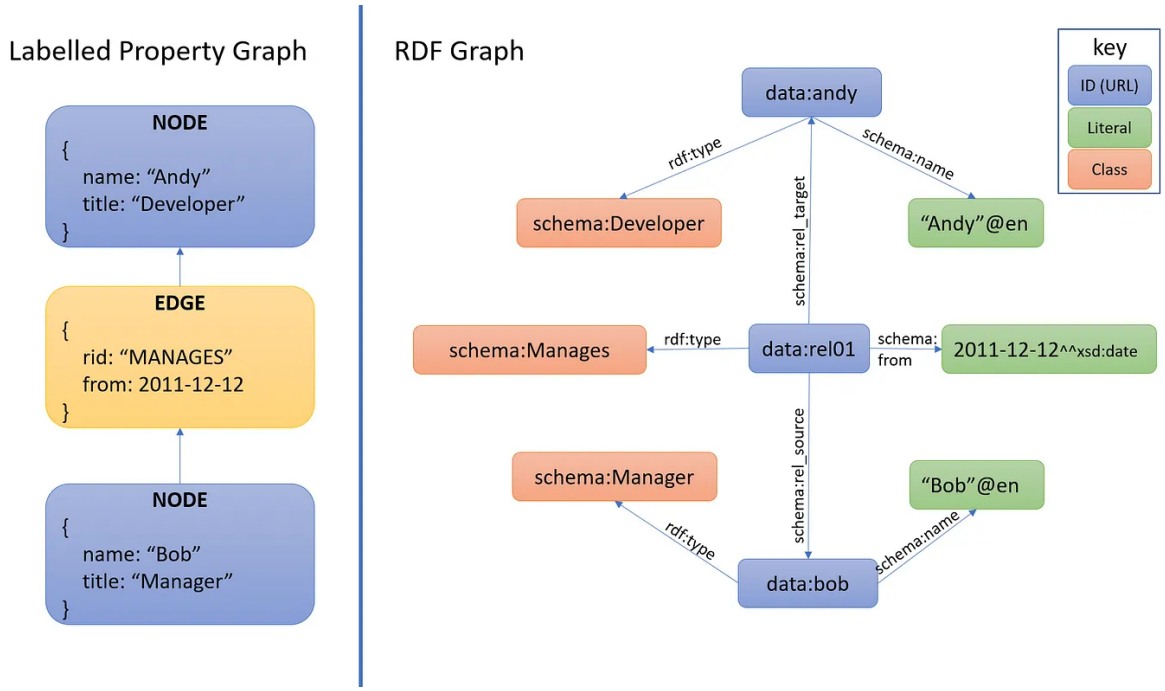


Figure 3.2: Example of a graph implemented with a Labeled Property Graph Model vs. a RDF model

In the following sections of this chapter, we explore the functionalities and properties of the Neo4j database technology.

3.1. Neo4j properties

Graphs are becoming a very powerful tool, useful in understanding a wide diversity of fields such as science, government, and business. Gartner [10], for example, identifies five graphs in the world of business—social, intent, consumption, interest, and mobile—and says that the ability to leverage these graphs provides a “sustainable competitive advantage”.

Figure 3.2 reports a comparison between the two mentioned types of graph data models: RDF and Labeled Property Graph Model. Property graphs are much simpler than RDF ones since they arrange data in nodes and edges both consisting of simple data structures with keys and values. This type of graph leaves the interpretation of keys and values to the consumer of the data; they provide a very intuitive approach to knowledge modeling: nodes represent things that are close to things in the real world, whereas edges represent relationships between those things.

A labeled property graph typically holds the following characteristics:

- It contains nodes and relationships.

- Nodes contain properties (key-value pairs).
- Nodes can be tagged with one or more labels.
- Relationships are named and directed and always have a start- and end-node.
- Relationships can also contain properties.

Neo4j is the tool adopted in the next phases of our study. It started out as a library for the Java programming language (hence the 4j) that enabled the developers to create and store graph-like data structures.

Neo4j responds to the three main properties of graph databases:

- **Performance**. A Reactive architecture - from client applications and consumers through the network stack all the way to the database – allows for efficient use of computation and network bandwidth and minimizes latency for processing. Moreover, it allows complex queries over large graphs and slash request latencies from minutes (or hours) to milliseconds, with fewer resources than RDBMS or NoSQL.
- **Flexibility**. Neo4j implements Cypher as its own query language. Writing, understanding, and maintaining Cypher queries is straightforward, simplifying application maintenance. At the same time, Cypher language is powerful enough for complex processing.
- **Agility**. Neo4j provides a workspace that allows you to directly design a data model, preview the model, import the data, and explore and query the database. Nonetheless, Neo4j offers extended functionalities such as Awesome Procedures On Cypher (APOC), a standard utility library with frequently used and powerful procedures and functions that can be used as building blocks within your Cypher queries– discussed more in detail in Section 3.2.

Previously we talked about how Neo4j sits in the wider space of NoSQL stores. However, this technology presents an aspect that other NoSQL technologies do not have, namely *transactional behavior*. NoSQL models prioritized the increase of performance, scalability, and consistency over the transactional attributes. However, the lack of ACID-base transaction handling is usually the first obstacle when it comes to introducing non-relational databases to any enterprise/corporate environment since transaction attributes still play an important and fundamental part in many practical use cases.

Neo4j has taken a different approach: it implements full ACID support making it a good option both for NoSQL enthusiasts and enterprise environments.

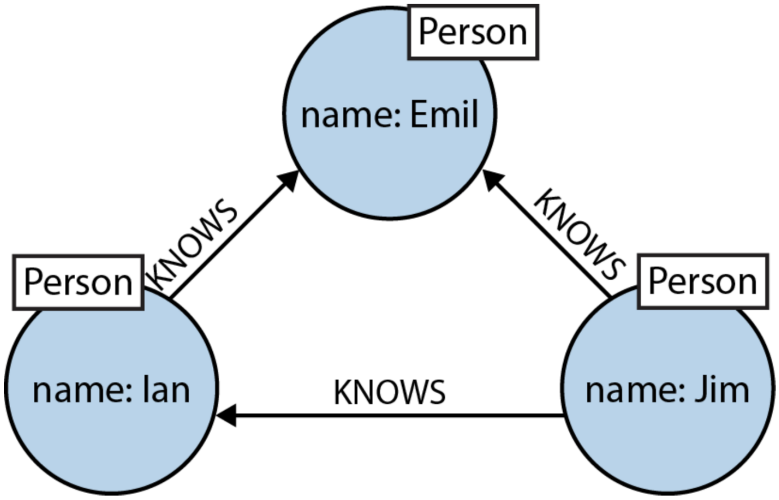


Figure 3.3: A simple graph pattern, expressed using a diagram

- **Atomicity (A)** - It is possible to wrap multiple database operations within a single transaction and make sure they are executed atomically; if one of the operations fails, the entire transaction will be rolled back.
- **Consistency (C)** - In Neo4j every client accessing the database after an operation will read the latest updated data.
- **Isolation (I)** - Operations within a single transaction will be isolated one from another (writes in one transaction don't affect reads in another transaction).
- **Durability (D)** - Data is written on disk and it is available after a database restart or a server crash.

For using the database, Neo4j introduces Cypher, a new expressive (yet compact) graph database query language to create, manipulate, and query data. Cypher is designed to be easily read and understood by developers, database professionals, and business stakeholders; it enables users to ask the database to find data that matches a specific pattern.

By means of the example pattern shown in Figure 3.3, we see how Cypher translates diagrams. The pattern describes three mutual friends. The equivalent representation in Cypher is as follows:

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

From this representation, we can appreciate that Cypher patterns follow very naturally from the way we draw graphs on the whiteboard. In the pattern represented *Ian*, *Jim*, and *Emil* are identifiers, thus they allow us to refer to the same node more than once

when describing a pattern.

The previous representation of a pattern, however, does not translate perfectly to the usage of Cypher. A user must specify some property values and node labels that help locate the relevant elements in the dataset, e.g.:

```
(emil:Person name:'Emil')<-[:KNOWS]-(jim:Person
name:'Jim')-[:KNOWS]->(ian:Person name:'Ian')-[:KNOWS]->(emil)
```

In this implementation, we used a property *name* and a label *Person*. The identifiers (e.g., *emil*) are bound to a node with specific label and property values.

Cypher is composed of clauses. The simplest queries consist of a `MATCH` clause followed by a `RETURN` clause. Most of all, the `MATCH` clause is at the heart of Cypher; it allows one to specify the patterns Neo4j will search for in the database. `MATCH` is often coupled to a `WHERE` part which adds restrictions, or predicates, to the `MATCH` patterns, making them more specific. Another important clause of the Cypher query language is `RETURN`. This clause defines the parts of a pattern (nodes, relationships, and/or properties) to be included in the query result.

A very simple query, returning the node bound to the identifier b, can for instance be:

```
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b
```

There exist many other clauses in the Cypher language. Some of them are:

- `WHERE`
  Provides criteria for filtering pattern-matching results;
- `CREATE & CREATE UNIQUE`
  Create nodes and relationships;
- `MERGE`
  Ensures that the supplied pattern exists in the graph, either by reusing existing nodes and relationships that match the supplied predicates or by creating new nodes and relationships;
- `DELETE`
  Removes nodes, relationships, and properties;
- `SET`
  Sets property values;

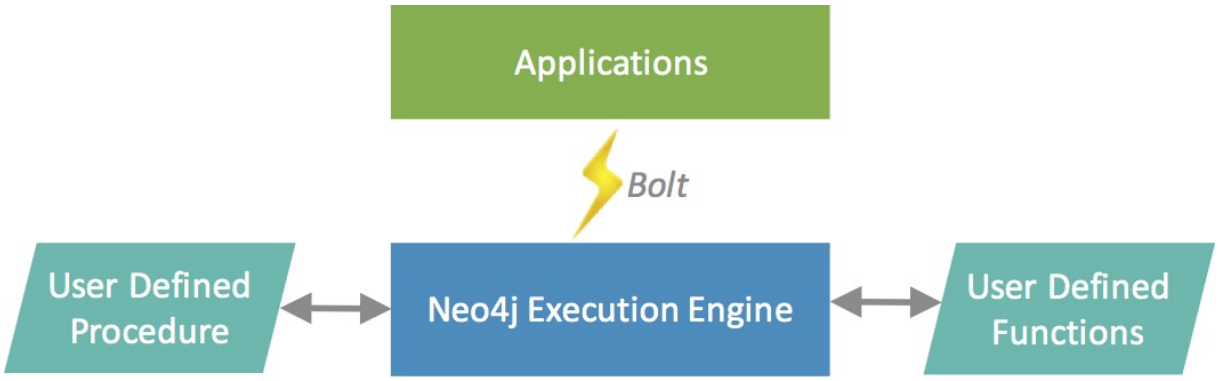


Figure 3.4: Visual representation of the connection between Neo4j and user-defined procedures and functions

- `REMOVE`
  Remove properties from nodes and relationships, or remove labels from nodes;
- `FOREACH`
  Performs an updating action for each element in a list;
- `UNION`
  Merges results from two or more queries;
- `WITH`
  Chains subsequent query parts and forwards results from one to the next. Similar to piping commands in Unix;

3.2. Management of transactions and triggers in APOC

Cypher is an expressive and powerful language, however, sometimes we would like to have more functionalities than what it offers currently (e.g., additional graph algorithms, parallelization or custom conversions). In order to fulfil this need Neo4j and Cypher have opted for the possibility to extend their functions through User Defined Procedures and Functions. Neo4j itself provides and utilizes custom procedures. Many of the monitoring, introspection and security features exposed by Neo4j-Browser are implemented using procedures. Namely, *Functions* are simple computations/conversions and return a single value whereas *Procedures* are more complex operations and generate streams of results that can be used within the `CALL` clause and `YIELD` their result columns.

In order to deploy the created procedures/functions, they need to be packed in a jar file which is then moved to the `plugins` directory of the employed Neo4j server, requiring its restart. Importantly, it should be noted that procedures and functions use the low-level

Java API so as to be able to access all Neo4j internals as well as the file system and machine. For this reason, only trusted sources should be employed.

In this thesis, we focus on the implementation of reactive processing in graph databases. Thus, we need some type of trigger method that runs some action whenever an event happens. Triggers are not natively supported by Neo4j and Cypher at the current time; however, we can make use of their implementation in the *APOC library.*

APOC stands for **A**wesome **P**rocedures **On C**ypher. It is the Neo4j's utility library[1] for handling data import, as well as data transformations and manipulations. It includes over 450 standard procedures, providing functionality for utilities, conversions, graph updates, and more. APOC has become a standard in Neo4j thanks to how well-supported and easy it is to run it through separate functions or to include it in Cypher queries. Starting from Neo4j 4.1.1, two versions of the APOC Library are available: *APOC Core*, including procedures and functions that do not have external dependencies nor require configuration; and *APOC Core*, extending APOC Core with additional procedures and functions.

Two relevant APOC's procedures used in this thesis are `apoc.do` and `apoc.trigger`. `apoc.do` is a collection of procedures used to express conditional queries in Neo4j and Cypher. This collection includes procedures like:

```
apoc.do.case(conditionals [Any],
             elseQuery String,
             params Map<String, Any>)
```

which runs the first query for which the conditionals are evaluated to true. Otherwise, the elseQuery is run.

```
apoc.do.when(condition Boolean,
             ifQuery String,
             elseQuery String,
             params Map<String, Any>)
```

which runs the given read/write ifQuery if the condition has evaluated to true, otherwise the elseQuery.

For achieving reactive processing in graph databases we can be further achieved by employing `apoc.trigger`. By inspecting the lowest level of the Neo4j API stack, it can be observed that this graph database technology is supported by the `TransactionEventHandler`

[1] See `https://github.com/neo4j-contrib/neo4j-apoc-procedures`

class[2] capabilities, which contains the basic functionality of a trigger: when a transaction occurs, it is possible to analyze that event and decide to take some connected action. The `TransactionEventHandler` interface contains three methods: `beforeCommit`, `afterCommit`, and `afterRollback`, that when implemented and registered with the database will be invoked at these key points for each transaction. The event handlers allow user code to listen to transactions as they flow through the kernel, and therefore react (or not) to them based on the data content and lifecycle stage of the transaction.

The `apoc.trigger` procedures in the APOC Core library implement the usage of the `TransactionEventHandler` by giving to the user a non-code experience with the usage of triggers in Neo4j. Within the `apoc.trigger` procedures it is possible to embed Cypher statements, called when data in Neo4j is changed (created, updated, deleted), and run them before or after a commit. With this collection of procedures, we can create triggers that register the creation or deletion of nodes. Moreover, we can pause and resume triggers whenever there is a need. Considering the APOC 4.3 version, the implementation of a new trigger with the APOC library follows this syntax:

```
apoc.trigger.add(name, kernelTransaction, selector, config)
```

In this syntax, the `name` property represents the string containing the name of the trigger just added; the `kernelTransaction` represents the string containing the statement to be executed in case an event happens; and the `selector` property specifies the instant in which the trigger will be activated:

- **before**: the trigger will be activated right before the commit; it corresponds to the default choice;
- **rollback**: the trigger will be activated right after the rollback;
- **after**: the trigger will be activated right after the commit;
- **afterAsync**: the trigger will be activated right after the commit and inside a new transaction and thread that will not impact the original one. The *afterAsync* phase is preferred when heavy operations should be processed in this phase without blocking the original transaction since 'after' and 'before' phases can sometimes block transactions.

The `apoc.trigger` procedures allow to specify which object to consume in the statement to run in the trigger. Thanks to many parameters, the user can indicate the specific data structure to consume. For example, if we wish the trigger to fire when a new node

[2] `https://javadoc.io/doc/org.neo4j/neo4j/2.3.12/org/neo4j/graphdb/event/TransactionEventHandler.html`

is created, then the parameter `createdNodes` should be used. Instead, if we need to fire the statement of the trigger when a node is deleted, we can use the `deletedNodes` parameter. Other parameters include `createdRelationships`, `transactionId`, `deletedRelationships`, `removedLabels`, `assignedLabels`, `assignedNodeProperties`, etc.

3.3. Research on language standardization

Following the growth of the popularity of graph databases, a great number of query languages have been developed (e.g., SPARQL, Cypher, Gremlin). The availability of many languages led to a fragmentation of the market and the detriment of the users.

For relational databases, SQL remains the standard language used to query data. Following the same traces of SQL, the property graph community is trying to create a new query language for graph databases [11]; this should combine new ideas from research (e.g., G-Core, Cypher) and next-generation implementations (e.g. Neo4j Morpheus). It is being built on the same foundations as SQL in order to standardize the querying of graph technologies. The tentative of the ISO/IEC JTC1 SC32 WG3 committee to create a new query language for property graphs is called **Graph Query Language** (also GQL). The project is the culmination of converging initiatives dating back to 2016. The proposal for a project to create a new standard graph query language (ISO/IEC 39075 Information Technology — Database Languages — GQL)[3] was approved in September 2019. As the GQL manifesto says, it is *"a composable declarative graphs-first query language based on pattern matching and graph operations"*. The creation of a standard property graph query language is a four-year project; although the first proposals of this query language have been released[4], the definitive version is expected to appear in 2024 [12].

GQL represents the first international standard database language project after SQL. The project received votes from ten countries [5], including China, Korea, Sweden, the U.S., Germany, the U.K., the Netherlands, Denmark, Kazakhstan, Canada, and Finland. However, five countries chose to abstain from voting due to a perceived lack of expertise to effectively evaluate the proposal. In Figure 3.5 are illustrated the countries that not only voted "Yes," but also committed to providing expert input for the project.

Currently three main query languages exist for property graph databases: *Cypher* (from

[3]See https://www.iso.org/standard/76120.html
[4]See https://github.com/OlofMorra/GQL-parser
[5]See https://www.gqlstandards.org/gql-blogs/critical-milestone-for-iso-graph-query-standard-gql?ref=pr-

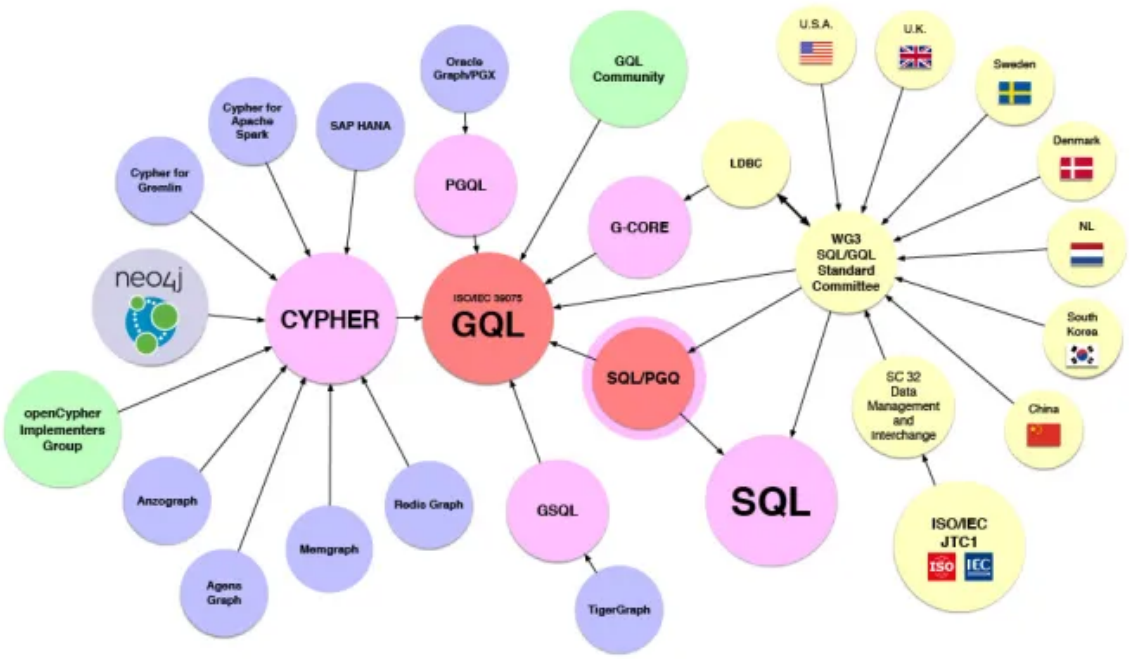


Figure 3.5: The GQL ecosystem. It includes the existing graph query languages, the SQL standard and the various nations that are committed in working at the project.

Neo4j and the openCypher community), *PGQL* (from Oracle), and *GCORE*, a research language proposal from Linked Data Benchmark Council. These languages are similar to each other, only differing in some approaches; GQL is supposed to define a new standard data model which, as we can see from the diagram in Figure 3.6, will likely be a superset of the other graph query languages variants.

The distinctive feature of the new GQL language stands in the freedom that is left to vendors in deciding the cardinalities of the labels and the possibility to support undirected relationships. This project is the product of an academia-industry collaboration model since the main market players are involved and interested in the development of GQL. For instance, the Neo4j Query Languages Standards and Research Team proposed initial ideas in a discussion paper called *"GQL Scope and Features"* [13]. This report (also discussed by the SQL Standards Committee at the time of the 2019 W3C workshop) proposes a well-defined design for GQL in order to cover a multitude of topics such as overall language structure, procedure composition, the type system, essential operations, and views. Moreover, it touches on topics such as expressions, error handling, schema, catalog, subqueries, execution, and security models.

Other studies have continued working on this topic. An example is the paper conducted

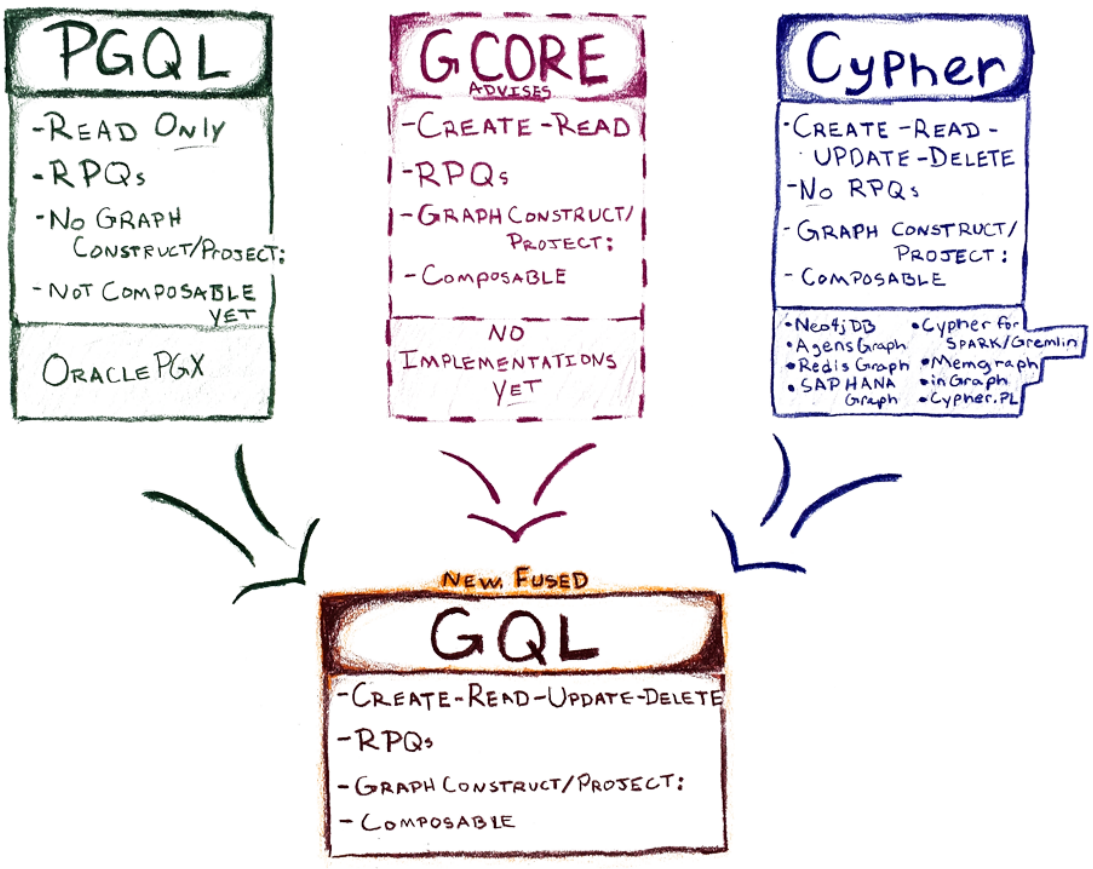


Figure 3.6: Diagram showing differences between existing graph query languages and the goal of GQL (Source: `https://gql.today/`)

by Sharma in 2020 [14] where he proposes `FLUX`, a query translation tool from SQL to GQL. This research aims to answer the research questions of the existence of a formalism for translating relational database queries into graph database queries and what query translation methods can be used to build a tool that translates SQL queries into GQL queries.

3.4. Research on property graphs

The large participation and attention on property graphs led to the start of the ongoing ISO standardization effort, aiming at defining a new standard Graph Query Language (*GQL*) which is expected to appear in 2024[12]. Yet, the proposed versions of GQL pose a few challenges to the development of this new standard query language. With the aim to inspire and incentives the development of GQL, a few articles have been introduced from

the scientific community. Specifically, the articles [15] and [16] provide a formalization of two important concepts for property graphs: the first one introduces the concept of PG-Keys, while the second proposes a definition of a schema for graph databases, the so-called `PG-Schema`.

PG-Schemas can be defined as a schema language that caters to defining nodes and edge types but also to express complex type hierarchies and integrity constraints. The majority of the data definition languages consist of two parts: *types* which define the basic topological structure of the data, and *constraints*, which define data integrity. Similarly, the proposed schema language centers around the two concepts of `PG-Keys` and `PG-Types`.

PG-Keys [15] are *"a flexible and powerful framework for defining key constraints in order to enforce data integrity and allowing the referencing and identifying of objects"*. Keys are used for a few reasons:

- **Constraints**
  Keys are used to prevent nonsensical or contradictory data patterns. For instance, they can be used to avoid the storing of two copies of the same data (e.g., duplication of the Social Security Number).
- **Reference**
  Relational databases include the concept of *foreign key*, which helps one record to reference another. Despite the presence of edges to represent relationships, in property graphs, a reference machine may still be required by external applications that access the database.
- **Identification**
  In some cases, it is crucial to have an identification system for real-life objects (e.g., data models and object-oriented database models which typically introduce an abstract object identifier and a link to real-life objects needs to be established).

The article identifies four types of PG-Keys that form a hierarchy where arrows lead from weaker to stronger types of keys as shown in Figure 3.7 :

- `IDENTIFIER`, e.g., *login* (every user is required to have precisely one, and no two users can have the same login);
- `EXCLUSIVE MANDATORY`, e.g., *email* (every user must have at least one email and no two users can use the same email);
- `EXCLUSIVE SINGLETON`, e.g., *preferred email* (every user may have at most one preferred email for contacting them but again no two users can have the same preferred

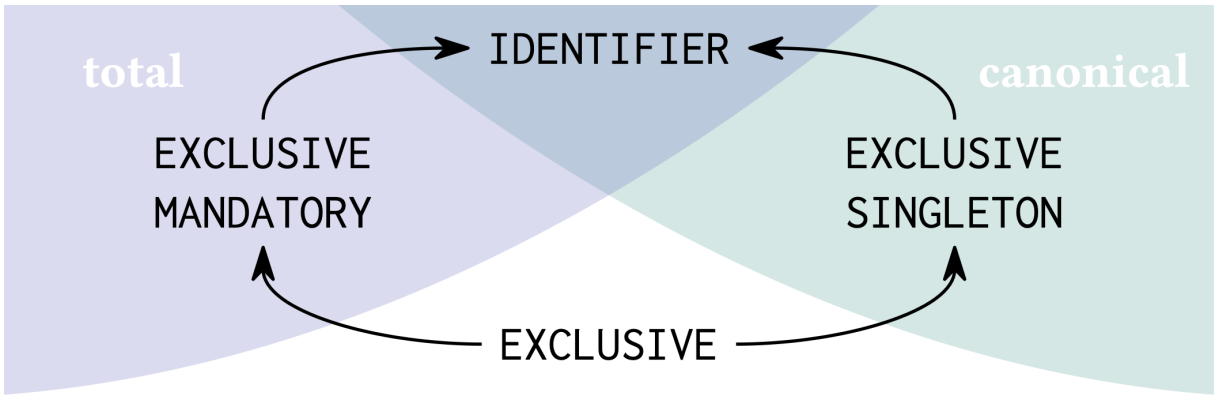


Figure 3.7: Hierarchy for PG-Keys (Source: [15])

email);

- `EXCLUSIVE`, e.g., *alias* (every user may have an arbitrary number of aliases but no two users can have a common alias

Thus, statements in PG-Keys are going to be in the form:

```
FOR p(x) <qualifier> q(x,y)
```

where <qualifier> specifies the kind of constraint that is being expressed and consists of combinations of EXCLUSIVE, MANDATORY, and SINGLETON, and both `p(x)` and `q(x,y)` are queries.

The other concept introduced in the PG-Schema article [16] is `PG-Types`. PG-Types describe the shape of data and the types of its components such as nodes and edges. Moreover, they specify node types, edge types, and graph types by allowing a list of labels and contents or, in the case of graph types, describing the types of nodes and edges present in the graph. For instance, with PG-Types we can associate multiple labels to a node type using the &-operator:

```
(customerType:  Person & Customer name STRING, OPTIONAL since DATE)
```

Eventually, in PG-Schemas, we slightly extend the syntax of PG-Keys by allowing the constraints to refer to a *type name* at each point where PG-Keys allows a label. To see how PG-Keys and PG-Types are used together we propose to inspect the following snippet of code (reported from [16]):

```
CREATE GRAPH TYPE socialGraphType STRICT
{ (personType:  Person name::STRING, id::INT),
(customerType:  Customer id::INT),
(:personType)-[friendType:  Knows & Likes & Bestie?]->(:personType),
```

```
// Constraints
FOR (x:personType) EXCLUSIVE MANDATORY SINGLETON x.id,
FOR (x:customerType) MANDATORY y.id
WITHIN (y:personType) AND y.id=x.personID,
FOR (x:personType) SINGLETON y
WITHIN (x)-[y:  friendType & Bestie]->()}
```

In this example we can notice two *types of nodes*, `personType` and `customerType` but we also have three types of *labels on the edges*: `Knows`, `Likes` and `Bestie`. Regarding the constraint, the code shown above has three types of PG-Key. The first, used for the value of the property *id*, should be a key for nodes of type *personType*. The second PG-Key expresses that every *personID* value of a customer should be an *id* of a person, which makes it a foreign key. Finally, the third PG-Key expresses that each person is allowed to have at most one best friend.

In summary, PG-Schemas represent a great improvement in the journey of the development of a standard Graph Query Language. The work done provides a basis for future research by the academic community leaving the hope to increase functionality and support for current and future customer demands, in addition to the impact on standardization efforts.

4 | Reactive Knowledge Management Concepts and Example

In this chapter, we discuss what knowledge graphs are and present an example to support our study. As established in the previous chapters, when traditional data management systems attempt to capture a context, they fail in reporting the main properties of real-world knowledge:

- *Situational*: since it depends and alters depending on the circumstances;
- *Layered*: associations between concepts allow for different understandings;
- *Changing*: new additions or discovering could change the entire meaning of the knowledge;

Knowledge Graphs, on the other hand, are built with the aim to represent the fluctuating nature of knowledge. Knowledge Graphs are flexible, reusable data layers used for answering complex queries across different data domains by creating connections between contextualized data, represented and organized in the form of graphs.

According to Hogan et al. [17], a knowledge graph can be seen as

> *a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities.*

A Knowledge Graph model comprises two elements: *nodes* and *edges*. Nodes can be resources with unique identifiers, or they can be values with literal strings, integers, or whatever. The edges (also called predicates or properties) are the directed links between nodes. The "from node" of an edge is called the subject. The "to node" is called the object. When connecting two nodes with an edge, a subject-predicate-object statement, known as a *Triple*, is formed. The edges can be navigated and queried in either direction.

Thus, a Knowledge Graph is a directed graph of *triple statements*.

In this chapter, we are going to introduce an example application for reactive processing rules in property graphs considering what was discussed in the previous chapters.

4.1. COVID-19 example

Human coronaviruses (CoV) are positive-stranded RNA viruses responsible for upper respiratory and digestive tract infections. In December 2019 in the Hubei province (China), there was an outbreak of a febrile respiratory illness due to the newly discovered coronavirus, later officially named by the World Health Organization (WHO) as SARS-CoV-2. This virus was considered responsible for the COVID-19 disease. The virus spread across most countries on all continents, causing an unprecedented pandemic [18].

Since the mentioned outbreak, the evolution of the virus has been constantly monitored. Actually, the availability of genome sequences collected over time has been very useful for molecular surveillance of the epidemic and for the evaluation and planning of effective control strategies. This has also implied a huge volume of new data added regularly in public databases [19–21].

Since April 29th, 2021, in Italy, the *I-Co-Gen* platform for genomics surveillance of SARS-CoV-2 variants has been active. As a matter of fact, in Italy, the collection of data about sequencing is done on a regional basis, and in this context, I-Co-Gen allows to retrieve and aggregate data coming from every Region, thereby allowing the analysis of the information upon qualitative standards and to communicate with other similar platforms all over the globe [22, 23].

As discussed in the previous chapters, given the many milestones reached in the research of property graphs, both in for-profit and not-for-profit domains, it is evident that modeling the COVID-19 knowledge as a graph could potentially have an important impact to get a deeper insight into new outbreaks of the virus or simply to monitor the spreading of the mutations.

The example that we are going to present in the following, allowed us to understand better what are the current challenges in building active processing rules in these types of scenarios. Figure 4.1 introduces a reactive knowledge graph for monitoring the spreading of a dangerous viral mutation in a geographical region.

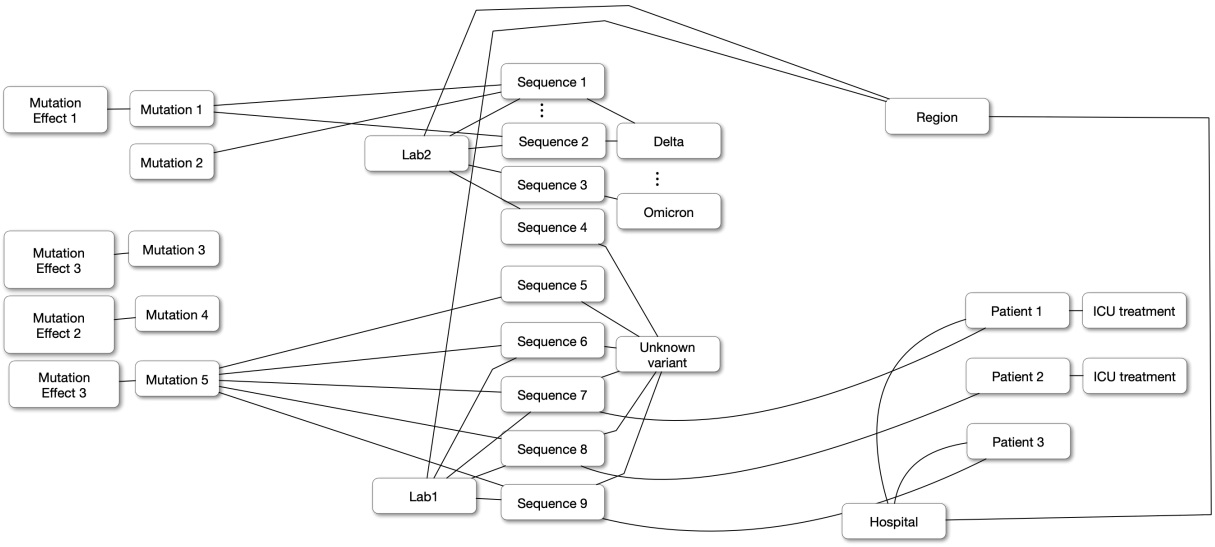


Figure 4.1: COVID-19 knowledge graph example

4.1.1. Entities and Relationships

From the diagram in Figure 4.1, we can identify nine types of entities and many relationships interconnecting the nodes. In particular, we define **Mutations**, **Effects** of the relative mutation, **Sequences**, **Variants**, **Laboratories**, **Regions**, **Hospitals**, **Patients** and **ICU treatments** that is nursing the corresponding patient. We detail each type next.

MUTATION nodes. A mutation refers to a single change in a virus's genome (genetic code). Mutations happen frequently and spontaneously, but only sometimes change the characteristics of the virus. The greatest concern about such emerging mutations is a risky change that could lead to an increase in the severity of the infection or a failure on the effects of vaccines – even though, as noted by Cosar *et al.* [24], most of the times mutations have no notable effect on the spread and the virulence of the virus. These nodes are described using the mutation signature, i.e., holding the original amino acid sequence, its position on a specific protein, and the new (changed) amino acid that has been introduced.

EFFECT nodes. The phenotype of this virus can be strongly affected by given amino acid changes that arise unexpectedly. The position of proteins in the sequence of the effect brought by the virus can change; the most critical effects are due to changes that fall in the Spike protein. The effect is usually reported as a result of a scientific study that used a given method (epidemiological, experimental, computational, or inferred). As reported

by Alfonsi et al. [25], there exist a number of effects types. A new mutation can have an *epidemiological impacts*, meaning that it can bring new viral transmissions, infectivity, disease severity, and fatality rate. It could also have an impact on *immunological* aspects, which can lead to higher/lower sensitivity to monoclonal antibodies, convalescent sera, or existing vaccines. New mutation can also have *protein kinetics impacts*, for instance on protein stability and flexibility. Ultimately, new effects can lead to a different *treatments impact* including vaccine efficacy and drug resistance. In our use case, the `EFFECT` node will be distinguished by its effect definition and will be linked to the relative mutation that manifests that effect.

SEQUENCE nodes. The sequence of a virus corresponds to the sequence (i.e., a string of amino acid residues, expressed with 20 possible letters) that represents the virus' genome. The first sequence of the COVID-19 virus was extracted on January 13, 2020 and deposited to GenBank [18]. The collection of genomic sequences from affected individuals is a fundamental step in helping scientists track the spread of a virus, how it is changing, and how those changes may affect public health. Scientists are indeed consistently accumulating sequences and analyzing similarities and differences among these sequences in a process called genomic surveillance. The sequences analyzed by scientists are collected from a portion of people who test positive for COVID-19 in laboratories or in hospitals. Then, those genomic sequences are analyzed to check whether the sequence is stable or if new changes can be detected. A specific sequence is usually stored in public databases (such as GISAID [19] or GenBank [20]) together with other properties: *accession id* that defines their origin, *lenght* of the genomic sequence, *collection date*, *location*, and *host organism properties* such as its species.

VARIANT nodes. A variant is a viral genome (genetic code) that may contain one or more mutations. When a phenotypic difference is demonstrated among the variants, they are called strains. As reported in [26], the WHO – conferring with the WHO SARS-CoV-2 Virus Evolution Working Group – has categorized the SARS-CoV-2 variants that might pose an increased risk to public health into the following three groups: *variant of interest* that is defined by variants observed to cause community spread to appear in multiple cases or clusters or has been detected in various countries; *variant of concern* which is defined by an increase in transmissibility and virulence or decrease in the effectiveness of the practiced public health; and *variant under monitoring* defined as a variant with genetic changes that are suspected to affect the virus characteristics that it may pose a risk in the future.

Viral evolution can be described by trees in which each sequence is mapped to a node

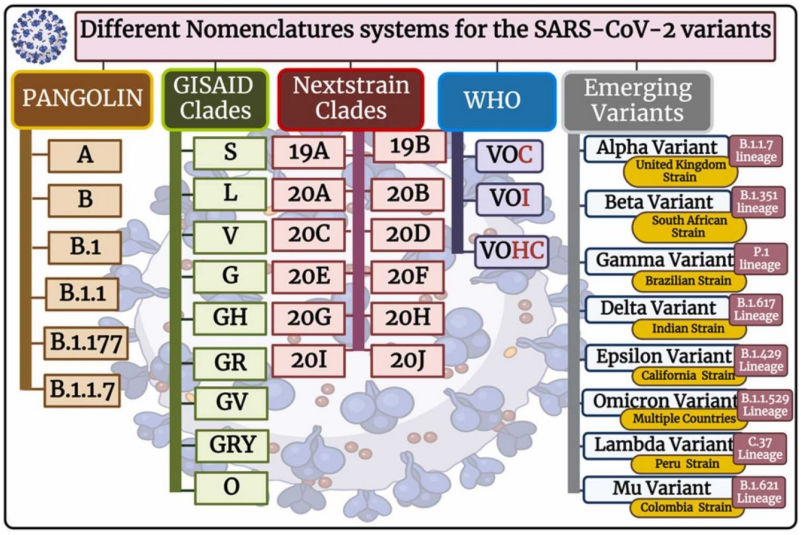


Figure 4.2: Different classifications for the COVID-19 variants (Source: [27]).

and placed in the tree based on its distance to other existing nodes. This mapping allows the partitioning of sequences into clusters, captured at different levels of the tree by different organizations (e.g., lineages, clades). As shown in Figure 4.2, indeed there are multiple ways in which SARS-CoV-2 viruses are classified. Each classification type can be appropriate, depending on the context in which SARS-CoV-2 is being communicated. SARS-CoV-2 is often discussed in the context of lineages (i.e., a group of closely related viruses with a common ancestor).

We decided to define this type of node by three properties, all three representing viruses classification systems: **pangoLineage** [28] and **whoLabel** [29].

ICU TREATMENT nodes. When an individual is admitted to the hospital and results positive for SARS-CoV-2 infection, the hospital needs to follow a series of guidelines on how the patient needs to be treated and what drugs must be used, based on some health factors. The infographics in Figure 4.3, provided by the WHO, illustrates the three disease severity groups and key characteristics to apply in practice.

Several therapeutic options are available for patients with non-severe COVID-19, as well as for those with severe or critical COVID-19. The WHO recommends selecting the therapy to adopt depending on factors such as the availability of drugs, routes of administration, duration of treatment, and time from symptom onset to starting treatment, among others. Patients who experience a critical development of the disease may eventually require life-sustaining treatment, which could mean admission to the Intensive Care Unit (ICU). During the early phases of the pandemic, the lack of guidelines and the severity of the

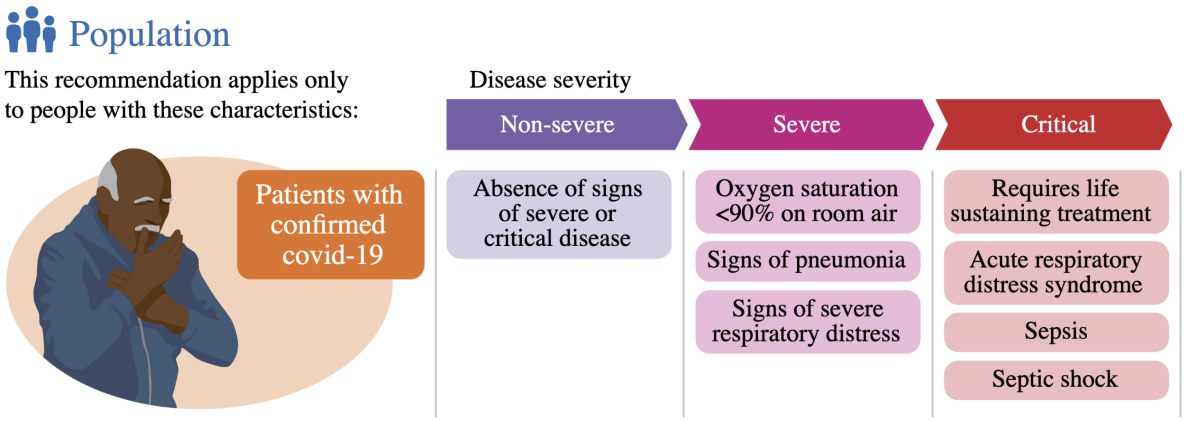


Figure 4.3: Illustration of the three disease severity groups and the treatment options (Source: [30]).

disease led to a huge crisis in hospitals, as they did not have enough available spots to admit people who needed ICU treatment. Even though the need for ICU treatment is decreasing, regional organizations are still monitoring the virus and its link to potential health problems in patients that can lead to critical conditions. To avoid another hospital emergency, it is crucial to have an alarm system that detects critical circumstances. In our case study, we will restrict to identifying the patients that need Intensive Care Unit treatment adding a relationship between the `Patient` node and a `ICU Treatment` node.

The node types we have not discussed explicitely (i.e., *Patient*, *Hospital*, *Region*, and *Lab*) still have a fundamental role in the functionality of the knowledge graph we present. They indeed allow to build the entire infrastructure of entities that operate in a specific region and have a role in capturing an eventual presence of an emergency. Laboratories situated all over the regional territory are in charge of analyzing the virus sequences, thus allowing the detection of possible changes in the development of the virus over time. Regions are also interested in observing the situations for the hospital in their territory by monitoring the entrance of new patients and the situation in the intensive care units.

In order to simplify the described situation and adapt it to our study, we assumed that the nodes of type **Variant**, **Lab**, **Hospital**, and **Region** do not change over time. Instead, modifications (such as creation or removal) are performed on nodes of the types Mutation, Effect, Sequence, Variant, and ICU Treatment.

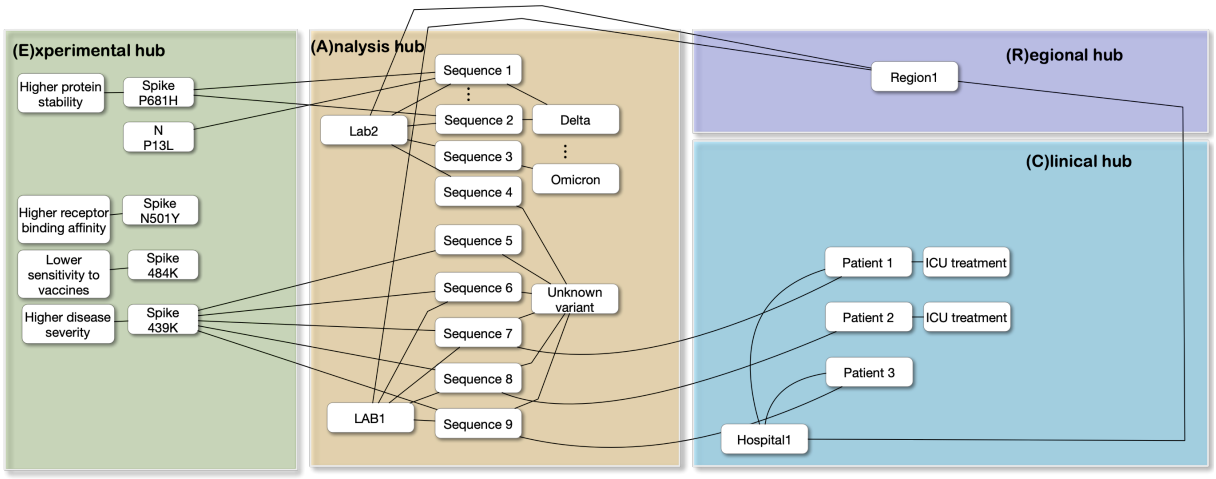


Figure 4.4: Knowledge graphs can be subdivided in hubs which contain the nodes with some type of shared knowledge

4.1.2. Knowledge Hubs

In the context of the case study presented in this chapter we can identify some areas of interest united by the knowledge they share forming an **interaction knowledge graph**. We can therefore map a partition of the domain of applications as we envision four knowledge hubs, as shown in Figure 4.4:

- **Experimental hub** (E) that studies mutational effects.
- **Analysis hub** (A) that associates sequences produced at a given location with known variants.
- **Clinical hub** (C) at a given hospital.
- **Regional hub** (R) at a given region, possibly hosting many hospitals.

Each hub is associated with a portion of the interacting knowledge graph, whose nodes represent (from left to right) mutation effects, mutations, locations, sequences, variants, patients, and their treatments. As observed in Figure 4.4 nodes can have relationships with other nodes *intra-hub* and *inter-hub*; this last type of relationship is the one that allows sharing of the knowledge between hubs. Moreover, hubs represent a central point for organizing and navigating the graph, allowing the identification of noticeable circumstances and making statistics about different domains of knowledge.

4.2. Reactive Knowledge Rules

Real-time monitoring is vital for containing the spread of any virus. Any latency in this process could possibly lead to a catastrophic emergency. Just like in active databases, it would be useful to introduce reactive processing rules also in knowledge graphs. The main contribution of this thesis is to extend the theory and concepts of active databases in order to build **reactive knowledge bases**, which respond to the need of reacting to knowledge changes.

Reactive behavior is modeled as follows. A **reactive rule** is defined as the triple (Event, Condition, Action), where, as in active databases, the event captures data modifications (in a graph data model, the creation and deletion of nodes or relationships). Then for the condition and action parts, we introduce two new components Guard and Alert. More specifically:

- The `Guard` is an existential predicate, true when it matches some nodes of the graph; it reveals situations that can be considered interesting and deserve further investigation.
- The `Alert` is a program that further analyzes the situation in order to understand if it is critical. If this happens, the Alert produces a new node, labeled `Alert`, which includes all the information that is necessary to manage the knowledge change. If, instead, the situation is not considered critical, then the Alert has no side effects.

Note that graph databases provide a coarse granularity for events, as they capture just the insertion or deletion of nodes and edges. Thus, the Guard's main ingredient becomes the selection of nodes that have the same label; in this way, rules become **targeted to node types**, as in the relational model, where each reactive rule is targeted to a specific table. This is aligned with the ongoing effort of adding semantics to property graphs, e.g., schemas [16] and keys [15]. Note also that, in this approach, an Alert does not include a knowledge change. We regard the change of a knowledge graph, as well as the real-world reaction that will occur as a result of an alarm, to be outside of our scope. Both these choices are *semantic restrictions* that we decided to introduce, as the syntax of guards and alerts would allow more liberal choices.

As reactive rules are designed to respond to changes in the graph data structure, each of them typically reference a particular knowledge hub. Each reactive rule refers to the knowledge hub that contains nodes sharing the same label as the "new-node" referenced in the rule.

For instance, a reactive process may want to monitor the creation of new nodes Patient. A rule in the graph database would capture a "created node" event, then it would move on with the conditional part, in the Guard. The Guard could check whether the node just created is of type "Patient" and if the patient just added to the database was connected to a viral sequence belonging to an unknown variant. The Alert would then control the trends in the number of ICU admissions, as it may be useful for the region to analyze some patterns that link new and unknown variants and their consequences on patients. If the Alert verifies that the number of patients admitted to the ICU in the last period is greater than a threshold, then the rule would create an Alert node, to signal to the regional hub a critical condition, so that the region may take action. In our study, we disregard how the alerts are managed, as this is application-specific and could be by means of any combination of human and automated processing.

Although we defined the reactive rules as a triple (Event, Condition and Action), the event of interest monitored always correspond to the creation of a new node. We will follow our discussion representing reactive rules as the couple Guard-Alert as shown in Figure 4.5.

The example mentioned earlier shows a knowledge rules whose effect can span over multiple hubs. However, rules can also be defined completely within a hub. Thus, we distinguish two types of reactive rules:

- `intra-hub`, whose scope (i.e., nodes and arcs affected by guards and alerts) is confined within a single hub.
- `inter-hub`, whose scope spans over multiple hubs.

In addition, knowledge rules may be expressed using information that is carried by the current state or instead comparing several states. Indeed, since their definition [4], reactive processes are actually equipped with variables denoted by keywords `OLD` and `NEW`. Those variables are used to refer to values before and after the data change produced by a modifying event. Thus, we further distinguish two types of reactive rules:

- `single-state`, whose Alert part can be expressed on the current state.
- `multi-state`, whose Alert part requires comparing several states of the knowledge base.

In particular, we will pragmatically allow considering *all the past Alert nodes produced by the same reactive rule*, and not just the current (new) and immediately past (old) state, as we are not constrained to design restrictions for this feature, which is not supported by graph databases.

	Intra-hub	**Inter-hub**
Single-state	Rule which considers the modifications in a single state of the knowledge base and whose domain is defined over a single knowledge hub	Rule which considers the modifications in a single state of the knowledge base and whose domain is defined over multiple knowledge hubs
Across-states	Rule which considers the modifications between multiple states of the knowledge base and whose domain is defined over a single knowledge hub	Rule which considers the modifications across multiple states of the knowledge base and whose domain is defined over multiple knowledge hubs

Table 4.1: Classification of Reactive Rules

Reactive rule on **Experimental Hub**	Single-state	Intra-hub
Reactive rules on **Analysis Hub**	Single-state	Inter-hub
Reactive rule on **Clinical Hub**	Multi-states	Inter-hub

Table 4.2: Classification of the reactive rules implemented in our use case.

These categories are orthogonal; hence, reactive rules can be classified along these dimensions, as illustrated in Table 4.1.

To carry out this study, we introduce four examples of reactive rules that refer to the knowledge graph presented before. Figure 4.4 illustrates four reactive rules. All the rules in Figure 4.4 monitor events of type "*new node created*".

Based on previous considerations, we classify the Guards and Alerts of our example in Table 4.2.

The **Experimental Hub** presents a single-state and intra-hub reactive rule. Following the event of the creation of a new node, the Guard is in charge of checking whether the new node is of type Mutation. The Alert then checks if the new node has a connection with a node of type Critical Effect.

The **Analysis hub** presents two reactive rules; they share the Guard, which looks for the existence of new Sequence nodes whose variant has not been defined yet (i.e., unassigned sequences). The first Alert checks that the number of unassigned sequences does not exceed a certain threshold, by counting the sequences analyzed in the labs of a specific region. The second Alert checks that the number of unassigned sequences with a mutation of critical type does not exceed a fixed threshold. Note that both rules are single-state, as their predicates refer to fixed thresholds and not to inter-state increases, and inter-hub as their Guard/Alert parts involve nodes from various hubs.

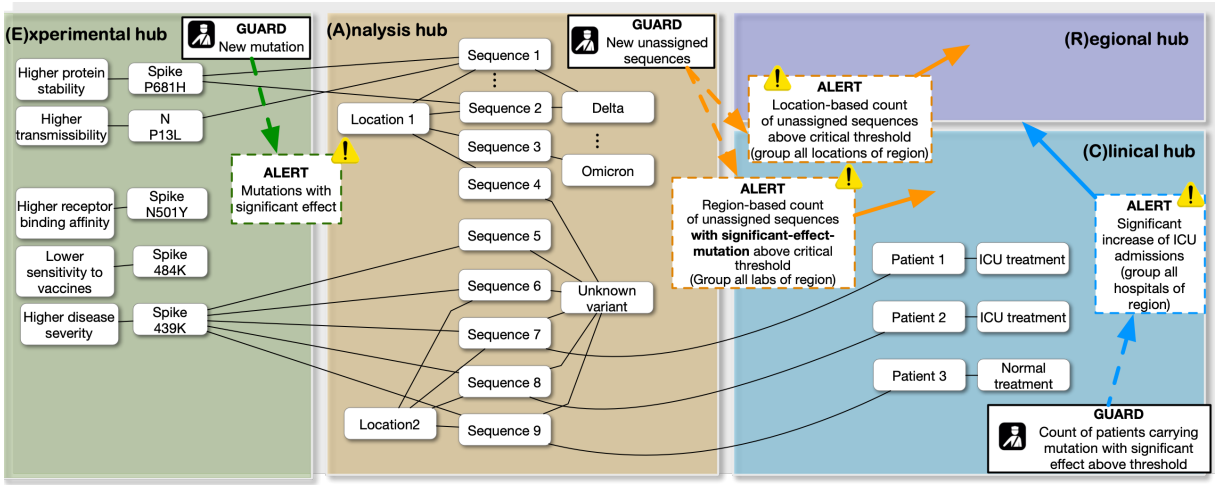


Figure 4.5: Representation of reactive rules in the use case. Each rule has a reference hub, hosting the nodes which are most relevant in the Guard condition. The Experimental hub has an intra-hub rule. The Analysis hub has two inter-hub rules, which share the same Guard but have different Alerts. The Clinical hub has an inter-hub rule.

Finally, the **Clinical Hub** has a single rule whose Guard tests two conditions: first, it checks that the new node belongs to either one of the Patient or Sequence types. Then it counts the number of patients associated with a mutation with a significant effect, and returns a true value when then this number exceeds a threshold. Note that also the addition of a Sequence may change this number if the sequence is linked to the Patient and, at the same time, it carries a critical Mutation. Then, in the Alert, two states are compared; most specifically, the Alert compares the number of patients in ICU across all hospitals in the region in current and past states, and produces an Alert node if such comparison indicates a significant increase. The Alert node includes significant information that may be beneficial to the Regional hub, as it may be opt for implementing strategies to prevent further pandemic peaks.

We represent alarm conditions as nodes, identified with the label `:Alert`, and further associated with several properties that offer a more detailed description of the information that led to the generation of the alarm. This node is not linked to any other node, as it should not alter the computation of other queries over the knowledge graph.

In this chapter, we briefly discussed the necessity to compare multiple states; this aspect deserves a general discussion, which is a specific dimension of rule management. In the next chapter, we delve into the implementation of the single-state reactive rules, while in chapter 6 we will discuss different models aimed at creating multi-state reactive rules, and then discuss their implementations.

5 | Reactive Processing Rules in Neo4j

In this chapter, we explain how reactive rules with guards and alerts have been implemented using the use case introduced in Section 4.1.

5.1. Nodes and Edges Labeling

In section 3.1 we introduced the *labeled property graph model* of Neo4j. Key features can be summarized as follows:

- A labeled property graph is composed by *nodes* that are the entities in the graph, *relationships* that connect two node entities, *properties* and *labels*.
- Nodes and relationships can be tagged with one or more *label*, which distinguish their different roles (e.g., Person entity) in the knowledge domain and groups entities together.
- Nodes and relationships can hold any number of key-value pairs, also called *properties*. Property can be considered as key-value pairs. In Neo4j, the keys are strings and the values are the Java string and primitive data types, plus arrays of these types.
- Relationships connect nodes and structure the graph. Relationships always have a *direction*, a type, a start node, and an end node. Together, a relationship's direction, label, and properties add semantic clarity to the structuring of nodes.
- Nodes can have any number or type of relationships.
- Although relationships are always directed, they can be navigated efficiently in any direction.

To construct a semantically rich model that accurately represents the COVID-19 scenario, we must identify the entities involved. As described in the previous chapter, nine distinct

entity types have been considered: *Mutation, Effect, Lab, Variant, Sequence, Hospital, Patient, ICU Treatment, Region.* These entities can be logically modeled as nodes; then, by using the labeled property model of Neo4j, we can differentiate these entities through labels and properties.

Each node will be assigned a label that specifies the entity type to which the data belongs. The nodes will be modeled as illustrated in the following table:

Entity	**Label**	**Properties**	**Description**
Mutation	`:Mutation`	`"name"`	The label *Mutation* indicates an node of type Mutation and its property `"name"` is fitted to contain the name of the protein that changes in the virus genome.
Effect	`:CriticalEffect`	`"description"`	The label *CriticalEffect* indicates the effect of the change in the genomic heritage of a virus. It is illustrated in Neo4j as a node with a `"description"` property.
Lab	`:Lab`	`"name"`, `"province"`, `"city"`	The label *Lab* indicates a node of type Laboratory and its properties describe the denomination but also its location, through the properties `"province"` and `"city"`.
Sequence	`:Sequence`	`"accession"`, `"collectionDate"`, `"isolate"`, `"mutations"`, `"host"`	The label *Sequence* The nodes with this label take in consideration a few properties. Firstly the `"accession"` which represent the unique identifier for a sequence record. But a node of this type specify also the `"collection date"`, the `"host"` which sequence was collected from, and the `"isolate"` and eventually the `"mutations"` that it presents.

Variant	`:Variant`	`"pangoLineage"`, `"whoLabel"`	The label *Variant* with the properties `"pangoLineage"` and `"whoLabel"` a node of type variant identified by two main scientific classification approaches.
Hospital	`:Hospital`	`"name"`, `"province"`, `"city"`, `"icuCapacity"`	The label *Hospital* simply identify an hospital of a specific region. Its properties helps to describe the hospital with the name and some location information but also the capacity of patients admissible in ICU areas.
Patient	`:Patient`	`"SSN"`, `"gender"`, `"age"`	The label *Patient* express the entity patients describing it through a unique identifier such as the Social Security Number.
ICU Treatment	`:ICUTreatment`	-	The node *ICUTreatment*, if connected to a patient, indicates that the related patient has been admitted to ICU.
Region	`:Region`	`"name"`	The label *Region* identify a node of type Region indicating its name.

Table 5.1: Entity labels and properties

A knowledge graph comprises not only nodes but also connections between them. Relationships are critical for constructing a comprehensive knowledge graph. In our specific domain of application, we have identified several essential connections between nodes. To describe and identify these relationships in our knowledge graph, we have utilized Neo4j labels. We have given these labels semantically meaningful values to facilitate data interpretation. In table 5.2, we present the nodes involved in each relationship, the direction of the connecting edge, and the corresponding label assigned to that relationship.

Relationship	**Label**	**Description**
Mutation →Effect	`RISK_OF`	It connects mutation to their relative effect.

Mutation →Sequence	`FOUND_IN`	It connects the mutation to the sequence where it was found in.
Sequence →Variant	`BELONGS_TO`	It connects the nodes Sequence to each variant group.
Sequence →Lab	`SEQUENCED_AT`	It connects the sequence to the lab that sequenced it.
Sequence →Patient	`AFFECTED`	It connects the sequence to the patient that was affected by a virus with that sequence.
Lab →Region	`LOCATED_IN`	It connects the laboratory to the Region where it is located in.
Hospital →Region	`LOCATED_IN`	It connects the hospital to the Region where it is located in.
Patient →Hospital	`CURATED_AT`	It connects to the patient to the hospital where is cured at.
Patient →ICU	`TREATED_WITH`	It connects the patients to a ICU Treatment node if he/she need the ICU.

Table 5.2: Introduction of relationships that connect the nodes and their identifying label

As outlined in Section 4.1.2, partitioning a graph into distinct areas, where nodes share common knowledge, is beneficial. These areas are referred to as "knowledge hubs". In our graph, we have identified four such hubs. The **Experimental** hub encompasses all information regarding mutations and their potential impact by understanding the genetic evolution of the virus and developing targeted treatments. The **Analysis** hub includes a collective knowledge of entities involved in analyzing and categorizing the SARS-CoV-2 virus is represented. These entities include laboratories, sequences, and associated variants. This hub is crucial in identifying the spread and prevalence of the virus. The **Clinical** Hub encapsulates all knowledge relevant to the hospital domain, including the hospital itself and the patients under its care, while also specifying their need for ICU treatment. Finally, the **Regional** hub serves as a monitor for the knowledge within a specific geographic region. This hub is important in tracking the spread of the virus and implementing targeted interventions.

To model these hubs in Neo4j, we can use labels – which allow us to group entities together. Labels in labeled graphs play a crucial role in identifying a node and grouping entities. By defining hubs as labels and associating them with the respective entities, we can easily query and analyze the graph based on the knowledge hubs.

As a consequence, the nodes that we model have: 1) a label defining the type of entity they represent, 2) a label specifying the knowledge hub they belong to, 3) incoming and outgoing edges that define the relationship with other nodes, and 4) potentially, a various number of properties to effectively differentiate it from other nodes. Figure 5.1 illustrates a small example of how the graph is going to look like.

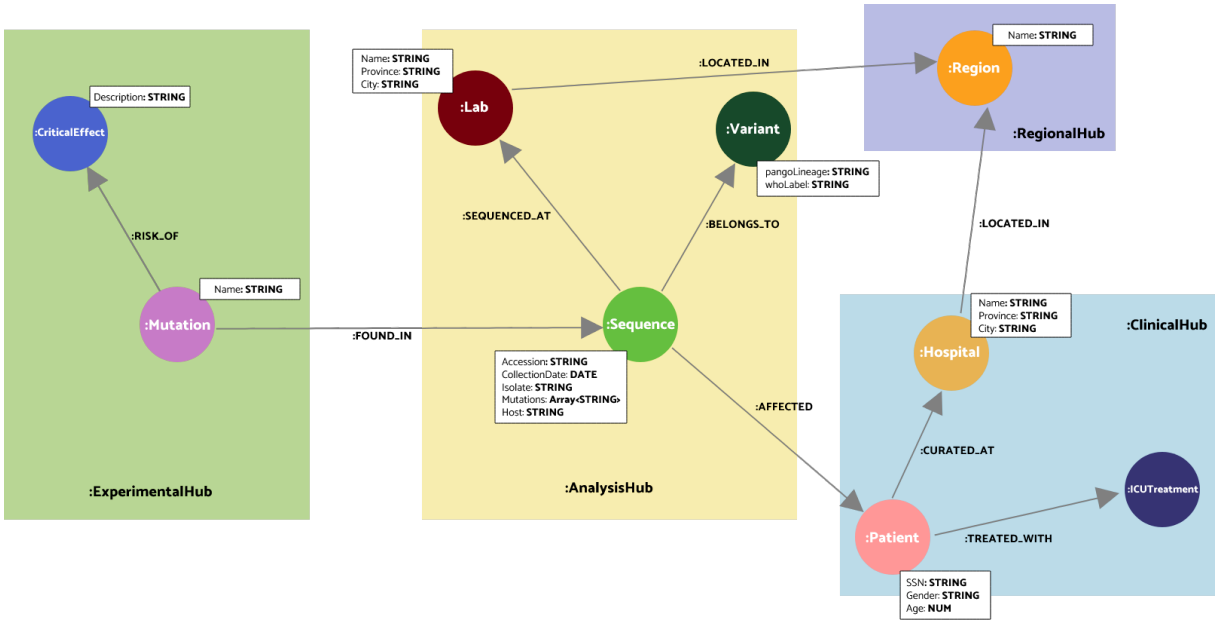


Figure 5.1: Representation of the graph in Neo4j.

5.1.1. Populating the graph

We defined how we are going to model the COVID-19 example; next, we discuss how we actually create nodes and relationships in Neo4j; our interest was in finding a way to automatically build the graph just like we described in Section 5.1.

The motivation to create an automatic process to build the graph came from the fact that -realistically- when users import new data to the knowledge graph, they are most likely not familiar with the Neo4j technology or with Cypher syntax. Thus, we aimed at creating a simple data structure that could ideally be provided by data submitters such as hospitals and laboratories. From such data structure, a small program could then craft Cypher queries based on the type of data that were inserted.

We structure the data into `csv` files designed as follows. These files represent comma-separated tables for the metadata of the various nodes. The first column of each file is intended to register the type of node that the row will contain (e.g., Hospital, Variant, Sequence, Region, Mutation, etc.). The other columns contain the values of the various

properties attributed to each type of node.

After data import, we were concerned with defining the queries that create the relative nodes and relationships. Neo4j offers drivers that enable developers to connect to the database and develop applications for creating, reading, updating, and deleting information from the graph. Then, we developed a small Java application that we called `NodeGenerator.java` that connects to Neo4j thanks to its driver. The process involves using `Gradle` or `Maven` (as in our case) to include the `neo4j-java-driver` dependency. Once Maven has updated its dependencies we can connect to Neo4j and proceed with the execution of queries. After the import of the dependency, we need to initialize the driver in our Java application. The `Driver` object is thread-safe and it is thought to be implemented only once in the lifespan of the application. Moreover, to connect the application to Neo4j we used a connection URL using the `Bolt` protocol (i.e., *bolt://*). For every session that we want to reproduce in Neo4j, a TCP connection is established with the database.

More in detail, after opening a connection with Neo4j, our Java application reads one line of the csv file in input. Then, depending on the value of the first column of each row, it executes one Cypher statement that will be composed through the parameters indicated in the input file.

Before explaining how we managed the creation of relationships we have to get back to the premises we made at the end of section 4.1. We assume that some types of nodes are "static", meaning that no operation of deletion, creation, or modification will be made on them. These nodes include *Region*, *Variant*, *Lab*, *Hospital*. We can therefore consider those nodes as already present when nodes of other types are created.

With that assumption, in the csv file, each node will have a reference to a node of a static type to which it is related. From that, together with the creation of the node, the Neo4j statement will include the creation of the relationships with the related node.

In Table 5.3 we indicate a small example of the creation of a node Patient and its relationships.

Operation	Type	SSN	Gender	Age	Hospital.
Add	Patient	YVOGVR84C57H473N	M	93	Policlinico Universitario A. Gemelli.

```
MATCH (h:Hospital:ClinicalHub) WHERE h.name='Policlinico
Universitario A. Gemelli'
MERGE (p:Patient:ClinicalHub ssn:'YVOGVR84C57H473N' ,
gender:'M', age:93 )-[:CURATED_AT]->(h)
```

Table 5.3: In the upper table, we present how one row of the csv file is structured: the first column contains the node label, while the other columns show the values that the properties of each node have. Moreover, each node has a reference value with all the other nodes connected to it (in this case, the last column contains the name of the hospital to which that patient was admitted). Below, we present the query that was taken into consideration in the csv above to create the node patient with all the properties together with the relationship with the node hospital indicated in the csv.

5.2. Programming of the reactive knowledge rules with guards and alerts using APOC

Here we discuss our approach to implementing reactive processing rules in Neo4j. As introduced in Section 3.2, the APOC utility library extends Neo4j functionalities in terms of handling data import, as well as data transformations and manipulations. Specifically, we are going to focus on the so-called `apoc.trigger` procedures. This collection of procedures allows the registration of Cypher statements that are going to be run in the database on the happening of a relevant event.

The syntax of the procedure of an APOC trigger is as follows:

```
apoc.trigger.add(name, kernelTransaction, selector, config, phase)
```

As parameters, we specify the name of the trigger, the statement to be executed, and the phase in which to fire the trigger after the targeted event. The ECA paradigm is going to be contained in the `kernelTransaction` part.

Similarly to the ECA rules in active databases, in the Guard-Alert system (Section 4.2), the Cypher statement is triggered by an event. APOC triggers do not distinguish the type of event occurred. In order to capture the correct type of event, these procedures provide

a set of parameters to select: the transaction data from Neo4j is turned into appropriate data structures to be consumed as parameters to a statement, as shown in Table 5.4.

Statement	Description
`transactionId`	returns the id of the transaction
`commitTime`	return the date of the transaction in milliseconds
`createdNodes`	when a node is created our trigger fires (list of nodes)
`createdRelationships`	when a relationship is created our trigger fires (list of relationships)
`deletedNodes`	when a node is deleted our trigger fires (list of nodes)
`deletedRelationships`	when a relationship is deleted our trigger fires (list of relationships)
`removedLabels`	when a label is removed our trigger fires (map of label to list of nodes)
`removedNodeProperties`	when a properties of node is removed our trigger fires (map of key to list of map of key,old,node)
`removedRelationshipProperties`	when a properties of relationship is removed our trigger fires (map of key to list of map of key,old,relationship)
`assignedLabels`	when a labes is assigned our trigger fires (map of label to list of nodes)
`assignedNodeProperties`	when node property is assigned our trigger fires (map of key to list of map of key,old,new,node)
`assignedRelationshipProperties`	when relationship property is assigned our trigger fires (map of key to list of map of key,old,new,relationship)
`metaData`	a map containing the metadata of that transaction. Transaction meta data can be set on client side

Table 5.4: Parameters of `apoc.trigger`

In our example, we proposed only triggers acting upon the creation of new nodes. Thus, we will consider the parameter `$createdNodes` from the list of data structures provided by APOC (Table 5.4) which will also provide a reference to the node just created.

`apoc.when()`	only have an if (and maybe else) queries to execute based on a single condition.
`apoc.case()`	check a series of separate conditions, each having their own separate Cypher query to execute if the condition is true.
`apoc.do.when()`	conditional if/else Cypher execution like apoc.when(), but writes are allowed to the graph.
`apoc.do.case()`	conditional case Cypher execution like apoc.case(), but writes are allowed to the graph.

Table 5.5: List of all the procedures and functions of APOC Conditional

We have shown how an event of type 'creation of a node' is captured; next, we focus on how to build the Condition and Action parts of the reactive rules. As illustrated in section 4.2, Guards and Alerts respectively describe the condition and action parts of the reactive rule.

To summarize the mechanisms of our ECA rules in Neo4j, at first we identify the type of event using the parameters made available by the APOC trigger procedure. Then, continuing with the Condition part, we check the assertion defined by the Guard, which will respond with a Boolean value.

In this context, we could consider it as a simple `if-then-else` structure. If the Guard responds with TRUE, we move into the Action part, which communicates a significant situation if the target nodes meet certain criteria. Otherwise, a response of the Guard with FALSE indicates that the event did not cause a critical situation and therefore we can move on.

At the time of writing, Cypher does not include native conditional functionality to address this case, but there are some workarounds that can be used. Some of these options include using correlated sub-queries or using FOREACH for write-only Cypher. (We are going to discuss all the possibilities in section 5.3).

Alternatively to those options, the APOC Procedures library also includes procedures designed for conditional Cypher execution. This collection includes the procedures illustrated in Table 5.5.

In our implementation, we need to include the possibility to write on the graph in case an alarming situation is presented. Thus, the optimal option for our approach is to build the trigger statement and the Guard and Alert system using the `apoc.do.when` procedure.

Consequently, the statement that the APOC trigger runs is composed as follows: first, we identify the type of event to consider (in our case, the creation of a new node); then, through the conditional procedure, we check the condition to be verified, after which we specify the action statement, to be executed in case the condition returns TRUE.

The following Cypher statements report the creation of a new reactive rule as just described: the first shows the structure as is, the second implements Guard and Alert within the first syntax.

```
apoc.trigger.add(TiggerName, "$createdNodes
CALL apoc.do.when(condition, ifQuery, elseQuery)", {phase:afterAsync})
```

```
apoc.trigger.add("TiggerName", "$createdNodes
CALL apoc.do.when ( GUARD, ALERT, ' ' )", {phase:afterAsync})
```

The *condition* and the *ifQuery* parameters of the rule are going to contain respectively Guard and Alert for the relative trigger. Since, the complexity of the two components is arbitrary Cypher may not offer enough functionalities to express all the possible Guard and Alert implementations.

Nonetheless, Neo4j allows the extension of its functionalities through *user-defined-functions*. These mechanisms enable us to extend Neo4j by writing customized code, which can be invoked directly from Cypher. Those functions and procedures are written in Java and compiled into JAR files. They are then deployed to the database by dropping the JAR file into the plugins directory on each database server. The implementation of user-defined function for Guards and Alert used in our use case is going to be discussed in Chapter 7.

5.3. Discussion of alternatives

In the previous section, we talked about the need to reproduce the Guard and Alert mechanisms in a conditional Cypher execution query inside a unique statement. We also noted that Cypher does not provide conditional functionalities yet.

In this section, we discuss the alternatives to perform conditional Cypher execution and why we decided to disregard the native procedures of Cypher in favour of the APOC ad-hoc procedures.

The CASE expression The first option to operate conditional queries within Cypher involves the use of the CASE construct. It can work in two ways. It allows an expression

to be compared against multiple values with the following syntax or it can be used to allow multiple conditional statements to be expressed.

However, the statement can only be used to output an expression. It cannot be used to conditionally execute Cypher clauses. Therefore, it would not be useful to reproduce the Guard-Alert mechanism.

Using correlated subqueries We can use subqueries to implement conditional Cypher execution by combining subqueries with filtering. This option requires the usage of the construct *WITH*. This clause allows query parts to be chained together.

Subqueries, however, are not independent of the outer query, and – if they do not yield any rows – the outer query will not receive any rows to continue its execution. To overcome this problem, some solutions have been found. One involves using a standalone aggregation to restore a row before the subquery return or using a *UNION* subquery to cover all possible conditionals.

This solution represents a concrete opportunity to have conditional queries in Cypher; however, it may not be the solution for our goal. The components that make our reactive rules, Guards and Alert, although related they may have very different domain of interests. That is why it may be interesting to have separate queries instead of correlated subqueries as they can limit the expressiveness of Guards and Alerts.

Moreover, it increases query complexity making the queries hard to read and requiring a solid understanding of the Cypher query language and its capabilities. This solution can also affect performance: subqueries can negatively impact query performance if not used correctly, especially when dealing with large datasets.

Using FOREACH Another option to execute a conditional statement in Cypher is the *FOREACH* clause which is used to perform the equivalent of an IF conditional.

The FOREACH statement can be used to execute a sub-query for each item in a list or collection. If a list has one element, then the Cypher in the FOREACH will be executed. If the list is empty, then the contained Cypher will not execute.

However, this solution allows only write-clauses (i.e., MERGE, CREATE, DELETE, SET, REMOVE), while excluding any other non-write-clause, such as MATCH, WITH, and CALL.

The alternatives presented above hold limitations both in the operation performed and in the readability of the syntax. Thus, the APOC procedures were considered a better option as they allow for more concise and readable queries compared to the other alternatives – ultimately simplifying complex conditional execution logic.

6 | Managing Time Intervals in Reactive Knowledge Management

In the previous chapters, we anticipated the importance of building a comprehensive approach to the description of rules that compare multiple states of the knowledge graph. In this chapter, we delve into this topic: we observe and track the evolution of the graph by creating two models of graph management that allow such comparison.

6.1. Modeling of alerts involving state comparisons

As mentioned in Section 4.2, alerts may compare *multiple states* of the knowledge in the graph. While this problem is well-managed by active databases for the relational model, in a graph database the concept of *OLD* state of the graph is missing. Thus, we need to think of an out-of-the-box solution to tackle this problem. Resuming our example about the spread of COVID-19, we recall our description of the Alert associated in the Clinical Hub, which must reveal a...

"Significant increase of ICU admissions in the hospitals of a region"

This Alert compares at least two states of the knowledge base; more in general, however, we could consider a daily distribution of the ICU admissions, and test the condition over a week, using a moving average. In this context, it is reasonable to define time intervals at which we can observe specific information extracted from the knowledge base. Accordingly, a period of observation refers to a specific time interval during which data is recorded and stored in the knowledge base; the interval can vary in length depending on the data being collected and the needs of the organization. This information can be used to accumulate information over the periods, for identifying trends and patterns over time.

In our example, we consider it reasonable to capture the state of the graph in periods

of 24 hours, hence providing a broad perspective on the daily COVID-19 evolution in a geographic region.

6.2. Ingredients

As just mentioned, managing observation periods introduces the necessity of performing the a book-keeping transaction periodically. However, we still encounter limitations with the Cypher language. While it supports querying graphs and importing and updating graph structures, it does not allow us to run queries periodically; however, implementing a periodic process in a graph database is fundamental for managing a discrete evolution of time.

6.2.1. APOC periodic execution

Once more, the APOC library comes to the rescue. Concerning graph updates, this library takes into consideration possible periodic executions of statements, by making available designated procedures, also called `apoc.periodic` procedures.

- `apoc.periodic.repeat()` submits a repeatedly-called background statement.
- `apoc.periodic.commit()` runs the given statement in separate transactions until it returns 0.
- `apoc.periodic.rock_n_roll()` runs the action statement in batches over the iterator statement's results in a separate thread.
- `apoc.periodic.countdown()` submits a repeatedly-called background statement until it returns 0.
- `apoc.periodic.submit()` submits a one-off background statement.
- `apoc.periodic.iterate()` runs the second statement for each item returned by the first statement and returns the numbers of batches and total processed rows.

Among all the procedures available, the most suitable for our need is `apoc.periodic.repeat()`. As suggested by its name, this procedure runs a defined statement in the specified period during its implementation. The procedure follows the syntax:

```
apoc.periodic.repeat('name',statement,repeat-rate-in-seconds, config)
```

6.3. Modeling of graph evolution

We propose two models for managing observation periods and highlighting their advantages and disadvantages.

6.3.1. Total Replication Model

An immediate approach is to periodically replicate the entire graph. In this way, we ensure a comprehensive representation of the graph evolution, encompassing the nodes and their relationships. This enables users to obtain a complete visualization of the data history and how it periodically evolves within the graph, therefore, gaining a deeper understanding of the data and making more informed decisions. To ensure the effectiveness of the periodic replication approach, it is essential to specify the time intervals at which replication occurs; this allows users to plan accordingly and retrieve the desired data at the appropriate time.

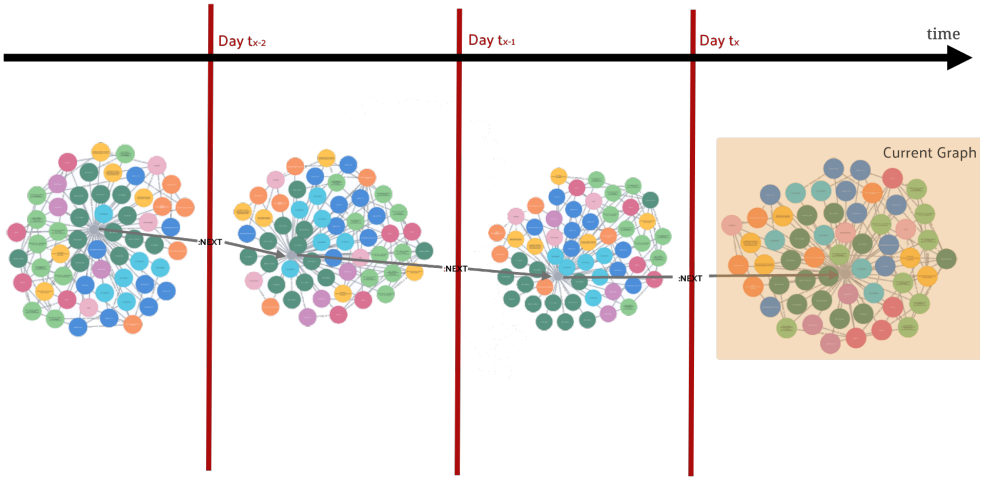


Figure 6.1: General representation of the Total Replication model, at different points in time, i.e., $t_x - 2$, $t_x - 1$, and t_x. Each version of the graphs is connected to its most recent replica through the relationship `:next`, with the chain ending at the "Current" graph.

For instance, using the example of the Clinical Hub's Alert, a complete replication of nodes and edges can help retrieve the number of patients in the ICU in the considered region at a previous time and compare it with the current situation. This approach can be used to implement other types of temporal queries that examine past situations; by querying the versions of the graph, users can obtain a comprehensive view of the data and identify patterns and trends. The graph connected to the labeled node `current` keeps all

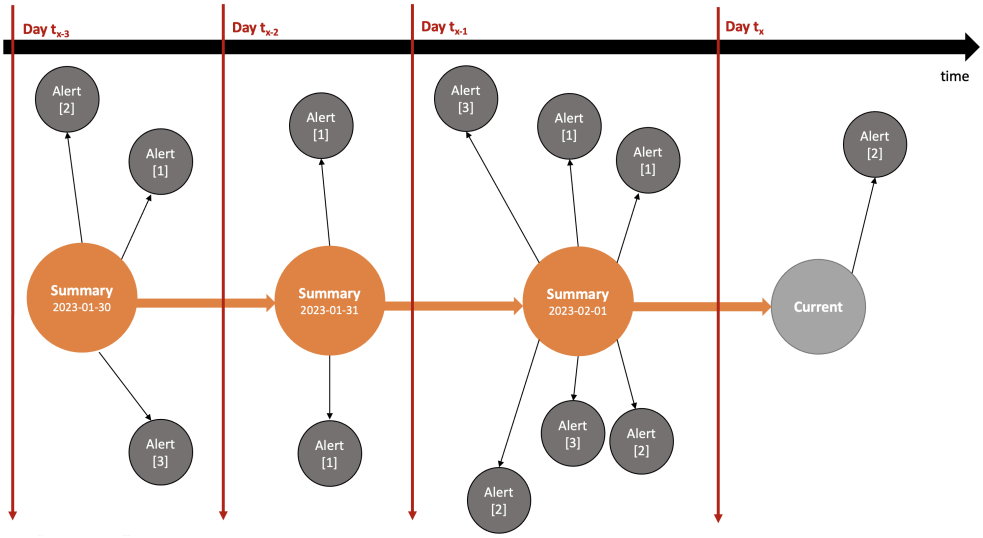


Figure 6.2: Schematic representation of essential summary creation, at four points in time, i.e., $t_x - 3$, $t_x - 2$, and $t_x - 1$ and t_x.

the information about the latest state, so that read and write operations are performed on the current graph with low latency, allowing users to access the data with a small delay, consisting in extracting only the current nodes and arcs of the graph. However, a total replication incurs in high storage/versioning costs, as we discuss later.

In this total replication setting, each replication of the graph database must refer to specific time period. When a copy of the graph is created, each node of that copy will be connected to a special node that has a reference to the date and time of the creation of the copy. This special node holds the label `:Version` and the properties `versionDate` and `creationTime`, which respectively contain the date and time on which that replica was created. Each special node labeled `:Version` is linked to the next version node by a relationship, labeled `:next`, forming a chain until the `:current` node. Alert nodes carry a timestamp, hence they can either remain dangling or be linked to the version in which they are created. Figure 6.1 shows an example of several versions of the graph, each linked to the next one, until the Current graph, with some dangling Alert nodes.

The main limitation of the global replication solution is the high replication cost; moreover, the need to establish a central node of the "Current" graph adds an additional complexity to any query on the current state, as it requires adding to the query matching conditions with the central node.

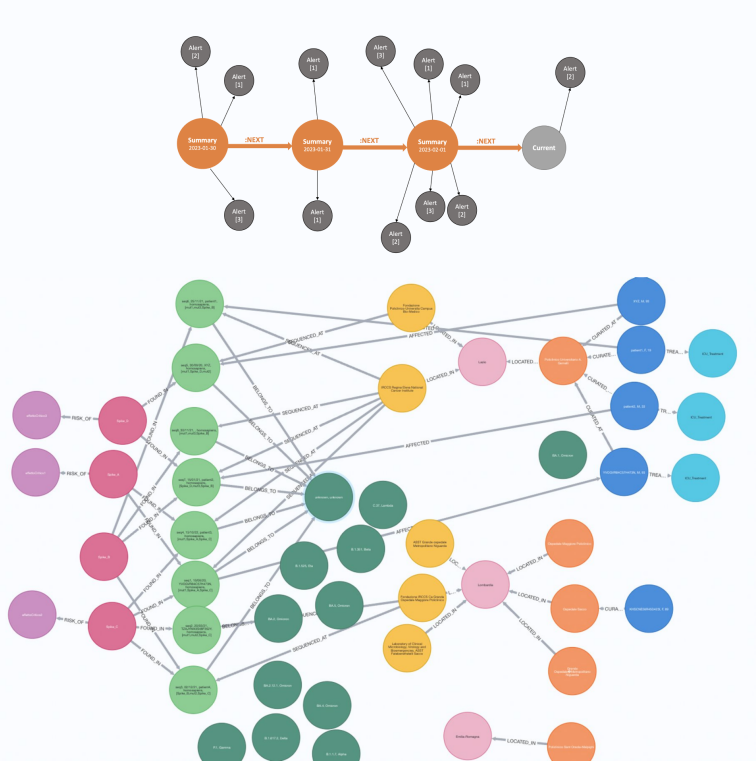


Figure 6.3: Exemplification of the knowledge base with the Essential Summary and the knowledge graph as separate components.

6.3.2. Essential Summary Model

The Essential Summary is a specific data structure which keeps track of the alerts nodes created over time, rather than the entire graph. In this model, new `:Alert` nodes are linked to the `Current` period; along the same approach used in the total replication mode, at each new interval the label of the current node changes into `Summary`, and a new current node is created, labeled with its specific time, and linked to the previous `current` nodes.

Alert nodes remain linked to the summary node, denoting the period in which they were created. As shown in Figure 6.2, the nodes Summary and the Current node are also connected by a `next` relationship, that allows traversing the Essential Summary data structure from the oldest summary up to the current node.

As shown in Figure 6.3, the knowledge base consists of the entire knowledge graph, which is subject to arbitrary changes of its content, and an Essential Summary structure, containing all the nodes of type alert created along the knowledge base history. We argue that this model, once coupled with a careful modeling of the information contained in the Alert nodes, supports in a nice way concepts that recall the old and new states of active databases, although with a periodic nature. Indeed, if for instance an Alert simply

contains as property the counter of patients in ICU, then this data structure supports arbitrary statistics on their distribution over the intervals (e.g. by date). Of course, this model can be much more powerful, as properties associated with alert nodes are arbitrarily complex.

6.4. Implementation

We here proceed to discuss how the two models are implemented; clearly, both models require a an application-independent query, managed by the reactive processing system, which performs the transformations needed at each new interval. The transformation is a classical *action logging process*, which is implemented within the database system internals (e.g. for creating the logs of the recovery system, see [31]); the log grows from its top with a new entry, as the process locates the `summary` node, creates a new `current` node, adds to it properties of the new interval, and then links the two nodes. In this way, all summary nodes are linked to create a chain, which ends in the current node.

This process is described in Figure 6.4, that describes the transformations required by the essential summary approach. The process has two parts. The first one checks that the current period has expired, the second part performs the transformation.

Periodic execution for the Essential Summary model. As anticipated in Section 6.2, we use the `apoc.periodic.repeat()` procedure to perform the periodic execution of a query. This procedure enables us to define the frequency of execution (specified in seconds) as well as the query to be executed. As Figure 6.5 shows, the procedure is fired at shorter intervals than the one used in the system; for our purposes, we fire the procedure every hour, while the time interval for action logging is set at every day.

The rationale of this choice, which comes after some tuning and testing, is the following: by setting the test directly on the action logging interval of 24 hours, in case of system or database unavailability an entire day would be skipped without performing an appropriate logging. Thus, our periodically repeated action wakes up every hour, checks if an entire day is completed by comparing the wake-up time with the time of the current node, and if condition is false it stops. This is rendered with the classical `do.when` APOC procedure.

When instead the wake-up time indicates that an interval has expired, a simple Cypher query implements the management of the action logging process, as discussed above. Note that the Summary node used in the procedure is still linked to the Alert nodes of the last period, whereas the new Current node has no connections to any Alert node; new Alert nodes will be generated and linked to it in the next period.

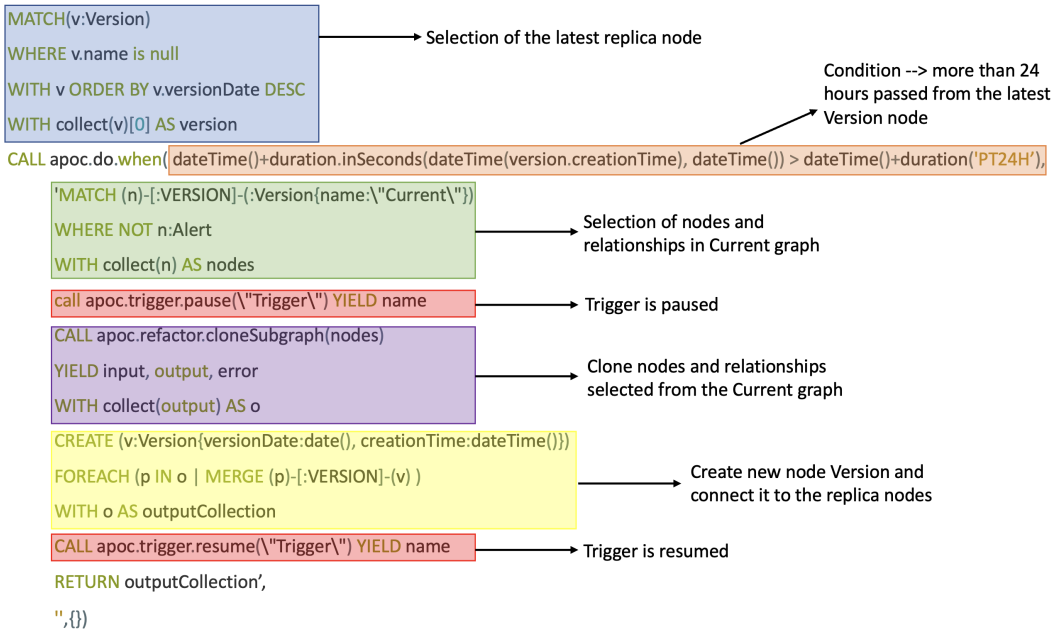


Figure 6.4: Creation of the Essential Summary. The APOC-based periodic repeat query activates at every hour. The first query checks that more than 24 hours have elapsed since the last creation of a new summary node. The second query is activated only when a new node must be created and appended to the chain of summary nodes.

Periodic execution for the Total Replication model. Although the "wake-up" mechanism is the same, the Total Replication model in much more complex, and it is just sketched below. Once the condition is verified, the triggering system is paused, since making duplicates of nodes corresponds to the insertion of new nodes that can easily trigger the execution of Alerts. APOC trigger collection provides a procedure that pauses the triggers `apoc.trigger.pause(name)`, where `name` identifies the trigger to be paused.

Then, all the nodes and relationships that are connected to the node “Version” and define the current graph are retrieved. The `apoc.refactor.cloneSubgraph()` procedure cloneS nodes with their labels and properties, as well as clone the given relationships. The copies of nodes and relationships are then connected to a new "Version" node (whose properties hold the datetime of its creation). Finally, the triggers stopped earlier are resumed right after the completion of the duplication of the Current graph, through the `apoc.trigger.resume(name)` procedure. Figure 6.4 shows the Cypher query in the context of the APOC periodic repeat procedure.

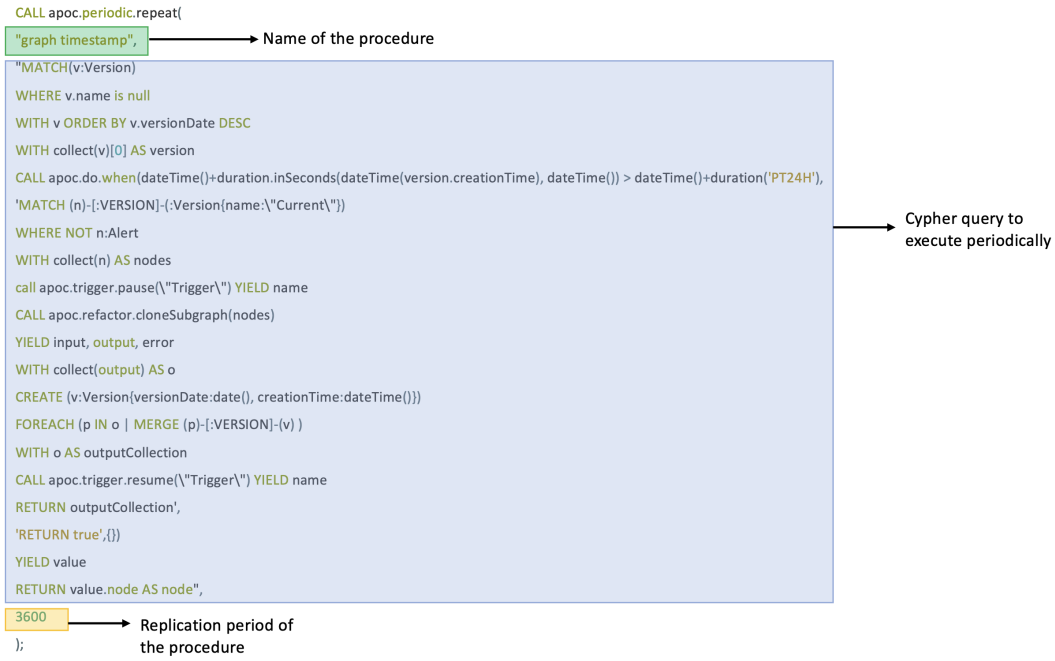


Figure 6.5: Creation of the graph clones required by the Total Replication model

This approach allows linking each period of the knowledge graph to include nodes and arcs that were present at the specific time when the complete copy was performed, but it is very heavy. The biggest drawbacks of replicating a graph is the storage space required to maintain all the copies. In addition, the copy occurs as a big transaction (with triggers disabled). Thus, for a long time, the system is unavailable. We do not discuss this option further, as in the testbed we will adopt the Essential Summary model.

6.5. Discussion of alternatives

Versioning has been a topic of open discussion for a long time. There is no precise, community-agreed solution for Neo4j. Thus, version control must be considered as part of the data modeling process. Beyond our current solution, other options are suggested by the Neo4j community for modeling the version control system, which are next reviewed,

Versioner-core. The Versioner-core[1] plugin is a collection of procedures, aimed to help manage the Entity-State model, by creating, updating, and querying the graph. These procedures define the so-called `Entity-State (ES) Data Model` and `Entity-State-R (ESR) Data Model`, which extends the first model by versioning also the relationships.

As shown by Figure 6.6 the data model is composed of the Entity nodes, created by the

[1] https://github.com/h-omer/neo4j-versioner-core

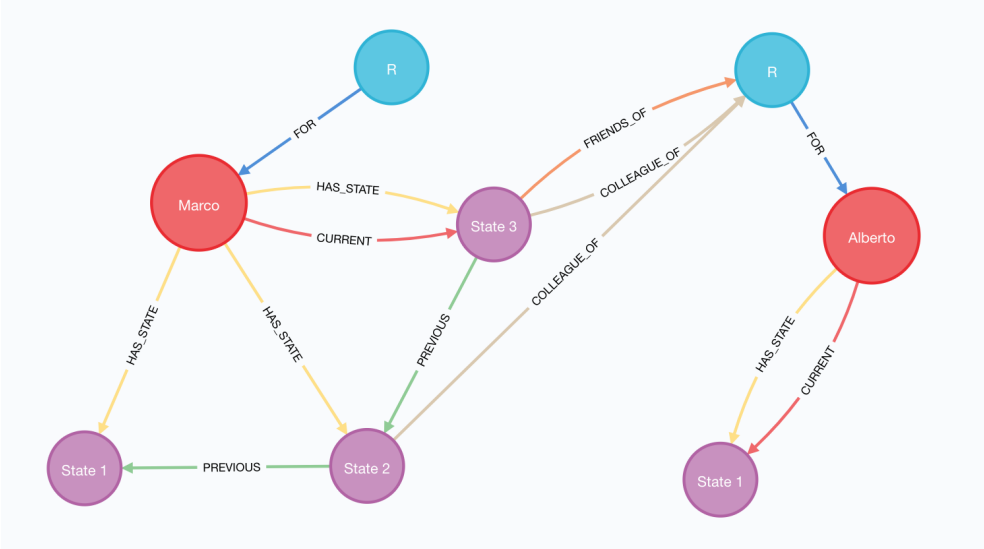


Figure 6.6: Data model of the Versioner-core plugin.

`graph.versioner.init`	Create an Entity node with it's R node and an optional initial State.
`graph.versioner.update`	Add a new State to the given Entity.
`graph.versioner.patch`	Add a new State to the given Entity, starting from the previous one. It will update all the properties, not labels.
`graph.versioner.get.current.state`	Get the current State node for the given Entity.
`graph.versioner.rollback`	Rollback the current State to the first available one.

Table 6.1: List of all the procedures and functions of APOC Conditional

user through a given Label, and the *State* nodes, managed by the Graph Versioner, which can be seen as the set of mutable properties which regards the Entity, which possesses only immutable properties. In the ESR model, when an Entity node is created, also its own R node is created, i.e., the Entity's access point for its own incoming relationships – considering also the version of the edge. The most relevant procedures provided by this plugin are shown in Table 6.1.

While this approach has its benefits, it also has some potential drawbacks. First of all, it can add significant complexity to the system as it involves a fair amount of setup and configuration, making it difficult to maintain and troubleshoot the database. Moreover, depending on the size and complexity of the data being versioned, queries may take longer to execute, and the database may require more resources to operate efficiently.

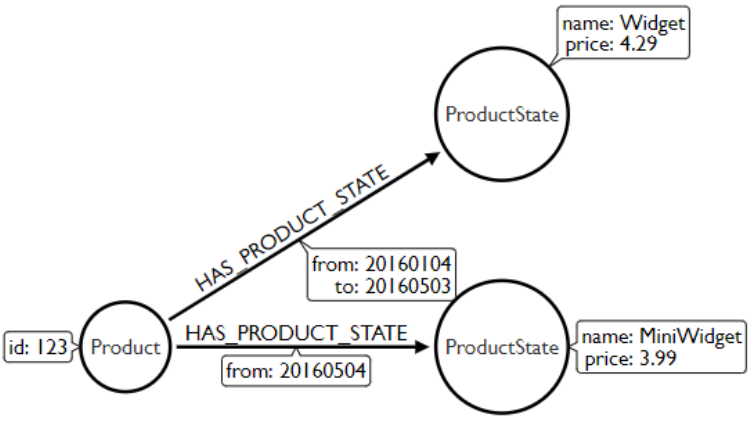


Figure 6.7: Time-based versioning modeled in a graph, including versioning on the relationships.

Time-based versioning. Time-based versioning is founded on two basic principles:

- Separation between the object and the state – these are linked by a relationship.
- Capture of change times within the relationship property linking these two entities.

Based on these two concepts, each node holds outgoing relationships with other entities that represent its own state. The state node contains properties with updated values.

In this approach, the focus is on the relationships that connect the state of the node and the node itself. As we can see in Figure 6.7, edges capture information about when the state was valid with `from` and `to` properties.

Just by separating out an object from the state, we are able to capture a lot of information about changes and pull back information depending on the time filter.

At times, we might be interested in versioning the relationships between objects. In this way, we would like to understand how entities are or were connected to other entities and how that changed over time. The principles behind versioning relationships are pretty much the same as those of versioning object states. In fact, we will need to provide a time range for when that relationship was activated, if relevant.

This solution brings the advantage of capturing all the changes, allowing us to trace back in time according to the questions we are looking to answer. At the same time, it brings the disadvantage of making queries much more complex; note that the mechanism does not provide default indexing on relationship properties, thus requiring further database tuning.

7 | Proof of Concept

In this chapter, we describe a proof-of-concept that was progressively developed to field-test an implementation for the concepts introduced in this thesis, so as to demonstrate their feasibility; some proof-of-concept aspects have been anticipated in the previous chapters. As the Essential Model described in the previous chapter was developed at the end of the thesis, the proof-of-concept still refers to a `current node` connected to the nodes of the knowledge graph. We move from Figure 7.1, illustrating the partitioned knowledge graph for our COVID-19 scenario.

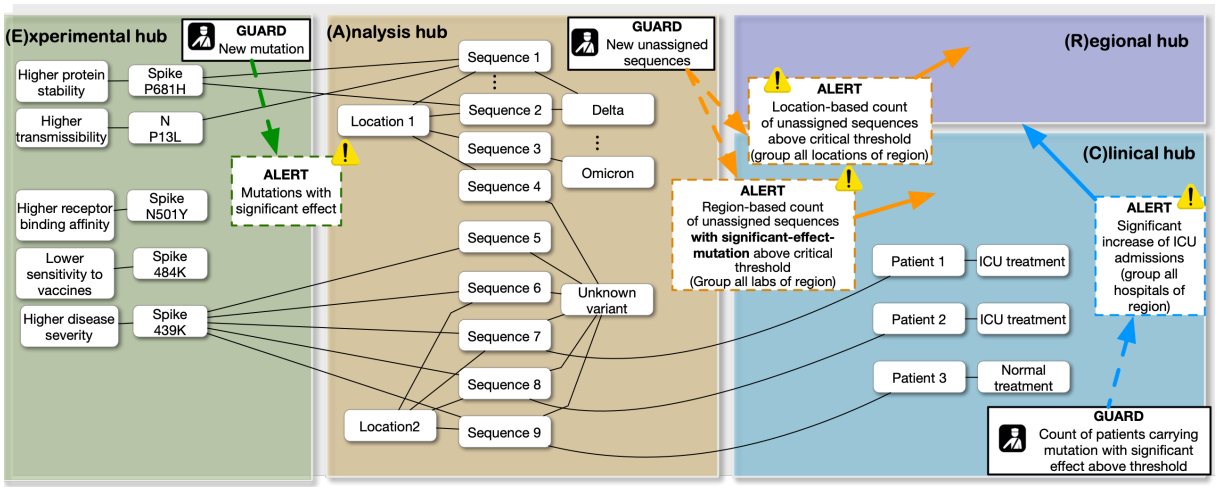


Figure 7.1: Overall structure of the analyzed COVID-19-related case study.

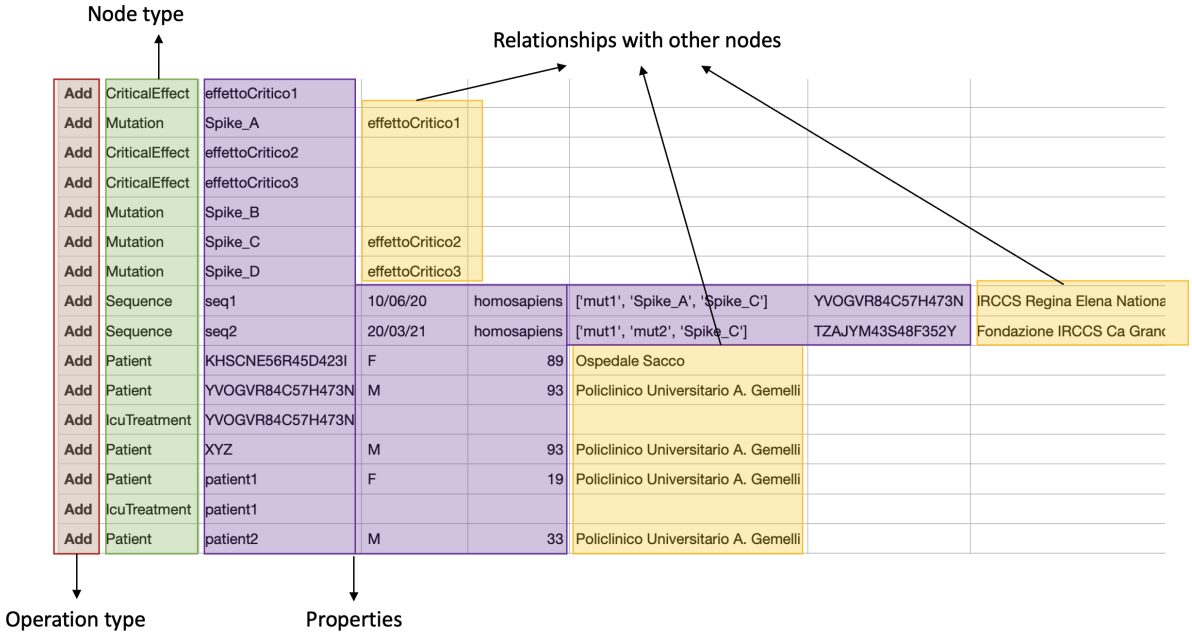
7.1. Node Generator

We start by illustrating how the example graph was populated. As described in Section 5.1.1, we imported data through `CSV` files to simulate a number of real-world scenarios. These files have a well-structured schema that can serve as a template for users who are not familiar with the technology.

Each row in the file represents a new entity to be either added or deleted. The first column specifies the operation to be performed: `Add` indicates that a node needs to be created

in the graph, while `Delete` means that a node with the specified characteristics must be removed from the graph.

The second column of each row indicates the type (i.e., label) of the node to be added or deleted. The following columns contain the values of the relevant properties for each type of node as well as references to the connected nodes. Figure 7.2 presents an example of a CSV file that served this purpose.



Add	CriticalEffect	effettoCritico1					
Add	Mutation	Spike_A	effettoCritico1				
Add	CriticalEffect	effettoCritico2					
Add	CriticalEffect	effettoCritico3					
Add	Mutation	Spike_B					
Add	Mutation	Spike_C	effettoCritico2				
Add	Mutation	Spike_D	effettoCritico3				
Add	Sequence	seq1	10/06/20	homosapiens	['mut1', 'Spike_A', 'Spike_C']	YVOGVR84C57H473N	IRCCS Regina Elena Nationa
Add	Sequence	seq2	20/03/21	homosapiens	['mut1', 'mut2', 'Spike_C']	TZAJYM43S48F352Y	Fondazione IRCCS Ca Granc
Add	Patient	KHSCNE56R45D423I	F	89	Ospedale Sacco		
Add	Patient	YVOGVR84C57H473N	M	93	Policlinico Universitario A. Gemelli		
Add	IcuTreatment	YVOGVR84C57H473N					
Add	Patient	XYZ	M	93	Policlinico Universitario A. Gemelli		
Add	Patient	patient1	F	19	Policlinico Universitario A. Gemelli		
Add	IcuTreatment	patient1					
Add	Patient	patient2	M	33	Policlinico Universitario A. Gemelli		

Figure 7.2: Example of the structure of a CSV file for the import of data.

As previously mentioned, certain nodes are considered fixed in the graph (i.e., Lab, Hospital, Variant, and Region). These are assumed to be present prior to the creation of any additional nodes and are expected to remain unchanged over time. Thus, they are not shown in Figure 7.2.

We developed a Java driver called `NodeGenerator`, used to directly connect to the Neo4j instance. Once the application establishes a connection with Neo4j, it executes a function containing a Cypher query to check for the existence of a node called *"Current"*. This node characterizes the graph on which changes are made, and it is always up-to-date.

If the query returns a non-empty result, then the *"Current"* graph already exists, and the following node creation or deletion operations will be performed on that graph. Else, if no *"Current"* node is found, the database is in its initial state. In this case, before proceeding with data import, a central node called *"Current"* needs to be created.

Only in the second case (i.e., empty database), an additional CSV file containing only static nodes is provided to the generator. This file contains the specifications about those nodes that represent the fundamental entities in the graph (i.e., Hospitals, Regions, Labs, Variants).

In both cases, the generator proceeds by reading the CSV with input notes. For each line, it checks the first column value to determine if it is a creation or deletion operation. Based on that, the driver performs the appropriate operation. The application reads the value in the second column, indicating the node's entity. Based on this value, the computation enters a specific function that manages the Cypher queries for each node type. The application provides specific functions to handle the queries specific to each type of node.

Each function defines a Neo4j transaction that includes numerous checks that need to be made during the creation or deletion of a node. Let us describe the example of Sequence node insertion Sequences are one of the central entities in the use case. A node Sequence is defined by an `accession` name and holds a `collection date`, and an `isolate`. In addition, in the CSV file, we specify also the `variant` to which the sequence is assigned (to indicate a relationship with a Variant node), `mutations` that can be observed in that sequence (leading to relationships with Mutation nodes), the (optional) `host` from which that genomic sequence was extracted (to indicate a possible relationship with a Patient node), and the laboratory in which it was sequenced (to indicate the relationship with a Lab node).

When focusing on the host from which a sequence was extracted, it may happen that the individual is already a patient in a hospital in that region. In this case, the new node Sequence should have a relationship with the node Patient found in the graph. To achieve this, the `addSequence` function in our Java application first checks if a Patient with the same SSN as the Sequence's host already exists in the graph before proceeding with the query that creates the actual node. If the Patient is found, the query that follows must add a relationship with that particular sequence.

Figure 7.3 shows the implementation of `addSequence`. First, a new session is opened to execute all transactions in Neo4j. Following that, a Cypher query checks if a node Patient with the same SSN as the Sequence's host already exists in the graph. Based on the result of that query, another transaction is executed, and the query actually creates the node Sequence.

```
private void addSequence(String[] csvLine ) {
try(Session session = driver.session(builder().withDefaultAccessMode(AccessMode.WRITE).build())){
      Result result = session.run("MATCH (p:Patient:ClinicalHub {cf: '"+csvLine[6]+"'}) return p");

      if(result.list().isEmpty()) {
            session.writeTransaction(tx->tx.run("MATCH (v:Variant:AnalysisHub) where v.pangoLineage= '"+csvLine[8]+"' "
            + "MATCH (ve:Version{name:'Current'}) "
            + "MATCH mutations = (m:Mutation) where m.name in "+csvLine[5]+" "
            + "MATCH (l:Lab:AnalysisHub) where l.name= '"+csvLine[7]+"' "
            + "MERGE (l)<-[:SEQUENCED_AT]-(s:Sequence:AnalysisHub:New {accession: '"+csvLine[2]+"', collectionDate: '"+csvLine[3]+"',
            isolate: '"+csvLine[4]+"', mutations: "+csvLine[5]+", host: '"+csvLine[6]+"'})-[:BELONGS_TO]->(v)  "
            + " FOREACH (n IN nodes(mutations) | MERGE (s)<-[:FOUND_IN]-(m))"
            + "MERGE (s)-[:VERSION]-(ve)"));
            }
            else {
            session.writeTransaction(tx->tx.run("MATCH (v:Variant:AnalysisHub {pangoLineage: '"+csvLine[8]+"'}) "
            + "MATCH (ve:Version{name:'Current'}) "
            + "MATCH mutations = (m:Mutation) where m.name in "+csvLine[5]+" "
            + "MATCH (l:Lab:AnalysisHub {name: '"+csvLine[7]+"'}) "
            + "MATCH (p:Patient:ClinicalHub {cf: '"+csvLine[6]+"'})  "
            + "MERGE (l)<-[:SEQUENCED_AT]-(s:Sequence:AnalysisHub:New {accession: '"+csvLine[2]+"', collectionDate: '"+csvLine[3]+"',
            isolate: '"+csvLine[4]+"', mutations: "+csvLine[5]+", host: '"+csvLine[6]+"'})-[:BELONGS_TO]->(v) "
            + "MERGE (p)<-[:AFFECTED]-(s) "
            + "FOREACH (n IN nodes(mutations) | MERGE (s)<-[:FOUND_IN]-(m))"
            + "MERGE (s)-[:VERSION]-(ve)"));

            session.writeTransaction(tx->tx.run("CALL apoc.log.info ('CREATE NODE SEQUENCE %s', ['"+csvLine[2]+"'])"));
                  }
            }
}
```

Figure 7.3: Extract of the `addSequence` function that defines the Cypher query for the creation of nodes Sequences and create relationships with other nodes.

In the Cypher query, we first match all the nodes already in the graph that have a relationship with the sequence to be added. When we have all their references, we create the node Sequence whose properties match the values specified in the CSV file. In the same transaction, we also use MERGE to create edges that indicate the relationships with the previously retrieved nodes.

In addition to the query just explained, the application `NodeGenerator.java` contains other Cypher queries that correctly implements all the nodes and relationships into the graph according to our data model. Figure 7.4 shows the result of the creation of the nodes in the "Current" graph in Neo4j.

7.2. Guards and Alerts implementation

To resume what has been said in the preceding chapters, `Guards` can be considered as predicates that monitors the consequences of event on a subset of nodes, returning a Boolean value in case some interesting condition has occurred. `Alerts` instead can identify (and then describe) a critical situation that may arise; they are evaluated only when the guard is true. Interpreting these functionalities as an ECA rule, the event that fires the trigger in our case is usually the creation of a node, the Guards represent the condition part of the rule, and Alerts consist in the action part.

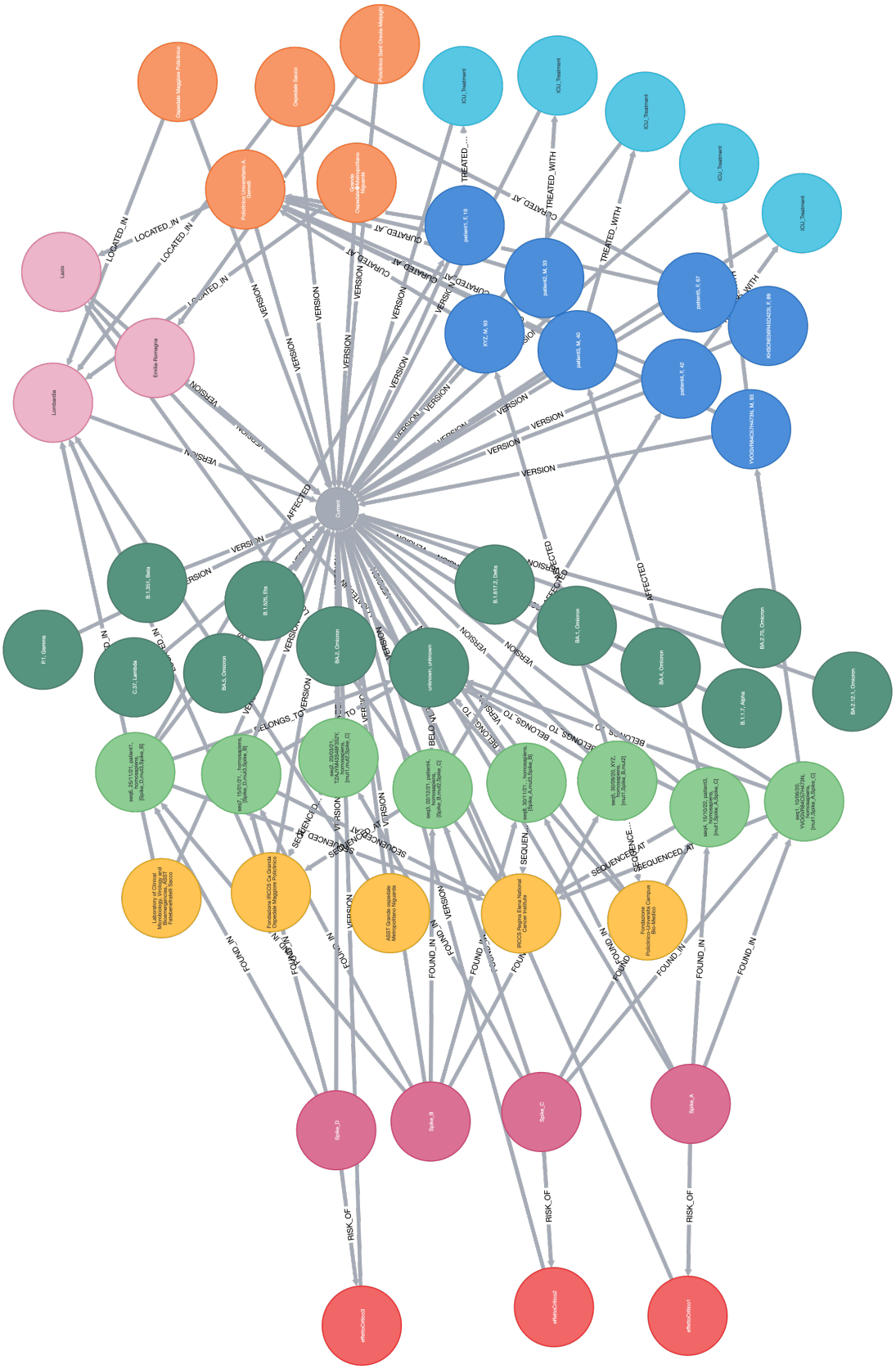


Figure 7.4: The "Current" graph of the described use case in Neo4j.

In our use case study, we defined three Guards and four Alerts, described in Table 7.1.

RULE	EV.	CONDITION	ACTION
NewCriticalMutation	Insertion of a new node	New node is of type Mutation	Alarm if region-based count of unassigned sequences with significant-effect-mutation context is above a critical threshold
SpreadingOfCriticalSequences		New node is of type Sequence and it is of type Unassigned (i.e., no correspondence to a Variant type)	Alarm if the region-based count of unassigned sequences with critical-effect-mutation context is above a critical threshold
SpreadingOfUnassignedSequences			Alarm if the region-based count of unassigned sequences is above a critical threshold
RiseOfCriticalClinicalConditions		New node is either of type Patient or Sequence or ICUTreatment and count of patients carrying mutation with significant effect is above a threshold	Alarm if there is a significant increase of ICU admissions in the region considered an increase of ICU

Table 7.1: Description of the Event-Condition-Action systems using our defined Guards and Alerts.

As mentioned in Section 5.2, the creation and insertion of a new node are captured by the `apoc.trigger` procedure; its execution returns as value either a NULL, when the trigger has no side effects, or the identifier of the Alert node, when the trigger has a node creation as a side effect. the execution of the trigger is `afterAsynch` to dente that it runs asynchronously after the completion of the transaction that has caused the event.

The Condition and Action parts of the reactive rule are connected by a conditional query using the `apoc.do.when` procedure. Guards and Alert are queries in Cypher; the former returns a Boolean value after including a condition whose first term includes a test on the node type label, thus anchoring a trigger to an event over a specific node type (thereby giving the same effect as targeting a trigger to a table in the relational model); the latter is a query that may create a node of type Alert when other conditions in the graph are met.

We may need to implement Guards and Alerts by creating custom functions, which are

crafted for expressing complex conditions. These user-defined functions are then called into the `apoc.do.when` procedure inside the trigger. The result is the definition of a number of functions, all directly called in Neo4j as an extension of Cypher, as described in the code below. The procedures are deployed in a Github repository, linked at the end of the thesis.

Alert nodes are associated with a new node, whose common information is defined by a triple (<type, SubType, timestamp>). Each node is associated with its *type*, which is defined by the label `Alert`. Then another label, *SubType*, is used to differentiate between the various types of Alert nodes. Finally, the *timestamp* represents the time at which the alert node is created. Each Alert node is further characterized by additional properties which are defined by the rule designer, who should consider all the information that may be interesting to later analyze the critical condition; these properties are saved at the time when the Alert node is created.

We next describe the implementation of the four reactive rule in our use case.

7.2.1. Experimental Hub's reactive rule.

The reactive rule in the Experimental hub includes: (1) a Guard that filters new nodes by label "Mutation" and (2) an Alert that checks if the new Mutation has a critical effect. The Guard can be easily expressed in Cypher without the need of implementing a new function, whereas for the Alert we define a new function called `criticalEffect`.

The alert reports an alarm condition in case the newly added Mutation node has a relationship with a node of type CriticalEffect. If the query that searches for this relationship returns a non-null result, then another query is going to create a node of type `Alert`, associated with its subtype `NewCriticalMutation` (or `NewCM`) and to its creation time. The additional properties of this node include the mutation that caused the creation of the alert (`Spike N501Y`) and its critical effects `[high receptor binding affinity, lowered sensitivity to vaccines]`, while in the node we include the alert description property: "New mutation with a critical effect has been added" (see Figure 7.5). Next, we report the implementation of the described reactive system.

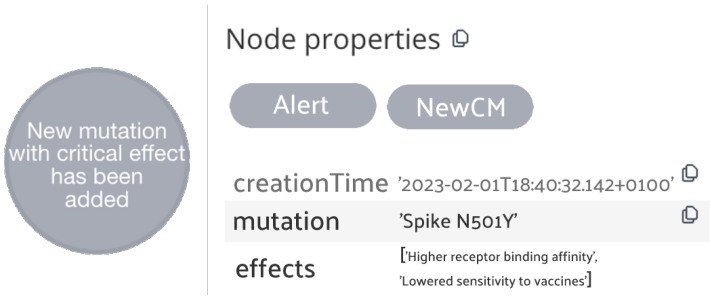


Figure 7.5: Example of node Alert for Experimental Hub.

```
CALL apoc.trigger.add('Experimental_Trigger','UNWIND
$createdNodes AS node CALL apoc.do.when(node:Mutation,
reactkg.criticalEffect(node.name)), {node:node}) YIELD value RETURN
*', phase:'afterAsync');
```

7.2.2. Analysis Hub's reactive rules.

In this knowledge hub, two reactive rules are defined. They share the same Guard, which firstly controls that the new node is a Sequence node, then verifies that the new Sequence does not belong to known Variants. To perform the second check we defined a new function called `unassignedSequence`. This function represents the Guard related to the Analysis Hub. Here, a query looks for the existence of the relationships between the just created Sequence node and the Variant node of type "unknown", indicating sequences whose variant has not been assigned yet. If the query returns a result, then the Sequence has not been assigned to any variant and thus the function returns TRUE, indicating a condition to be further investigated by the relative Alert.

For the Alerts of this knowledge hub, we created two new functions. The first define the Alert named *SpreadingOfCriticalSequences*. The first function – called `sequenceWithCriticalEffect` – checks if the number of unassigned sequences is above a critical threshold, considering all the laboratories of the region within which the sequence was collected. The function includes a query that evaluates the condition just described.

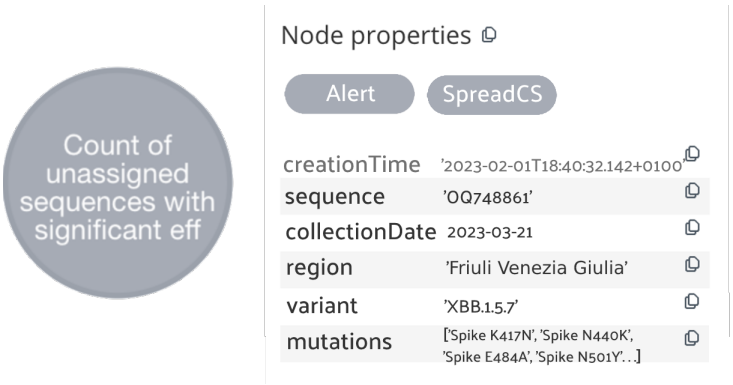


Figure 7.6: Example of Alert node *SpreadingOfCriticalSequences* for Analysis Hub.

```
CALL apoc.trigger.add('Analysis_Trigger2','UNWIND
$createdNodes AS node CALL apoc.do.when( node:Sequence
and reactkg.unassignedSequence(node.accession),
reactkg.sequenceWithCriticalEffect(threshold, node.accession)),
{node:node}) YIELD value RETURN *', phase:'afterAsync');
```

The second function – called `regionalUnassignedSequences` – returns a string defining the query for the creation of an alert node when the number of unassigned sequences harboring mutations with a critical effect (for the given region) is greater than the set threshold. As shown in Figure 7.7, this Alert is named *SpreadingOfUnassignedSequences*

The Alert nodes resulting in this trigger systems, in addition to the labels defining the type and their subtype, it will hold a property referencing the sequence (e.g., `OQ748861`), the collection date (`2023-03-21`), the region of its collection (`'Friuli Venezia Giulia'`), its variant (t 'XBB.1.5.7') and the array of mutations that compose the sequence `['Spike K417N', 'Spike N440K', 'Spike E484A', 'Spike N501Y']`. Figures 7.6 and 7.7 show a representation of the two types of Alert nodes. Next, we show the implementation of the two triggers.

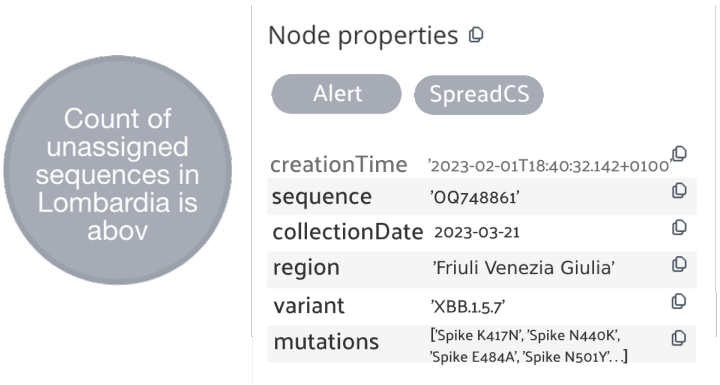


Figure 7.7: Example of Alert node *SpreadingOfUnassignedSequences* for Analysis Hub.

```
CALL apoc.trigger.add('Analysis_Trigger1','UNWIND
$createdNodes AS node CALL apoc.do.when( node:Sequence
and reactkg.unassignedSequence(node.accession),
reactkg.regionalUnassignedSequences(threshold, node.accession)),
{node:node}) YIELD value RETURN *', phase:'afterAsync');
```

7.2.3. Clinical Hub's reactive rules.

Two functions were developed for this hub. The function that implements the Guard is called `patientsAboveThreshold`. It aims to assess if the number of patients in a region carrying mutations with critical effects exceeded a certain threshold after the creation of a given node. The custom function accepts arguments for the threshold (as an Integer) and for the ID of the node just created.

Given the node ID, it queries the graph to retrieve its type which will help to identify the relationships and the nodes that need to be involved in the query that retrieves the number of patients with sequences (holding critical mutations).

Then, a query counts all the patients with sequences with a critical mutation (in a particular region). If the number of patients is greater than the threshold (given in input to the function), then the function returns TRUE, indicating a condition to be further investigated with the relative Alert.

The function used to define the Alert query is called `patientIncrease`. It builds alerts related to the Clinical Hub and is the most complex function among all the routines developed for the alerts. The alert checks whether a particular region shows a significant increase in patient admissions to the ICU. To successfully check this, we rely on historical records of what happened over time. As discussed in Section 6.3.2, our approach uses the

Essential Summary approach.

The function retrieves the number of patients in the ICU in the current graph, for a specific region. Then, it retrieves the number of patients in the ICU at the previous time point, using the Essential Summary. In our case, 'previous time point' means the state of the graph 24 hours earlier. Note that – when we first deploy this type of trigger – we make sure to save the number of patients present in each region in the current graph by adding this values as properties of the last created Essential Summary. From there, we can then retrieve the number of ICU patients based on the number of new and deleted nodes involved after that point in time.

The arguments of this function represent the ID of the node just created (which fired the trigger) and the percentage that defines a "significant increas" according to the user.

Note that this trigger may be fired by three different types of nodes (i.e., Sequence, Patients, or ICUTreatment). Thus, the function first identifies the type of node, since the following procedure differs based on this. Retrieving the type of node allows also us to identify the region we need to take into consideration. Then, we query the Current graph in order to retrieve the number of patients in ICU in the Current graph for that region.

Once we have the two values (the current and the past number of patients in ICU), we check whether their difference exceeds the percentage in input. If this condition is verified, then the function returns to the trigger a query that creates an Alert node (shown in Figure 7.9) whose subtype is called *RiseOfCriticalClinicalConditions*. This node is also defined by the description *"Significant increase in ICU"* and complemented by the additional properties: the ID of the node that fired the trigger (`<512>`), the number of patients in ICU in that moment for the region considered (`10`) and the region of reference (`'Friuli Venezia Giulia'`). Finally, we show the implementation of the trigger.

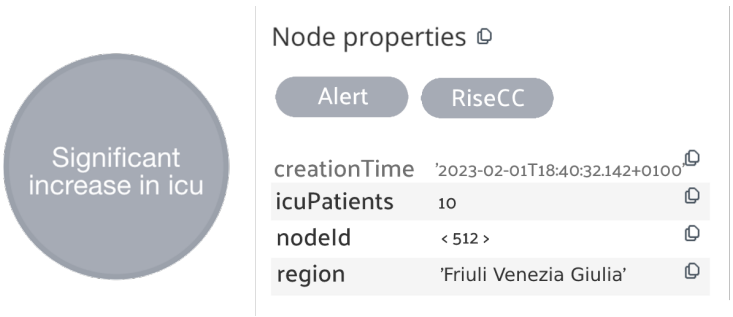


Figure 7.8: Example of Alert node *patientsAboveThreshold* for the Clinical Hub.

```
CALL apoc.trigger.add('Clinical_Trigger','UNWIND $createdNodes
AS node CALL apoc.do.when(node:Patient or node:ICUTreatment
or node:Sequence) and reactkg.patientsAboveThreshold(threshold,
toString(apoc.node.id(node))), reactkg.icuIncrease(toString(apoc.node.id(node)),
threshold), {node:node}) YIELD value RETURN *',
phase:'afterAsync');
```

7.3. Rule Enactment

The aim of this section is to present the functioning of the reactive rules through a step-by-step example. We consider the example proposed in section 4.2. As we briefly see, this reactive rule, can be classified as a multi-state and inter-hub rule. Thus, the involved queries will match path that go through multiple hubs but in the alert component will consider multiple periods of observation. Following the definition of the reactive knowledge rules, this process starts with a modifying event, the creation of a new node Patient.

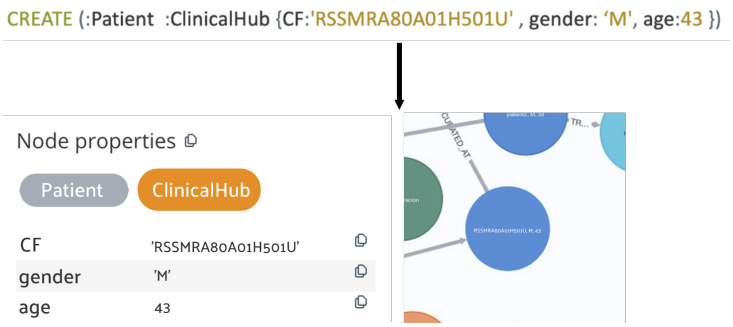


Figure 7.9: Illustration of the query performed for the creation of a new Patient, followed by the resulting node in the database.

The conditional component of the reactive rule, the Guard follows the creation of this new node. At first, it verifies the presence of the label *Patient* that identify the new node. Then it moves on with its conditional part where it will check if this new patient is connected to a viral sequence belonging to an unknown variant. This two conditions are verified since the Guard query matched a path that connect the new Patient to an unassigned Sequence.

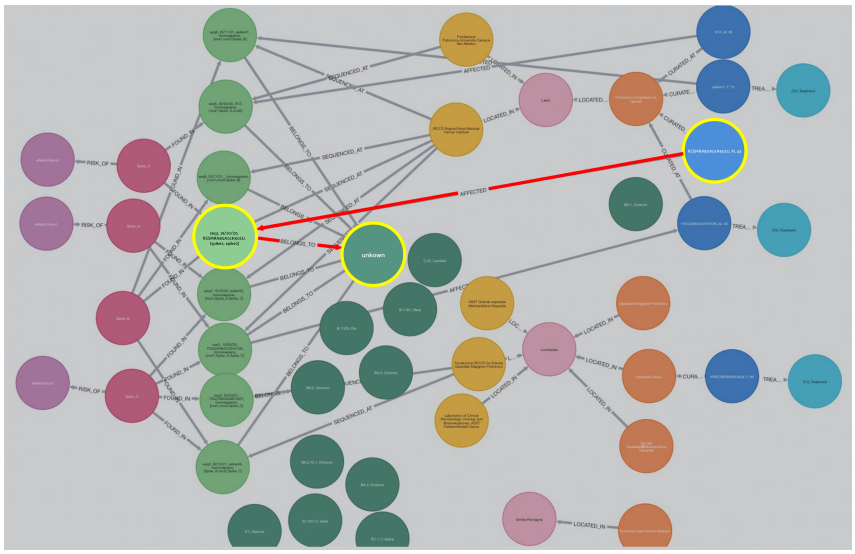


Figure 7.10: The new node Patient has a relationship with a viral sequence whose variant is yet to assigned by the scientific community.

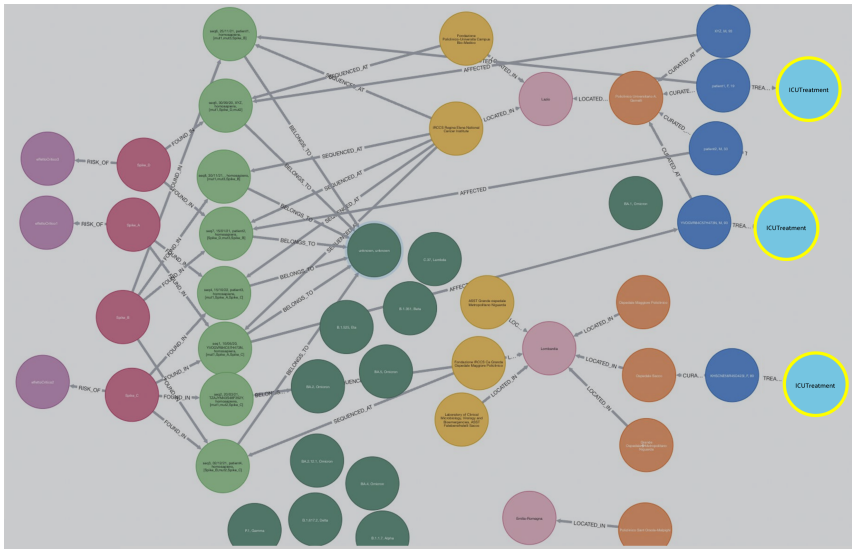


Figure 7.11: We illustrate the `ICUTreament` nodes found in the current knowledge graph; three patients are in the intensive care unit.

Now that the conditional component of the rule is verified we move to the action part. The Alert component in this example wants to know if after the modifying event, the number of Patients in ICU is greater by 10% compared to the situation that the knowledge presented the previous day. The Alert at first controls the number of patients in ICU currently presents in the graph. For the sake of this example we are going to consider a small number of patients in ICU, nonetheless the operations performed remain the same even when dealing with a large amount of data.

As highlighted in Figure 7.11, the query finds 3 patients currently cured in the intensive care unit. The query then proceeds by retrieving the old number of patients in ICU in the Summary node whose *date* property report the date of the previous day. From there the query finds the Alert node with the same *subtype* of the considered reactive rule and looks for the property indicating the number of patients in ICU in that day. The number here retrieved and the actual number of ICU patients are compared. As the number of ICU patients yesterday was 2, the condition is true, and a node of type Alert needs to be created, with the label and the properties that are designed for that reactive rule.

7.4. Deployment

To deploy our proof-of-concept on a testbed, a Docker (`https://www.docker.com/`) image was used. Docker allows for having a portable, isolated, and scalable solution, which evolved with our study. Likewise, it provides a reliable solution making it easier to have a safe and usable working environment. The docker image containing an instance of Neo4j (version 4.4.5) was hosted and run on the server of the GeCo Lab at Politecnico di Milano.

Neo4j offers drivers that facilitate connecting to the database and developing applications to perform CRUD (create, read, update, delete) operations on the graph. To facilitate the creation of new nodes in the Neo4j database, we developed a Java driver application called `NodeGenerator.java`. This driver connects to the Neo4j server using the binary Bolt protocol and was implemented using Java 15, along with the neo4j-java-driver (version 4.4.9 version of the series). Data were then imported into Neo4j using structured CSV files that defined the specifications of the nodes. The driver relied on a CSV-reader dependency (version 4.1) to read the data files and create the corresponding nodes in the graph.

On an application-specific level, the APOC library played an important role in the implementation of reactive rules and periods of observation. As previously explained, APOC is a standard utility library for common procedures and functions. The reference version of the employed plugin is 4.4.0.6. The library needs to be installed in each database project: this can be done using the "Plugins" folder of the database.

Similarly, it was possible to extend the functionalities of Neo4j to model the Guard and Alert functions. In the implementation of the reactive rule, together with the APOC trigger procedure, we created a plugin for Neo4j containing the functions of the two key components of reactive rules. The functions were created through a custom plugin project, written in Java, and compiled into a JAR file. It was then deployed to the database by saving the JAR file into the plugins directory on the database server. All the components involved in the production of the proof of concept (such as the CSV

files, NodeGenerator, and the Guard-Alert plugin) can be found on a dedicated GitHub repository at the following link `https://github.com/Alessia-G/ReactKG`.

8 | Conclusions and Future Work

In this chapter, we summarize the main results of this thesis, we discuss its limitations, and indicate directions for future work.

8.1. Results

Graph databases have become increasingly popular in recent years due to their ability to store complex and highly interconnected data; they are increasingly used as the underlying technology for managing large knowledge graphs. In this thesis, we show that graph databases can be also used for supporting an extension of knowledge graph management, called **Reactive Knowledge Management**, based on the new concept of reactive knowledge rule. This thesis comprises several contributions both at the conceptual and implementation levels.

On a conceptual level, we propose the use of **Partitioned Knowledge Graphs** to represent shared knowledge. This approach enables us to manage large amounts of data in a more structured and efficient manner by grouping the data into hubs that share portions of the knowledge graph, typically though edges that interconnect the partitions.

A crucial aspect of this thesis is the definition of **Reactive Knowledge Rules**. They are triples $< Event, Guard, Alert >$ that introduce the Event-Condition-Action (ECA) paradigm in this setting. while events monitor simple operations (e.g., node additions and deletions), the Condition-Action pair enables us to model conditions of arbitrary complexity to best fit the use cases at hand. The Guard component specifies simple conditions that must be met for the rule to be further considered, and the Alert component then explores a situation of arbitrary complexity, summarizing the critical situation observed in the knowledge base in the form of an Alert node.

After the recent consolidation of the research on property graphs with PG-Schemas and PG-Keys, reactive rules contain all the ingredients that can be distilled into a proposal for **PG-Triggers**; this is our most immediate future objective. PG extensions are finalized towards the development of standard Graph Query Language (GQL), whose aim is to

standardize and enrich the functionalities of the graph query languages present nowadays on the market, with the ambition of establishing a new universal graph query language, just like SQL has been for relational databases.

Along with Reactive Knowledge Rules, we also use Alert nodes to model critical situations. By utilizing Alert nodes, we can also compare alert-specific information across time intervals, so as to **identify trends and patterns over time intervals**. The use of Alert nodes and Reactive Knowledge Rules work together to create a comprehensive monitoring mechanism for graph databases, enabling us to identify important changes and take appropriate action.

The need of monitoring multi-state conditions in active rules, along the classical notion of comparing old and new states, has brought us to define periods of observation and to support a mechanism for inspecting relevant aspects of the knowledge base in these states states. We first proposed a naive approach consisting in building a complete copy of the graph database. Due to the complexity and low performance of this solution, we eventually explored a new solution: the **Essential Summary**, a model which keeps track of the Alert nodes progressively added during the execution of reactive rules.

We then designed a mechanism for building the Essential Summary, interpreted as the **log of the reactive rule engine**; as in a log, nodes representing time intervals are chained in a sequence whose top node is the current state. Coupled with a thorough design of the Alert nodes, the Essential Summary model allow us to create expressive and complete queries across multiple states of the graph; by traversing the chain, it is possible to extract relevant data distributions and explore trends and patterns.

On a practical level, we mapped all the aspects of reactive processing rules management on top of the **Neo4j graph database system**. Due to the limited functionalities of the query language Cypher, we were forced to implement some mechanisms through APOC functions, an external library of No4j which is jointly contributed by a community of developers. Thanks to this library, we managed to express reactive rules as database triggers, as well as express the conditional execution of the Guard-Alert mechanism in a readable and simple way; APOC functions are also used for implementing the logging activities.

To test and validate our proposed solutions, we collected all the software components in a **Proof of concepts**, including a simple partitioned knowledge graph and four reactive rules which explore most of the options supported in our theory; the software is loaded on a GitHub repository, bundled into a Docker image, and deployed to a Docker platform, which is available on one of the `GeCo` server of our laboratory.

8.2. Limitations

In this thesis, we have only discussed reactive rules for managing node creations, but clearly rules can also **monitor node deletions**; these have a dual function of revealing *negative* changes and have been omitted so far, just for the lack of time. In addition, reactive rules could also monitor the creation or deletion of relationships between nodes; however, nodes belong properly to specific knowledge hubs, whereas edges may interconnect hubs. Hence, a greater focus on the creation and deletion of nodes gives a better design of how reactive processing is distributed among knowledge hubs.

Our management of state changes has considered two extreme solutions, one consisting of a full copy of the knowledge graph at each new temporal interval, and one keeping only the alert nodes. These solutions are extreme cases that have been developed rather fast, at the end of the thesis, but many other solutions are possible, including a **full trace of node additions and deletions**, which may be used to support a stronger notion of old and new states. It should be noted that a complete copy of the knowledge base, allowing a richer management of temporal queries, goes beyond the scope (and focus) of the thesis, which is concentrated on reactive processing.

We did not manage the **creation or dropping of a rule** in the knowledge base system. Our expectation is that the current Essential Model is robust to rule creation and deletion, as new node alert of given sub-types will pop-up just after the creation of the knowledge rule and will cease to be created after its removal, but we have not explored this aspect in all its consequences. For instance, after rule removal, we may decide whether to keep track of alert nodes associated with removed rule, thereby keeping an historical trace of alert conditions, or remove all of them and clean up the reactive knowledge base.

8.3. Future work

This thesis has just briefly addressed a number of dimensions whose deeper exploration remain open and will inspire future work. In particular, we will start by formally describing the reactive rule model and turn it into a robust proposal for building **PG-Triggers**, along the style which is used in the proposals of PG-Keys and PG-Schemas. We intend to give priority to this aspect, and build our proposal for PG-Triggers in the form of a research article, which will probably include only some of the aspects of the thesis (e.g., it will not deal with knowledge partitioning).

Then, we will explore **design methods for knowledge rules**, and more specifically focus on the design of Alert nodes, that should accurately represent the main issues that

arise in managing the evolution of knowledge graphs. This would ensure that critical situations are accurately represented along a temporal dimension.

In addition to the proposed solutions, we will further extend the triggering model by including in the Alert part actions for **creating and deleting knowledge nodes and edges**. This will enable us to construct more complex computations that could potentially be cyclic, and make use of the power of reactive rules in cascading mode. This further opens up the design dimension in the direction of properly design cascading rules from the perspective of termination, confluence, and determinism of observations, to ensure their accuracy and reliability.

Several direction remain to be explored also in order to turn the test-bed into a working prototype and better demonstrate the various features of a reactive, partitioned knowledge graph.

From an engineering perspective, as we have just listed some alternative modes of implementing our solutions, it will be interesting to **benchmark each alternative solution against the preferred one**, as well as to evaluate how to optimize each component so as to improve the overall system performance.

A full prototype will also require **user-friendly interfaces** providing facilities for creating and deploying new active rules, for visualizing rule enactment over graphs, for exploring the Alert nodes in the current and past states, and for summarizing the reactive rules, e.g. by adding a rule dictionary.

Bibliography

[1] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9):62–71, 2021.

[2] Neo4j. `https://neo4j.com/`. Last accessed: April 11th, 2023.

[3] Adrian Paschke and Harold Boley. Rules capturing events and reactivity. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 215–252, 2009.

[4] Kapali P. Eswaran. Aspects of a trigger subsystem in an integrated database system. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 243–250, Washington, DC, USA, 1976. IEEE Computer Society Press.

[5] Jennifer Widom and Stefano Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1995.

[6] Moshé M Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438, 1975.

[7] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[8] Jan Hidders and Jan Paredaens. *GOAL, A graph-based object and association language*. Springer, 1994.

[9] Diogo Fernandes and Jorge Bernardino. Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In *Data*, pages 373–380, 2018.

[10] R Valdes. The competitive dynamics of the consumer web: five graphs deliver a sustainable advantage. *Gartner: Stamford, CT, USA*, 2012.

[11] Stefan Plantikow. Towards an international standard for the gql graph query language. In *W3C workshop in Berlin on graph data management standards*, 2019.

[12] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2246–2258, 2022.

[13] A Green, P Furniss, P Lindaaker, P Selmer, H Voigt, and S Plantikow. Gql scope and features. *ISO, Tech. Rep*, 2019.

[14] Chandan Sharma. Flux: From sql to gql query translation tool. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1379–1381, 2020.

[15] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W Hare, Jan Hidders, Victor E Lee, Bei Li, Leonid Libkin, Wim Martens, et al. Pg-keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2423–2436, 2021.

[16] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savković, et al. Pg-schema: Schemas for property graphs. *arXiv preprint arXiv:2211.10962*, 2022.

[17] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. Knowledge graphs. *ACM Computing Surveys (CSUR)*, 54(4):1–37, 2021.

[18] Fan Wu, Su Zhao, Bin Yu, Yan-Mei Chen, Wen Wang, Zhi-Gang Song, Yi Hu, Zhao-Wu Tao, Jun-Hua Tian, Yuan-Yuan Pei, et al. A new coronavirus associated with human respiratory disease in china. *Nature*, 579(7798):265–269, 2020.

[19] Yuelong Shu and John McCauley. GISAID: Global initiative on sharing all influenza data–from vision to reality. *Eurosurveillance*, 22(13), 2017.

[20] Eric W Sayers, Mark Cavanaugh, Karen Clark, James Ostell, Kim D Pruitt, and Ilene Karsch-Mizrachi. GenBank. *Nucleic Acids Research*, 47(D1):D94–D99, 2019.

[21] The COVID-19 Genomics UK (COG-UK) consortium. An integrated national scale SARS-CoV-2 genomic surveillance network. *The Lancet Microbe*, 1(3):E99–E100, 2020.

[22] Chaoran Chen, Sarah Nadeau, Michael Yared, Philippe Voinov, Ning Xie, Cornelius Roemer, and Tanja Stadler. CoV-Spectrum: analysis of globally shared SARS-CoV-

2 data to identify and characterize new variants. *Bioinformatics*, 38(6):1735–1737, 2022.

[23] Karthik Gangavarapu, Alaa Abdel Latiff, Julia L Mullen, Manar Alkuzweny, Emory Hufbauer, Ginger Tsueng, Emily Haag, Mark Zeller, Christine M Aceves, Karina Zaiets, et al. Outbreak.info genomic reports: scalable and dynamic surveillance of SARS-CoV-2 variants and mutations. *medRxiv*, 2022.

[24] Begum Cosar, Zeynep Yagmur Karagulleoglu, Sinan Unal, Ahmet Turan Ince, Dilruba Beyza Uncuoglu, Gizem Tuncer, Bugrahan Regaip Kilinc, Yunus Emre Ozkan, Hikmet Ceyda Ozkoc, Ibrahim Naki Demir, et al. Sars-cov-2 mutations and their viral variants. *Cytokine & growth factor reviews*, 63:10–22, 2022.

[25] Tommaso Alfonsi, Ruba Al Khalaf, Stefano Ceri, and Anna Bernasconi. Cov2k model, a comprehensive representation of sars-cov-2 knowledge and data interplay. *Scientific Data*, 9(1):260, 2022.

[26] Deepa Vasireddy, Rachana Vanaparthy, Gisha Mohan, Srikrishna Varun Malayala, and Paavani Atluri. Review of covid-19 variants and covid-19 vaccine efficacy: what the clinician should know? *Journal of Clinical Medicine Research*, 13(6):317, 2021.

[27] Ahmed I Abulsoud, Hussein M El-Husseiny, Ahmed A El-Husseiny, Hesham A El-Mahdy, Ahmed Ismail, Samy Y Elkhawaga, Emad Gamil Khidr, Doaa Fathi, Eman A Mady, Agnieszka Najda, et al. Mutations in sars-cov-2: Insights on structure, variants, vaccines, and biomedical interventions. *Biomedicine & Pharmacotherapy*, page 113977, 2022.

[28] Áine O'Toole, Emily Scher, Anthony Underwood, Ben Jackson, Verity Hill, John T McCrone, Rachel Colquhoun, Chris Ruis, Khalil Abu-Dahab, Ben Taylor, et al. Assignment of epidemiological lineages in an emerging pandemic using the pangolin tool. *Virus evolution*, 7(2):veab064, 2021.

[29] World Health Organization. Tracking SARS-CoV-2 variants. `https://www.who.int/en/activities/tracking-SARS-CoV-2-variants/`. Last accessed: April 11th, 2023.

[30] World Health Organization et al. Therapeutics and covid-19: living guideline, 31 march 2021. Technical report, World Health Organization, 2021.

[31] P Atzeni, S Ceri, S Paraboschi, and R Torlone. Database systems: Concepts, languages and architectures 2001.

List of Figures

List of Tables