ADRIANE DE MARTINI FONSECA

# REFACTORING RULES FOR GRAPH DATABASES

*REGRAS DE REFATORAÇÃO PARA BANCO DE DADOS BASEADO EM GRAFOS*

LIMEIRA

2015

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Tecnologia

ADRIANE DE MARTINI FONSECA

# REFACTORING RULES FOR GRAPH DATABASES

*REGRAS DE REFATORAÇÃO PARA BANCO DE DADOS BASEADO EM GRAFOS*

Dissertation presented to the School of Technology of the University of Campinas in partial fulfillment of the requirements for the degree of Master, in the area of Technology

*Dissertação apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestra em Tecnologia, na Área de Tecnologia e Inovação*

Supervisor/*Orientador*: Prof. Dr. Luiz Camolesi Jr.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DISSERTAÇÃO DEFENDIDA PELA ALUNA ADRIANE DE MARTINI FONSECA, E ORIENTADA PELO PROF. DR. LUIZ CAMOLESI JR.

________________________________

LIMEIRA

2015

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Faculdade de Tecnologia
Felipe de Souza Bueno - CRB 8/8577

F733r

Fonseca, Adriane de Martini, 1988-
Refactoring rules for Graph Databases / Adriane de Martini Fonseca. – Limeira, SP : [s.n.], 2015.

Orientador: Luiz Camolesi Júnior.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Tecnologia.

1. Banco de dados. 2. Teoria dos grafos. 3. Software - Refatoração. I. Camolesi Júnior, Luiz,1967-. II. Universidade Estadual de Campinas. Faculdade de Tecnologia. III. Título.

Informações para Biblioteca Digital

**Título em outro idioma:** Regras de refatoração para banco de dados baseado em grafos
**Palavras-chave em inglês:**
Databases
Graph theory
Software refactoring
**Área de concentração:** Tecnologia e Inovação
**Titulação:** Mestra em Tecnologia
**Banca examinadora:**
Luiz Camolesi Júnior [Orientador]
Luís Augusto Angelotti Meira
Elaine Parros Machado de Sousa
**Data de defesa:** 27-02-2015
**Programa de Pós-Graduação:** Tecnologia

**DISSERTAÇÃO DE MESTRADO EM TECNOLOGIA**

**ÁREA DE CONCENTRAÇÃO: TECNOLOGIA E INOVAÇÃO**

Refactoring Rules for Graph Databases

**Adriane de Martini Fonseca**

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

[illegible]

Prof. Dr. Luiz Camolesi Júnior
UNICAMP
Presidente

[illegible]

Prof. Dr. Luís Augusto Angelotti Meira
UNICAMP

[illegible]

Profa. Dra. Elaine Parros Machado de Sousa
USP

## ABSTRACT

The information produced nowadays does not stop growing in volume and complexity, representing a technological challenge which demands more than the relational model for databases can currently offer. This situation stimulates the use of different forms of storage, such as Graph Databases. Current Graph Databases allow automatic database evolution, but do not provide adequate resources for the information organization. This is mostly left under the responsibility of the applications which access the database, compromising the data integrity and reliability. The goal of this work is the definition of refactoring rules to support the management of the evolution of Graph Databases. The rules presented in this document are adaptations and extensions of the existent refactoring rules for relational databases to meet the requirements of the Graph Databases features. The result of this work is a catalog of refactoring rules that can be used by developers of graph database management tools to guarantee the integrity of the operations of database evolution.

Keywords: Graph Database, Data Refactoring, Evolutionary Databases.

## RESUMO

A informação produzida atualmente apresenta crescimento em volume e complexidade, representando um desafio tecnológico que demanda mais do que a atual estrutura de Bancos de Dados Relacionais pode oferecer. Tal fato estimula o uso de diferentes formas de armazenamento, como Bancos de Dados baseados em Grafos (BDG). Os atuais Bancos de Dados baseados em Grafos são adaptados para suportar automaticamente a evolução do banco de dados, mas não fornecem recursos adequados para a organização da informação. Esta função é deixada a cargo das aplicações que acessam o banco de dados, comprometendo a integridade dos dados e sua confiabilidade. O objetivo deste trabalho é a definição de regras de refatoração para auxiliar o gerenciamento da evolução de Bancos de Dados baseados em Grafos. As regras apresentadas neste trabalho são adaptações e extensões de regras de refatoração consolidadas para bancos de dados relacionais para atender às características dos Bancos de Dados baseado em Grafos. O resultado deste trabalho é um catálogo de regras que poderá ser utilizado por desenvolvedores de ferramentas de administração de bancos de dados baseados em grafos para garantir a integridade das operações de evolução de esquemas de dados e consequentemente dos dados relacionados.

Palavras-chave: Banco de Dados baseado em Grafos, Refatoração de Dados, Banco de Dados Evolutivos.

# CONTENTS

# ACKNOWLEDGMENT

I acknowledge my friends and family, who have supported and loved me during this journey, mainly my brothers and my parents who are my inspiration, my pride and to whom I dedicate this dissertation; *amo vocês*.

I acknowledge the professors and my colleagues of FT/Unicamp, who taught me a lot and who have helped me to grow in the past years.

I also acknowledge the Faculty of Sciences of Ghent University and the Babel Project (Erasmus Mundus Programee), who granted me a scholarship for the master mobility related to this research.

“Any fact becomes important when it's connected to another.”

*Umberto Eco*

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| ACID | Atomicity, Consistency, Isolation, Durability |
| BASE | Basically Available, Soft-state, Eventual consistency |
| DB | Database |
| DBMS | Database Management System |
| DDL | Data Definition Language |
| NoSQL | Not Only SQL |
| RDBMS | Relational Database Management System |
| RDF | Resource Description Framework |

# 1. INTRODUCTION

A relational database is a system which stores data in collections of tables (DATE, 2004). A database system based only on the relational model can be inefficient given the growth in data volume, density and complexity, such as in cases where information systems use large amount of connected data. The alternative to these new demands is the project of database systems which adopt different data models instead of the relational one, known as NoSQL Databases (ANGLES, 2012).

NoSQL Databases ensure high availability, flexibility and scalability and are categorized according to their data model (STRAUCH, 2011). Graph Databases are classified as a NoSQL database which can meet the recent demands previously mentioned and which stores the data in graph structures, providing a natural way of handling highly connected data (ANGLES, 2012).

Another recent trend which motivates the use of different database technologies is the evolutionary nature of modern software development processes (AMBLER & SADALAGE, 2007). This process differs from the serial approach in which, first, all the requirements are identified, and only after a detailed design the implementation can be done; it is an incremental approach where the system will have series of releases, first modeled with an overview of the scope and adding new requirements and details in each release. One evolutionary database development technique is the database refactoring. Database Refactoring is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics (AMBLER & SADALAGE, 2007).

A graph database automatically supports this incremental approach due to its optional schema, but the application which accesses the database is responsible to manage and organize the changes. This evolution could be done in a more consistent and organized way if the changes could be done similarly to how they are currently done in relational databases.

This work suggests one technique for supporting the evolutionary approach in graph databases with the definition of a set of refactoring rules; adapting the typical problems of database management from the relational model to the graph model.

## 1.1 Goals

Evolutionary development is an iterative and incremental approach. The model is released incrementally and evolves over time instead of being developed after the creation of a requirement specification (AMBLER & SADALAGE, 2007). This software development approach comes with a need for databases that can also evolve their design over time.

The goal of this project is the definition of a refactoring rules set, presented as a catalog, to ensure the evolution of Graph Databases in an organized way, preserving its features, such as flexibility, scalability and performance, and establishing procedures to maintain the project architecture of the database.

## 1.2 Motivation

The complete or total absence of schema in Graph Databases enables the flexibility for storing data as it is and contributes to increase the availability. The flexibility and scalability are important because they make possible to work with any new type of data, independent of the content structure and the volume of data. The down side is that there is no data referential integrity guarantee like in the fixed schema of relational databases (LÓSCIO et al., 2011).

The motivation for this work is the need of controlling and organizing the graph database evolution, avoiding the loss of data integrity, but also the loss of flexibility and scalability.

## 1.3 Work Methodology

The initial step to produce the results of this work was the research on refactoring database processes for relational databases which provided a better overview of database evolution; at the moment this work started, the researches about graph database evolution were in an initial stage. The refactoring rules for Graph Databases were established using the study of those existent techniques as a guideline, mainly the ones described by Ambler and Sadalage (2007).

Many refactorings of relational databases can be applied in a similar way in graph databases, thus, some basic refactorings were omitted from this work and the most important refactorings of the catalog developed are those bringing typical operations from graph databases, for example, operations in nodes and relationships. The starting point for finding those typical refactoring operations was the study of real life examples using graph databases; followed by the analysis of the possible changes that could be applied to them.

The structure of the catalog was developed to present the refactorings in a homogeneous way, trying to make most of refactorings as simple as possible, except for some complex refactorings involving a combination of refactorings.

## 1.4 Text Structure

This text is organized as follows:

- Chapter 2 presents an overview of NoSQL Database and Graph Databases;
- Chapter 3 explains briefly database evolution and the concept of refactoring;
- Chapter 4 brings the catalog created in this work, containing refactoring rules and the process for performing them in Graph Databases;
- Chapter 5 shows the conclusion and directions for continuation of this research.

# 2. GRAPH DATABASE

The amount of information produced by the use of data capture devices (e.g. sensors), scientific instruments, social networks, user-driven content, Internet, among others, does not stop growing, generating a big volume of data in a small amount of time. Studies from the IDC iView have estimated that the total amount of enterprise data will double every two years until 2020, growing from about 130 exabytes in 2005 to 40,000 exabytes, or 40 trillion gigabytes, in 2020 (GANTZ & REINSEL, 2012). This information generated is valuable and it should be stored and manipulated in a good timing, so it is not lost or unusable.

Besides volume, variety and velocity, technological advances also contributed to changes in the nature and complexity of data, for example, unstructured, geometric and multimedia data. This large scale data, distributed, complex, and that are difficult to collect, store, manage and analyze with conventional database management systems is becoming part of many sectors of the global economy, representing a technological challenge which demands more than the structure of relational database can currently offer and stimulating the use of different forms of storage.

The solution adopted by some companies, like Google and Amazon, was the development of their own database solutions, known as NoSQL Databases, tailored to the requirements of their data. The next sections will describe relevant properties of NoSQL Databases, giving more attention to Graph Databases.

## 2.1 NoSQL

NoSQL Databases emerged from the need to implement solutions which required distribution and scalability. Relational Databases were not the best suited for those solutions because it is difficult and not very efficient to make transactions and join operations in a distributed system which uses them (OREND, 2010).

The term NoSQL was first coined in 1998 for a Relational Database which omitted the use of SQL (STRAUCH, 2011; DATE, 2004) and it was reintroduced by Eric Evans in 2009. Although some NoSQL Databases already existed at that time, such as Apache

Cassandra[1] and Google Big Table[2], this boosted the development of NoSQL (BRUHN, 2011).

NoSQL is short for "*Not Only SQL*" and translates the idea that Relational Databases can coexist with other technologies and each one has its own place, without meaning that new ways to store data have to replace Relational Databases.

NoSQL is not only a product or a technology, it is a hypernym of all databases that do not follow the RDBMS principles (DATE, 2004) and are often related to large data sets manipulated on Web scale (TIWARI, 2011). Typically, NoSQL technology is used to solve the scalability problem of traditional databases, where flexibility and velocity of data are important, but consistency and a predefined schema are not requirements.

Some properties that distinguish NoSQL Databases Systems from RDBMS are: NoSQL Databases do not require a fixed schema (schemaless); do not support join operations which must be implemented by hand; they are built to scale horizontally; and often do not attempt to provide all ACID guarantees (BRUHN, 2011). These properties are explained in more detail in the following section.

## 2.2 NoSQL Concepts and Properties

NoSQL Databases share some basic concepts which differ these databases from Relational Databases. Sharding, CAP Theorem and Schemaless are examples of those concepts.

The volume of data grows together with the need for scalability and performance improvement. Several solutions have been suggested for those requirements, for instance, the vertical scalability, where more power (CPU, RAM, robust server) is added to the existent machine, and the horizontal scalability, in which you scale adding more machines (nodes) to a system, such as a new computer to a distributed system.

A way to implement horizontal scalability is by using **sharding**, which consists of splitting data horizontally using data partitioning, i.e., split tables to reduce the number of rows, grouping similar data. Data typically requested and updated together are stored on the

[1] http://cassandra.apache.org/

[2] http://research.google.com/archive/bigtable.html

same node of a cluster (STRAUCH, 2011). As they are related, they should all be treated on the same physical machine.

In a sharded configuration, as in Figure 2.1, the client connects to a database process which abstracts the sharding. Thus, from the application point of view, the database is configured as a nonsharded environment and if there is a need to scale the database, the application does not have to change.

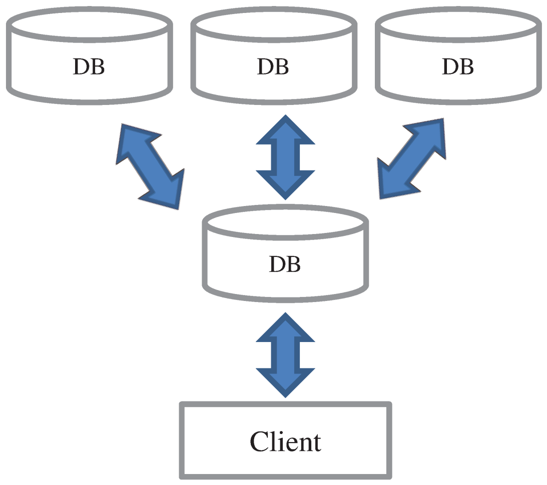

Figure 2.1: Sharded client connection (adapted from CHODOROW; 2013, p. 233)

The advantage of storing a table on a cluster instead of in a single machine is that nodes can be added to the cluster to increase its capacity and performance of read and write operations without changing the application, providing greater availability, decrease in query response time and parallelism (RUFLIN et al., 2011). The downside is that sharding makes some typical database operations complex and inefficient, for example, joins between data shards are not possible (OREND, 2010).

Some NoSQL Databases were already designed to use sharding, but its application in Relational Databases is more difficult due to some properties of the Relational Model. The first is that while a RDBMS follows normalization criteria, sharding favors the denormalization of data. Another one is that RDBMS usually apply a vertical scalability strategy and sharding works by parallelizing data in multiple servers to enable horizontal scalability (RUFLIN et al., 2011).

**Eventual Consistency** is a NoSQL Database property that states that the consistency does not have to be ensured in all points of a system and has the CAP theorem as the underlying principle.

The CAP theorem was introduced by Eric Brewer in 2000 and formalized by Gilbert and Lynch in 2002 (TIWARI, 2011). It states that in a distributed data storage system only two of the three features: consistency, availability and partition tolerance can be provided together (OREND, 2010). Before explaining the theorem, it is important to understand those features:

- Availability means that data has to be always accessible. The system is designed to always allow clients to read and write data in a specific period of time. If a system is busy, uncommunicative, or unresponsive when accessed, the system is not available (STRAUCH, 2011).
- Partition tolerance is the ability of the database to keep operating despite network and machine failures (WEBER, 2010). A partition tolerant system can only provide strong consistency by reducing its availability, because it has to ensure that each write operation only finishes if the data is replicated to all necessary nodes (OREND, 2010).
- A system is consistent if after a write update, all the concurrent operations see the same valid and updated data in a shared data source, i.e., the data has to be always the same in every replication on every server (WEBER, 2010).

As mentioned before, NoSQL uses eventual consistency. If there is an update, it does not guarantee that all processes have the same version of an item. In the case that no updates were identified, the data returned will be the last one updated. This can compromise consistency, but increases flexibility, availability and performance (RUFLIN et al., 2011).

Eric Brewer uses the term BASE (*Basically Available, Soft-state, Eventual consistency*) to represent the case of eventual consistency (TIWARI, 2011). BASE stands in contrast with ACID (*Atomicity, Consistency, Isolation, Durability*), although they are not opposites (TIWARI, 2011). While the ACID model enforces the consistency in the end of all operations, BASE allows the database to be basically available, appearing to work most of the time and eventually exhibiting a consistent state (ROBINSON et al., 2013).

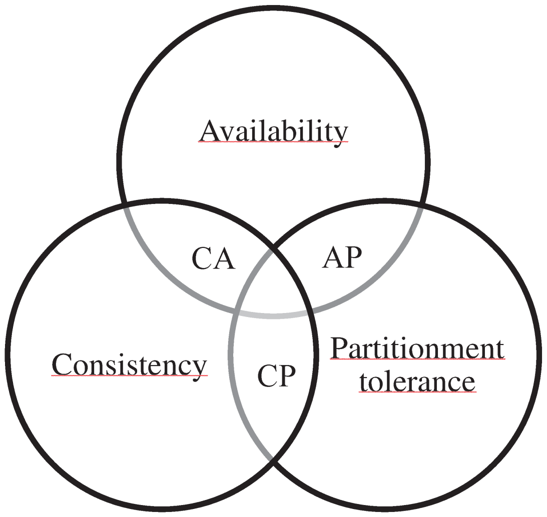

Figure 2.2: CAP Theorem

Figure 2.2 shows the possible options according to CAP theorem. The consistency and availability (CA) can be chosen by traditional RDBMS, where the consistency is most important. In some web applications, the data availability is more important than the consistency, so the alternative AP is more adequate and is the one used in Graph and Document Databases. Finally, the option CP is used for Key-Value Stores and Column Databases.

Another defining feature of NoSQL databases is the lack of a required schema. Database design, or modeling, is an essential activity in the development of information systems used to specify business rules and database structure. The database design is composed of Conceptual Design, Logical Design and Physical Design, as shown in Figure 2.3, being the first two of which the most important ones for this text.

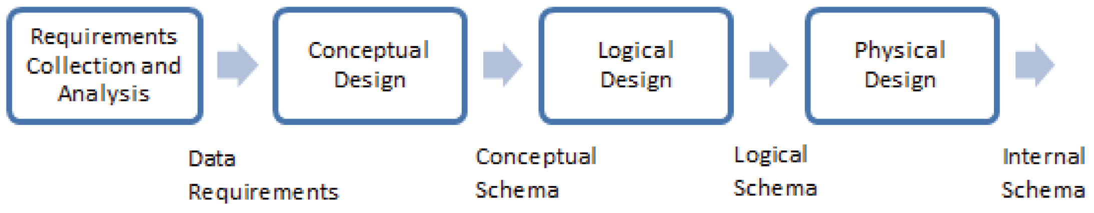

Figure 2.3: Simplified Diagram illustrating the main phases of Database Design (adapted from ELMASRI & NAVATHE, 2000)

The conceptual design is the high-level conceptual data model developed from the requirements collected. It results in the conceptual schema, a concise description of the user requirements, including detailed descriptions of entity types, relationships and constraints. This schema is usually a description of the database structure (usually graphical, as the Entity-Relationship model) independent of the technology or the application adopted (ELMASRI & NAVATHE, 2000).

The logical design consists in mapping the conceptual schema to meet the requirements of the database, make refinements when necessary. A logical schema is the description of the database structure that can be processed by a DBMS and it depends on the DBMS which will be used (ELMASRI & NAVATHE, 2000).

The physical design uses the logical schema to write the implementation of the database, including its storage structures and methods of data access. The physical project is directed to a specific DBMS and as soon as it is completed, the database can be created and filled with data.

Even after executing the whole process of database design, it can be necessary to incorporate new abstractions and modifications to the model and, although it is possible to make these changes on the relational database schemas, usually they are complex and demand cost and time.

In a large scale application, it is difficult to achieve the schema flexibility in SQL databases because of a high number of tables, causing many joins and unions with no good performance (AKRAWI, 2010). In order to avoid those difficulties, NoSQL Databases do not require a fixed schema (**schemaless**).

The complete or almost total absence of a schema to define the data structure of a model is a NoSQL Database characteristic (LÓSCIO et al., 2011). This absence supports the flexibility for storing data as it is and contributes to increase the availability. Although it brings advantages, there is no data referential integrity guarantee like in the relational databases using a fixed schema (LÓSCIO et al., 2011).

The loss of the data integrity guarantee due to the schemaless characteristic of NoSQL Databases is a motivation for this work. Since the beginning of this work, some NoSQL Databases recognized that it could be a problem to not ensure data integrity, developing alternative ways to solve the problem, such as offering an optional schema, which is explained in the Section 2.4.2 of Graph Database Management Systems.

## 2.3 Categories

Although NoSQL Databases have some features in common, there are different approaches to classify them. The categorization adopted for this work is the one proposed by Ben Scofield (STRAUCH, 2011), which classifies them in: Key-Value Stores, Column Stores, Document Stores and Graph Databases. This categorization was chosen because it is aligned with the proposal of this work, as it places Graph databases in its own category.

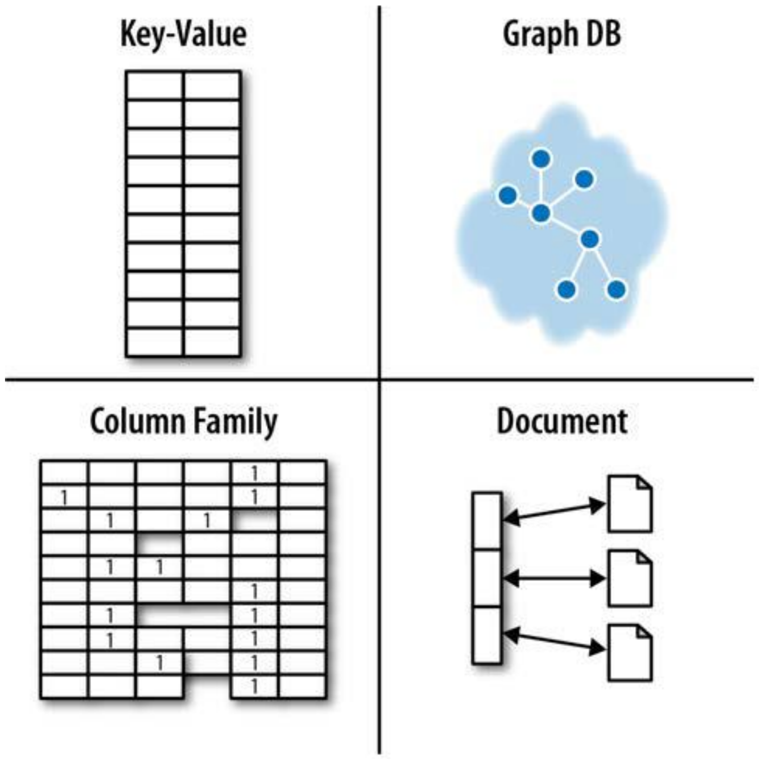

Figure 2.4: The NoSQL store quadrants (ROBINSON et al., 2013)

The different categories of NoSQL databases can be seen in Figure 2.4 and they are explained in the next sections, highlighting the Graph Database category since is the one used for this work.

### 2.3.1 Key-Value Stores

This model is based on a hashmap or an associative array, where there is a collection of unique keys and pointers to a particular data (values) associated with the keys. The key of a <key, value> pair should be unique in the set and it can be easily looked up to access the data (TIWARI, 2011). Values can be of different types like strings, integers, floats or byte arrays (WEBER, 2010).

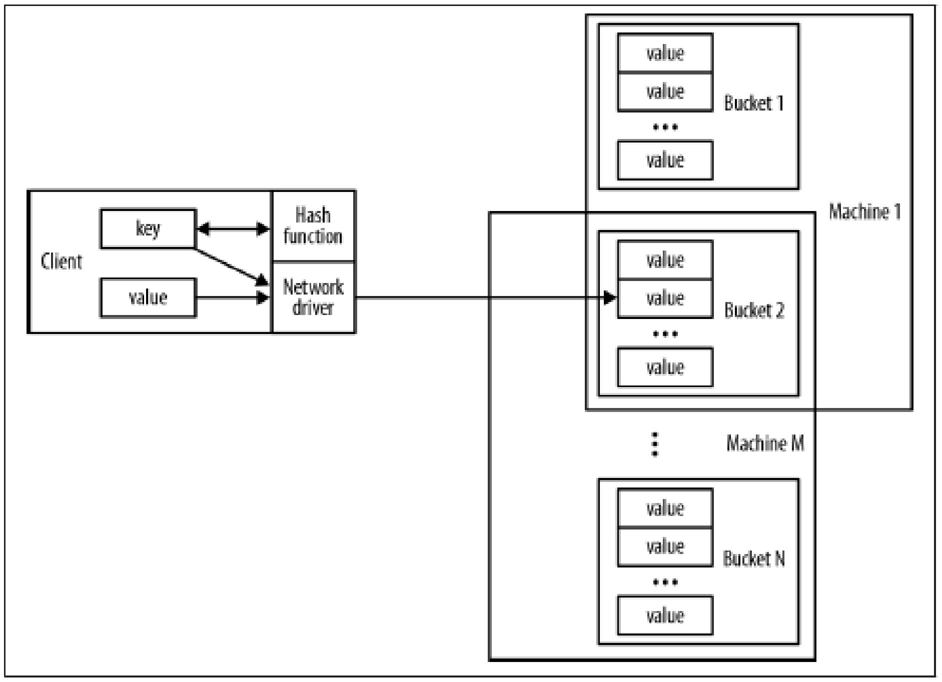

Figure 2.5: Key-value stores act like a distributed hashmap data structure (ROBINSON et al., 2013)

The key space of the hashmap is spread across numerous buckets on the network and, for fault-tolerance reasons; each bucket is replicated on several machines, as illustrated in Figure 2.5 (ROBINSON et al., 2013).

Key-values stores have a simple interface, with three primitive operations: to get the data associated with a particular key, to store some data associated with a key and to delete a key and its data (WARDEN, 2011). More functionalities or complex operations should be handled by the application and they are omitted in favor of high scalability.

With a key-value store, it is easy to achieve high performance; there is no single point of failure and high availability because of its flexible schemaless data models and fine granularity in the partitioning of the data (AKRAWI, 2010). The data model is simple, without relationships or structure (WEBER, 2010) and it is efficient because adding or removing a record is extremely flexible and it scales to large number of nodes (AKRAWI, 2010).

Some well-known Key-Value Stores are Amazon Dynamo, Redis and Google Big Table.

## 2.3.2 Column Stores

The approach to store and process data by column instead of row has its origin in business intelligence (STRAUCH, 2011). Instead of storing data by rows, as the RDBMS, this data model is based on a “sparsely populated table whose rows can contain arbitrary columns” (ROBINSON et al., 2013).

The idea is to store one attribute of a set of datasets in one unit (column oriented), unlike SQL databases that would store a dataset with its attributes in one unit (row oriented) (WEBER, 2010). In this way, similar data are stored together making data access more efficient if the query is made by specific columns or data (AKRAWI, 2010). Besides the efficiency in query by columns, this model avoids consuming space to store nulls by simply not storing a column when the value does not exist for that column (TIWARI, 2011).

To explain how this model works, consider a table with three attributes (identifier, name and address):

Table 2.1: Example of a table with three columns

| ID | Name | Address |
|---|---|---|
| **212** | Edward | New York |
| **213** | Mary | Seattle |
| **214** | Richard | Minneapolis |

In a model oriented by rows, the information of this table would be serialized to the hard drive or to the RAM in a sequence in this order:

```
212,Edward,New York;213,Mary,Seattle;214,Richard,Minneapolis;
```

The structure of the same example stored in a column-oriented way, would be something like this:

```
212,213,214;Edward,Mary,Richard;New York,Seattle,Minneapolis;
```

With this way of storing data, operations of aggregation can be done very quickly, because the values of the same attribute are stored successively. Also, as in the key-value stores, there are no relations between datasets (WEBER, 2010) and if there is a need to query all attributes of one record, each column has to be accessed separately.

Example of column stores are: SimpleDB[3] (Amazon), HBase[4] and Cassandra, explained briefly in the following section.

**Cassandra**

Cassandra was developed by Facebook using Java in 2008 (STRAUCH, 2011) to improve their Inbox Search feature (LAKSHMAN, 2010). According to Akrawi (2010), it is one of the most widely-used NoSQL Databases and it brings together the distributed systems' technologies from Dynamo and the data model from Google Big Table in a single model. "It provides a simple data model that supports dynamics control over data layout and format" (LAKSHMAN, 2010).

The equivalent to a table in Cassandra is a distributed multi-dimensional map indexed by a key (LAKSHMAN, 2010). Its data model is composed by key-value pairs of data, columns (key-value pairs with a timestamp), supercolumns (any number of columns combined) and column families (group both columns and supercolumns with a finite number of rows) (ROBINSON et al., 2013). It is classified as a column-oriented database and also as a distributed key-value store based on a distributed hash table (AKRAWI, 2010).

[3] http://aws.amazon.com/pt/simpledb/
[4] http://hbase.apache.org/

Cassandra is designed for high availability, eventual consistency, scalability, no single point of failure, minimal administration and continuous development, being scalability the primary reason to adopt it (AKRAWI, 2010). A huge amount of data is distributed across many servers, for instance, Facebook was running a 150 node Cassandra cluster without making changes to it (AKRAWI, 2010).

Failure detection is a Cassandra's mechanism by which each node can locally determine if any other node in the system is up or down. This is used to avoid attempts to communicate with unreachable nodes during various operations (LAKSHMAN, 2010), and to replace nodes without downtime.

Next to Facebook, also Twitter and Digg are companies who already used this database.

### 2.3.3 Document Stores

This category of NoSQL Database uses entire documents of different types as datasets. Those versioned documents are collections of other key-value collections. Examples of document types are structured human readable data files such as XML-, JSON- or YAML files (WEBER, 2010).

The main advantage of this approach is its flexibility (WARDEN, 2011). In traditional relational databases, the user has to specify the columns types and names for a table; additional values that were not specified when the table was created are not allowed and every value must be present, even if it is as a null value. Different from it, document stores allows the user to enter each record as a series of names with associated values (WARDEN, 2011).

It means that they do not use a predefined structure and that the database structure is the same as the datasets structure. If there is a requirement to create an attribute that will be used only by one record, it can be made directly to this record in the document, without the need for creating the attribute for all the records with null values, as it would be done in a relational database.

Document databases can be interpreted as particular cases of key-value stores. The difference is that the database has to know what kind of document is saved in it and it has to interpret it (WEBER, 2010). MongoDB[5] and CouchDB[6] are the two major representative databases of this class of NoSQL databases.

Data in CouchDB is organized in documents that can be stored and accessed with JSON[7]. Documents consist of named fields that have a key/name and a value. A fieldname, or identifier, has to be unique within a document and its assigned value may be string, number, boolean, date, an ordered list or an associative map (STRAUCH, 2011). Code 2.1, based on a MongoDB example from Plugge et al. (2010), illustrates the data storage flexibility because it does not require a predefined structure, thus different types of documents (representing a book or a cd) can exist in a collection called *Media:*

```
{
     "Type": "CD",
     "Artist": "Nirvana",
     "Tracklist": [
     {"Track": "1", "Title": "Smells like teen spirit"},
     {"Track": "2", "Title": "In Bloom"}
     ]
}
{
     "Type": "Book",
     "Title": "Definitive Guide to MongoDB",
     "Author": [
     "Plugge, Eelco", "Membrey, Peter", "Hawkins, Tim"
     ]
}
```

Code 2.1: Different types of documents in a MongoDB collection

The queries are made using the keys (unique identifiers) or any other value in the document. MongoDB is described in more detail in the following section.

[5] http://www.mongodb.org/
[6] http://couchdb.apache.org/
[7] JSON, or JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.

**MongoDB**

MongoDB is a Document Store developed in C++. It is open source and schemaless. It was designed by the company 10gen and provides flexibility, scalability and rapid application development (MONGODB, 2014).

Like the other NoSQL Databases, this category does not use tables, schemas, joins or SQL (PLUGGE et al., 2010). Instead of storing data into tables, data is stored in JSON documents.

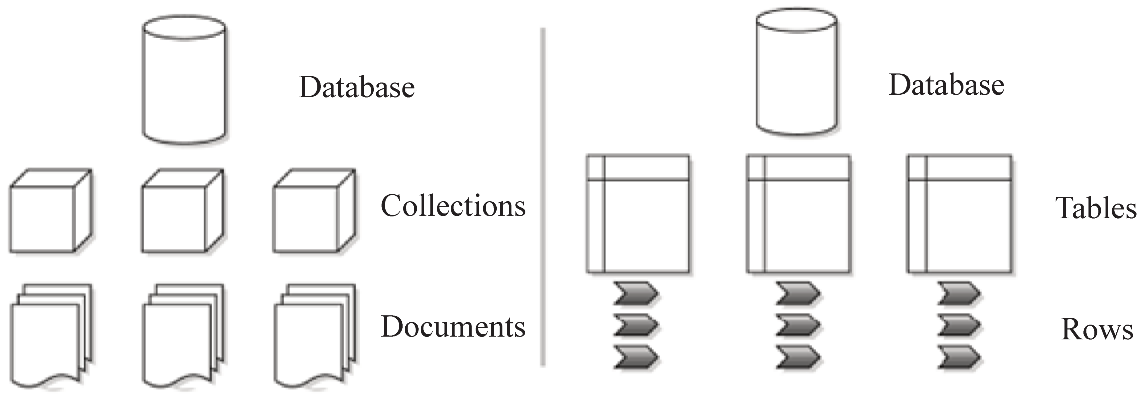

Figure 2.6: MongoDB Database Model and Typical RDBMS model (PLUGGE et al., 2010)

In MongoDB, a document contains the data, equivalent to records in SQL and collections that store the documents, equivalent to tables, as shown in Figure 2.6. Although the concepts are similar in an abstract way, the way that they store the data is different.

The main difference is that relational databases define attributes on the table level, while document databases store attributes on the document level. Documents can store records with completely different attributes, because MongoDB supports also complex data types (HAN et al., 2011). Another advantage is the high-speed access to mass data: when the data exceeds 50GB, MongoDB access speed is ten times faster than MySQL (HAN et al., 2011). MongoDB is fast, scalable and easy to use (PLUGGE et al., 2010).

Its philosophy is “one size does not fit all” (PLUGGE et al., 2010). Each data has to be analyzed to decide the best way to store it. In the relational approach, it did not matter if the data was a good fit for the model, because it would be stored using a relational database anyway.

MongoDB works with data redundancy; each isolated document has to contain all information it requires (CHODOROW, 2013). Although it makes it hard to update related records, this property increases query performance.

This database serves financial services institutions, electronic companies, media and entertainment companies, healthcare companies, having more than 1,000 customers, including Cisco, eBay, SAP, Foursquare and Telefonica (MONGODB, 2014).

### 2.3.4 Graph Databases

Graph Databases can be characterized as those where data structures for the schema and instances are modeled as graphs or generalization of them (ANGLES, 2012). As opposed to Relational Databases which store data in tables, data is stored in connected objects.

A graph consists of a set of nodes and edges connecting nodes. A graph where the edges have a direction associated to them is called a directed graph and if they have no direction, it is called an undirected graph.

In Graph Databases, vertices represent entities and the edges represent the kind of association between them. In this way, there is no need to think about how to represent the data, the developers only have to think about the relations between nodes and focus on manipulation of the data instead of how to represent it (AKRAWI, 2010).

The edges of the graph can have properties that describe the relationship between vertices (WEBER, 2010), for example, a *KNOWS* relationship type can represent that two nodes of the database know each other. In some cases, the direction of the relationship is also important, as in the relationship type *WORKS_IN* where a node representing a person can have this relationship connecting it to a company, but the other direction would not make sense.

The first and the third table represented in Figure 2.7 (a) have a many-to-many relationship and the table between them is used as a join table. In a Graph Database (Figure 2.8 (b)), there is no need to introduce a join table between the data because the relationships are already explicit in the model.

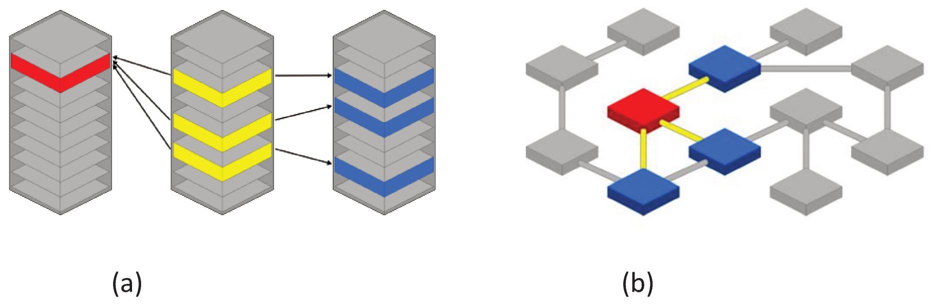

Figure 2.7: Comparison between Relational Database and Graph Database Structure

There are several different graph database models (ROBINSON et al., 2013), including:

- Property graph: contains nodes and relationships (including name and direction), both containing properties that are key-value pairs.
- Hypergraph: a generalized graph model in which a relationship can connect any number of nodes; and
- Triples: a way to express information about entities is using semantic triples, in the form of subject-predicate-object: a resource (the subject) is linked to another resource (the object) through an arc labeled with a third resource (the predicate). A combination of triples results in a directed graph.

Graph Databases are easy to scale horizontally (WEBER, 2010). Graphs can be partitioned so that each portion has a determined size and fewer connections between them, ensuring a better performance (WEBER, 2010). Another advantage is that the data query is fast for connected data and it is done directly on the graph structure, without the need for SQL join operations (ANGLES, 2012).

Just like relational databases have their operations (*create*, *insert*, *select*, etc.), there are many operations that can be done in graph databases. They have CRUD (create, read, update and delete) methods and, according to Akrawi (2010), the basic operation is traversing. The searching for information is made through the navigation between nodes, keeping track of which nodes were already visited and which ones were not visited yet. Graph databases are optimized for highly related data with graph traversal of high performance (AKRAWI, 2010).

In the example of Figure 2.8, the purchase history of a user is being modeled. The graph links the user to his orders and each order is linked to the products bought, so it is possible to have an insight of customer behavior. This is only an example, because graphs are everywhere: social networks, related products, spatial data, Internet, human brain, etc. It is hard to talk about data without talking about connections and, therefore, a database that make possible to store data structured as graphs is interesting because it makes easier to represent, visualize and manipulate this kind of data.

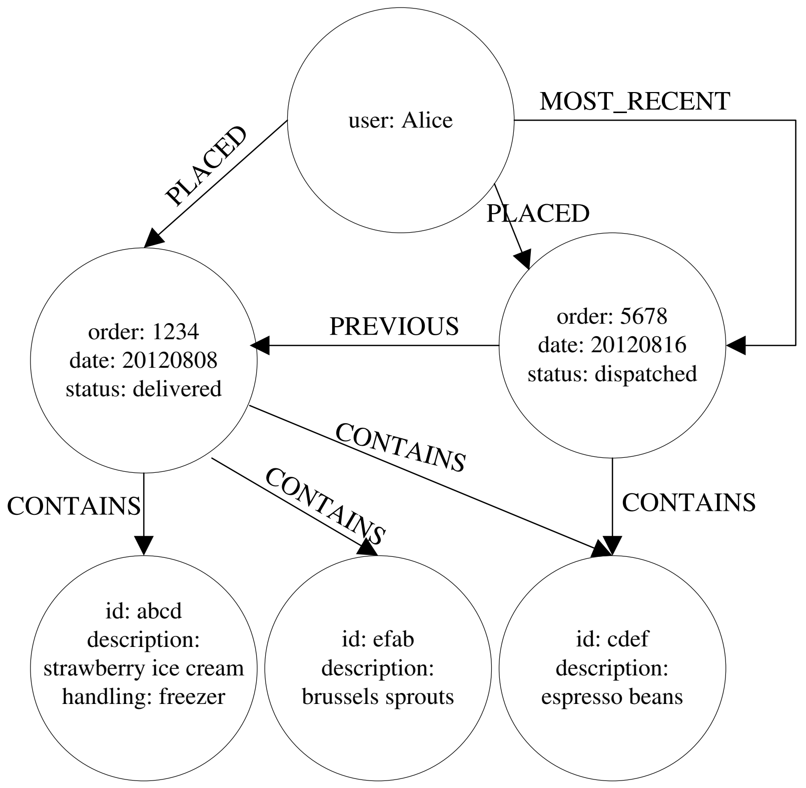

Figure 2.8: Example of Graph Database model (ROBINSON et al., 2013)

## 2.4 Graph Database Management Systems

In order to choose a graph database to be used for this work, a brief study about some of the tools available were made. This section presents a description of some relevant graph databases.

### 2.4.1 Graph Databases Overview

Of the many graph databases available, there are DEX (Sparksee)[8], HyperGraphDB[9], InfiniteGraph[10], AllegroGraph[11] and Neo4j. This section brings a short overview of them based on the information from their websites and the comparison of current graph databases models made by Angles (2012).

**DEX** is a scalable high-performance graph databases. It is suitable for huge amounts of data and it was developed by researches of the Technical University of Catalonia. DEX is natively available for .Net, C++, Python and Java, and for any operating system, even Android and iOS. It also has native indexing, which allows fast access to the graph data structures. DEX offers a restricted version for personal use, but it is not open-source.

**HyperGraphDB** is based on generalized hypergraphs as its underlying data model. This model allows a natural representation of higher-order relations, being interesting for artificial intelligence and semantic web projects. It is a Java embedded database and open-source.

**AllegroGraph** store data and meta-data as triples. It is a database for building semantic web applications (RDF – Resource Description Framework), but it also includes support for social network analysis and temporal reasoning.

**InfiniteGraph** is a distributed database oriented to support large-scale graphs, available in both free and paid license versions. It aims the efficient traversal of relations across massive and distributed data stores and its language is Java.

**Neo4j** is an open-source property graph, fully transactional Java persistence engine that provides different API's for Ruby, Python, and Java with support for various web technologies (AKRAWI, 2010). It provides support for full ACID transactions (NEO4J, 2014).

When this research first started, Neo4j was a schemaless database, meaning that the organization of information was left under the responsibility of the application, compromising the data integrity and reliability. Since the release of Neo4j 2.0, Neo4j is

[8] http://sparsity-technologies.com/
[9] http://www.hypergraphdb.org/index
[10] http://objectivity.com/infinitegraph
[11] http://www.franz.com/agraph/allegrograph/

now a schema-optional graph database (NEO4J, 2014). This means that it is possible to use Neo4j without any schema, but there is the possibility of a built-in schema, with indexes (improving the performance of looking up nodes in the database), constraints (helping to enforce data integrity, specifying the rules for the data and denying any changes that break these rules; the only constraint type available in Neo4j until now is the unique constraint.) and labels (a named graph construct that can group nodes into sets making queries more efficient) that can be associated to the constraints.

Neo4j is a leading graph database and besides the social network application, it has customers, such as Hewlett-Packard for the use case of network and data center management, eBay for routing and logistics, Cisco for content management and Walmart using Neo4j for make recommendations (NEO4J, 2014).

Table 2.2 shows the comparison made by Angles (2012) between the data structures of the graph databases described above. The data structures refer to the types of entities or objects that can be used to model the data.

In the Table 2.2 there are three graph data structures: simple graphs, hypergraphs and attributed graphs (or property graphs), notions that were explained in the last section. Additionally to the type of graphs, the edges can be directed or not and nodes/edges can be labeled or attributed (i.e., edges between edges are possible). The introduction of attributes for nodes and edges is oriented to improve the speed of retrieval for the data directly related to a given node.

Table 2.2: Graph Data Structures

| | Graphs | | | Nodes | | Edges | | |
|---|---|---|---|---|---|---|---|---|
| | Simple graphs | Hypergraphs | Attributed graphs | Node labeled | Node attribution | Directed | Edge labeled | Edge attribution |
| AllegroGraph | X | | | X | | X | X | |
| DEX | | | X | X | X | X | X | X |
| HyperGraphDB | | X | | X | | X | X | |
| InfiniteGraph | | | X | X | X | X | X | X |
| Neo4j | | | X | X | X | X | X | X |

It was also verified in this study their usage of integrity constraints, such as:

- Types checking (test consistency of an instance in respect to previous definitions);
- Node/edge identity (a node/edge can be identified by a value (id) or by values of its attributes);
- Referential integrity (test that only existing entities are referenced);
- Cardinality checking (verify uniqueness of properties or relations);
- Functional dependency (test that an element in the graph determines the value of another); and
- Graph pattern constraints (to verify a structural restriction).

Graph databases lack support to integrity constraints and they justify it with their agility and support for evolving schemas. According to Angles (2012), this argument is not valid assuming that data consistency in a database is as important as a flexible schema and the evolution of the schema could be supported by flexible structures in the schema, which is also a motivation for this work.

DEX, HyperGraph and InfiniteGraph, for example, support types checking and node/edge identity, also supported by Neo4j, but most of the integrity constraints are not natural supported by the databases and this still has to be developed.

## 2.5 Final Considerations

The use of new ways to store data, besides Relational Databases, is growing each day, requiring a greater attention for NoSQL Databases. This section summarized the basic concepts and characteristics of NoSQL Databases and its categories.

After an analysis of the available Graph Databases, Neo4j was the one used to support this work because it had more material available making easier to understand its data model, because it is becoming a widely used database and because its current version offers an optional schema.

With the new optional-schema version of Neo4j and the increase of the amount of companies using Graph Databases for other applications rather than social networks, the importance of the organization of the evolution of Graph Databases becomes clearer, motivating the research in this document. Despite it, NoSQL Databases still have deficiencies and among them are the methods of Refactoring, which will be addressed in the next section.

# 3. DATABASE REFACTORING

In traditional database design, the process to determine and organize the information which is necessary to keep is very time consuming. A lot of time is spent from the requirement analysis until the actual implementation of the database, with the goal to ensure that all data objects required by the database are accurately represented and maximize the use of resources. The physical model is implemented after many discussions and careful analysis and, only after finishing the whole process, the application is allowed to access the data. If subsequent modifications are required after this process is done, it can be considered that a failure has occurred in the database design.

Contrasting with this almost serial manner to model the data, there is an evolutionary approach to data modelling (AMBLER & SADALAGE, 2007). Evolutionary database design accepts the changes in the model as part of the process, because there will be constantly changes in the requirements, allowing them to occur even late in a development project. “Changes are controlled, but the attitude of the process is to enable change as much as possible” (FOWLER et al., 1999). The development techniques that support evolutionary databases are: Evolutionary data modeling; Database regression testing; Configuration management of database artifacts; Developer sandboxes; and Database Refactoring, which will be discussed in more detail in this chapter.

## 3.1 Concept

Refactoring is defined by Fowler (1999) as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”, i.e., it is a small change to the code in order to improve the design without changing its external behavior and it enables an evolutionary approach to programming.

Fowler (1999) also stress that there is no functionality added when refactoring code, it is only an improvement of the existing code so that it can be applied before adding functionality to the program.

Just like it is possible to refactor the source code to improve the quality of the design, it is also possible to refactor the database schema. Similar to the definition of refactoring, a database refactoring is “a simple change to a database schema which improves its design while retaining both its behavioral and informational semantics” (AMBLER & SADALAGE, 2007), in other words, database refactoring neither adds anything nor breaks the functionality or the data that is already stored; it merely improves the database.

The process of database refactoring requires a deprecation period or a transition window. If many applications (hundred) access the same database, there should be a transition period where the old and the new schema work in parallel. Since not all applications can be changed at once to work with the new schema, first a new schema is created and the old one is set as deprecated, together with scaffolding code to keep both schemas synchronized. This transition period ensures the agility of refactoring because it is not necessary to wait for all applications to change to the new schema, although, a transition period may last two years (AMBLER & SADALAGE, 2007).

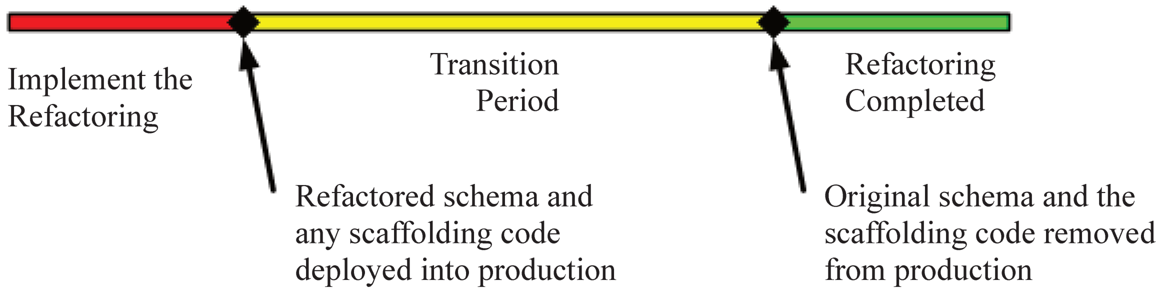
Implement the Refactoring
Transition Period
Refactoring Completed
Refactored schema and any scaffolding code deployed into production
Original schema and the scaffolding code removed from production

Figure 3.1: The lifecycle of a database refactoring (AMBLER & SADALAGE, 2007)

After the transition period ends, the old schema and the scaffolding code can be removed and the refactoring is complete, as illustrated in Figure 3.1. It is important to consider that the applications will either access the old or the new schema, but they will never be updating both schemas. It is the responsibility of the database to keep both schemas updated and it does this in some automated way or by using triggers – a procedure that initiates an action when an event (insert, update or delete) occurs.

Even requiring some extra operations, the evolutionary approach together with refactoring techniques can make the database implementation agile, the next sections show examples of the refactoring process.

## 3.2 Relational Databases Refactoring

Ambler and Sadalage (2007) distinguish six categories of database refactoring:

1) Structural. A change to the table structure of the database schema. (e.g. the union of complementary columns)

2) Data Quality. A change which improves and/or ensures the consistency and usage of the values stored within the database. (e.g. adjustment of business rules)

3) Referential Integrity. A change which ensures that a referenced row exists within another table and/or that ensures that a row which is no longer needed is removed appropriately. (e.g. cascade delete)

4) Architectural. A change which improves the overall manner in which external programs interact with a database. (e.g. the migration of methods to the database)

5) Method. A change which improves the quality of a stored procedure, stored function, or trigger. (e.g. reorder parameters); and

6) Non-refactoring Transformations. A change which changes the semantics of your database schema by adding new elements to it or by modifying existing elements. (e.g. insert data, introduce new column).

To explain the process of database refactoring, it is easier to introduce some examples chosen because of their simplicity, the structural refactorings *Rename Column* and *Merge Columns* excerpted from the book of Ambler and Sadalage (2007). The first can be applied to increase the readability of the database schema or, for example, before exporting your data to a new database that uses the old column name as a reserved key and the second when two columns are always queried together.

The steps of the process of database refactoring are presented in Figure 3.2 and the example of the refactoring Rename Column in Figure 3.3 will be used to support the explanation of those steps.

In the example of Figure 3.3 there is a *Customer* table with a column *FName*. Since the name is not intuitive, the user decides to change it to *FirstName*. The first step in the process of database refactoring, shown in Figure 3.2, is to verify if the refactoring is needed and to pick the right one to perform (it could be better to create a new column with another data type than to rename it).

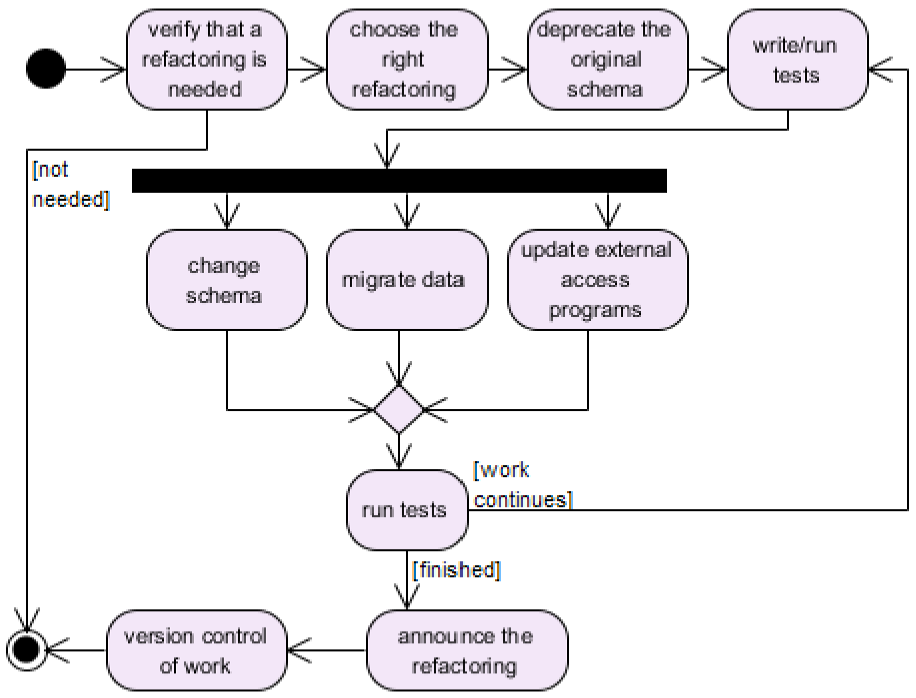

Figure 3.2: The process of database refactoring (AMBLER & SADALAGE, 2007)

The next step is a deprecation period (transition period). A *FirstName* column is added to the table, the data from column *FName* is migrated to column *FirstName* if necessary and the old schema is marked as deprecated. During this step, a trigger can be introduced to keep the values contained in the two column synchronized.

After the original schema is deprecated, external programs that access the database should be updated to work with the new version and the database refactoring is validated by writing and implementing unit tests. While running the tests to discover problems, there will be a need to rework things until they get right, and, then to communicate the changes that have been made to all application teams. The last step of the process is to put any DDL created, changed scripts, test data, test cases, documentations and models into a version control tool.

### 3.2.1 Renaming Column

As shown in Figure 3.1 and explained before, there are three sections now: the original schema, the transition period where *FName* is marked to be dropped at a specific date and time and coexists with the new column and the resulting schema. It is required to introduce a synchronization trigger to copy data from one column to the other during the transition period whenever a row is inserted or updated in the database. This trigger is also marked to be deleted together with the old column.

After running tests to check if the new schema is working properly, the next steps are to announce the changes made by refactoring and keep a version control of the work.

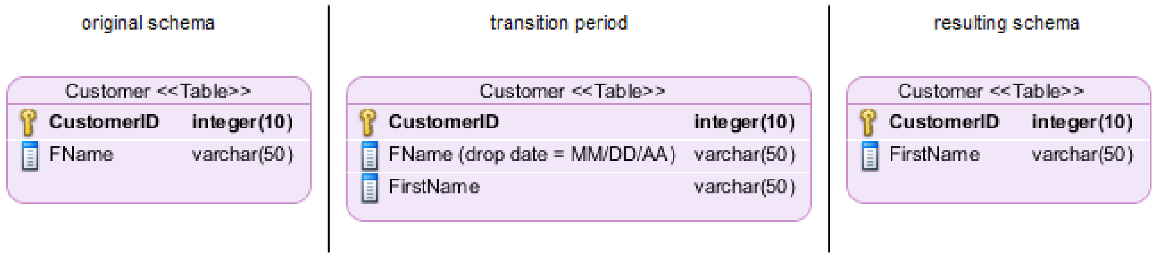

Figure 3.3: Renaming the *Customer.FName* column (AMBLER & SADALAGE, 2007)

Renaming a column in a production database without breaking applications accessing it is a very simple example used to illustrate the process of database refactoring because it is necessary to think about how to evolve a database in a small way first.

Database refactoring is hard because of the degree of dependence between two items, what is called coupling. The best case scenario is one database being accessed by one application, but usually a database is accessed by a wide variety of software systems, as external systems, applications or other databases.

### 3.2.2 Merge Columns

Another example of database refactoring is the structural refactoring *Merge Columns*. A Structural Refactoring is applied to improve the database schema, cleaning, organizing, standardizing, and improving the structure and the logic of a database.

When a database evolves, it is possible to create a column containing additional information to another column, which both are always used together, motivating the *Merge Columns* refactoring to improve the structure of the database.

Figure 3.4 brings an example of this refactoring; the table Customer has two columns, *PhoneAreaCode* and *PhoneLocal*, which are always used together and could be merged into one column. During the transition period there is a new column storing the combination of the other two columns and a synchronization function. In the date specified the old columns and the function to synchronize them are removed from the database resulting in the new schema.

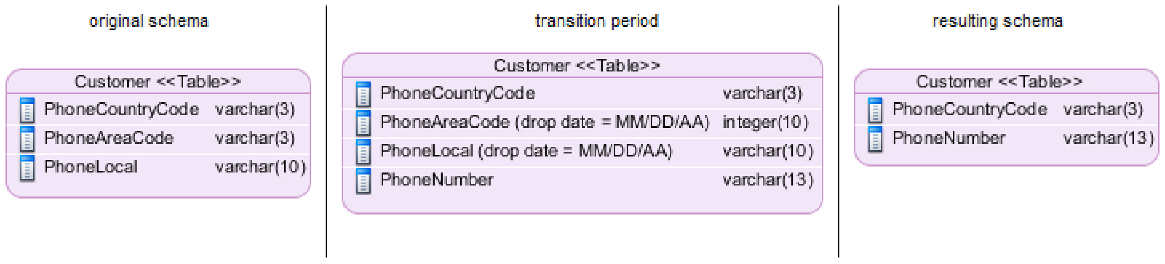

Figure 3.4: Refactoring *Merge Columns*

Similar to the *Merge Columns* refactoring from relational databases, a *Merge Properties of a Node* can be created for Graph Databases. An example of what was developed in this research is in Figure 3.5 where two properties of a node are combined into a third one resulting in a new schema.

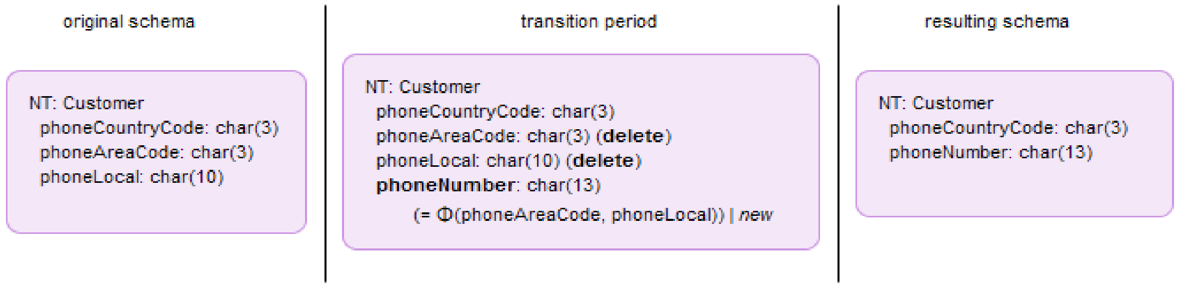
original schema
NT: Customer
phoneCountryCode: char(3)
phoneAreaCode: char(3)
phoneLocal: char(10)
transition period
NT: Customer
phoneCountryCode: char(3)
phoneAreaCode: char(3) (delete)
phoneLocal: char(10) (delete)
phoneNumber: char(13)
(= Φ(phoneAreaCode, phoneLocal)) | new
resulting schema
NT: Customer
phoneCountryCode: char(3)
phoneNumber: char(13)

Figure 3.5: Refactoring *Merge Properties of a Node*

This and other Graph Database Refactorings will be introduced and explained in Chapter 4.

## 3.3 Schema Changes in NoSQL

NoSQL databases are not entirely schemaless; the schema is actually defined by the application. The application is responsible to parse the data before saving in the database or reading from the database (SADALAGE & FOWLER, 2012). Thus, even in schemaless databases, the schema of the data has to be taken into consideration when refactoring the application (SADALAGE & FOWLER, 2012).

Sadalage and Fowler (2012) suggest a technique known as incremental migration to make sure that data, before the schema changed, can still be parsed by the new code, and when it is saved, it is saved back in the new schema. This technique migrates data over time as the data is being accessed. With this technique that could be many versions of the object on the application side that can translate the old schema to the new schema and also requires a transition period.

In the case of graph databases it is not possible to change only some of the edges because it would not be possible to traverse it anymore. Possible solutions are to traverse all the edges and change the type of each edge or to create new edges between nodes and later drop the old edges (SADALAGE & FOWLER, 2012).

Schemaless databases still need careful migration due to the implicit schema in any code that accesses the data and the same migration techniques as databases with strong schemas.

## 3.4 Final Considerations

This chapter brought an overview about refactoring in general, applied to relational databases and schema changes in NoSQL. The study of relational database refactoring was essential because supported in many ways the development of refactoring rules for graph database.

The main points are the idea of a transition period where the old and the new schema work in parallel, the categorization of each group of refactorings to make the rules more organized and two examples of the existent refactoring operations for relational database that were used as a baseline for the development of the first refactoring rules for graph database. The study of the process of database refactoring showed in Figure 3.2 is important to maintain the integrity of the database while evolving the database schema in small steps and it can also be applied to graph database refactoring.

Section 3.3 explains an option to consider the data schema described on the application of a NoSQL Database when refactoring an application called as incremental migration. This and the relational database refactoring approach are processes that enable the development in an evolutionary manner and were the basis for the next chapter.

# 4. GRAPH DATABASE REFACTORING

The main purpose of this work, as mentioned before, was to develop refactoring rules for graph databases using the rules for relational databases proposed by Ambler and Sadalage (2007) as a guide. For this purpose, the assumption was made that there exists an ideal graph database platform providing full support for an optional schema, which would include constraints, labels for nodes and relationship types.

The refactoring rules proposed in this document were elaborated using the graph database Neo4j as support and some specific features from this database can be noticed, such as the use of properties in nodes and relationships, the labels to identify different node types and the data types used in the examples. However, the rules are intended to be generic and they could be applied to other graph databases.

Chapter 3 explains that data refactoring is a simple change to a database schema, in order to improve it, which does not add functionality and preserves the data stored in it. Before applying a refactoring, the data modeler should evaluate whether the behavior and information semantics of the object changed will be maintained after the changes. If the semantics are not the same, the refactoring is not applicable and it should not be allowed.

All database refactoring rules were developed thinking about situations with well determined logic. Because of this, one refactoring can include another refactoring or it can be a combination of other simple refactorings and, in these cases, they are treated like one unit. Although they can be used together as one unit, two refactorings cannot be launched simultaneously by the user to preserve the atomicity and ensure the consistency of the operations. The rules were also evaluated trying to avoid the propagation of their event, because it could trigger unexpected events by the user, i.e., a refactoring will not trigger new events which were not required by the user, for example, when trying to delete a property which is used to calculate another property, the refactoring will be blocked.

Refactorings which include creation of nodes will also require the creation of a specific identification besides the general *id* created automatically by the database. This *id* can be a surrogate key or a natural key created for a specific type of node.

## 4.1 Metadata

Metadata Schema is a set of metadata elements (structured information) designed to describe, explain, manage and use a particular type of information. In databases, a metadata schema generally specifies other information, such as the domain of the data and rules of content representation.

For this work, two levels of Metadata were created in order to detail the basic structure of a Data Schema. These levels make possible to apply refactoring and support the Database Evolution in Graph Databases. They are general to any graph database management system assuming that they have nodes, edges and properties. Figure 4.1 shows the data schema called Metadata level 0 (zero) represented by nodes and edges. Each node has a property key-value in which the key is the property name and the value what it represents. In Figure 4.1 there are four main domains: `node_type`, `relationship_type`, `node_property` and `relationship_property`; and other nodes representing properties of the main domains.

A `node_type` (NT) is the type of a node and a `relationship_type` (RT) represents the type of an edge connecting two nodes in a determined direction. Both `node_type` and `relationship_type` can have properties (`node_property` and `relationship_property`) containing their name, datatype, size, if accept null values (nullable) and default value, besides others not used in the refactoring rules presented in this work.

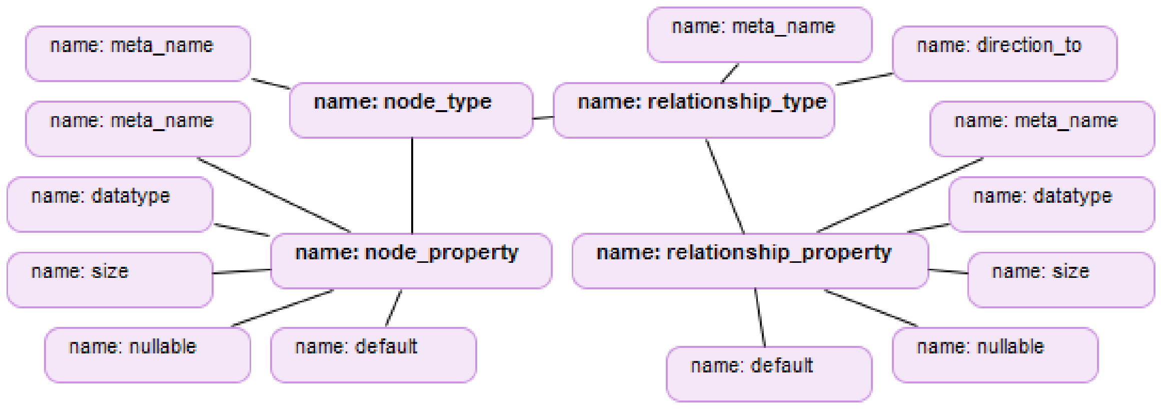

Figure 4.1: Metadata level 0

Metadata level 1 (one) uses the metadata described in Metadata level 0. Each node can have a `node_type` (NT) with determined properties and their datatype and each edge has a `relationship type` (RT) with its properties. Figure 4.2 brings an example of Metadata level 1 with two `node_type`'s and one relationship type `has`. The `node_type` Customer, for example, has three `node_property`'s describing their datatype and their size when necessary.

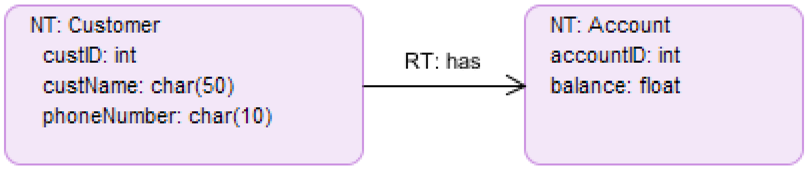

Figure 4.2: Example of metadata level 1

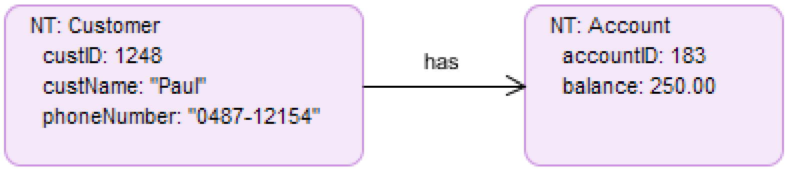

Figure 4.3: Example of the data using metadata of Figure 4.2

Figure 4.3 shows a partial example of how a database using the metadata level 1 of Figure 4.2 would look like. The nodes of determined NT would have the properties described in metadata level 1 working as an optional schema.

Some graph databases, such as Neo4j, are already changing to support an optional schema, offering labels that work similar to node types and indexes to improve query performance, but since the rules in this work are general to any graph database, these metadata levels were created independent of the Neo4j optional schema to handle database constraints and schema evolution in any graph database.

## 4.2 Classification

Some of the refactoring rules cited in this catalog, for example the *merge properties*, are very similar to the ones from relational database, (e.g., *merge columns*). Even with this similarity, some of them are included in this catalog in order to make easier the understanding of the rules. However, most of them were omitted in this document, since they do not bring relevant additional information to the process and they can be adapted straight from the refactoring catalog for relational database. Other refactoring rules are very specific for graph databases, such as a *relationship becoming a node* and *invert the direction of a relationship*.

Because the classification of database refactoring cited in Section 3.2 is related to specific features of relational database, that categorization cannot be applied to this catalog, making it necessary to use a new classification to organize the text. The refactoring rules were placed in four categories:

- Expansion Refactoring is a structural change aiming for the improvement of the database. A refactoring in this category improves the database by expanding it, bringing improvements as standardization, organization and separation of the data.
- Reduction Refactoring rules are used to improve the database schema, cleaning up and improving the structure of the database;
- Improvement Refactoring is a change in the database schema to improve the logic of the database;
- Data Quality Refactoring rules are used to improve or ensure the consistency and usage of the values stored within the database.

## 4.3 Specification Guideline

Each refactoring rule described in the next section is organized using the following structure:

- Goal;
- Motivation;
- Inputs (what the user should provide to apply a refactoring);
- Preconditions (points which should be evaluated to verify if the refactoring is useful and can be applied to the database; it’s the minimum analysis required and can be used as a general orientation; the possibility of adding new

preconditions has to be considered according to the database which is being refactored);

- Postconditions (points which should be evaluated after the refactoring process is completed in order to test it);
- Implementation (what the data modeler has to implement to create the refactoring);
- Steps (it describes the main activities that should be done by the data modeler during the transition period and the directions to complete the refactoring);
- Metadata (how would the schema change during the refactoring process);
- Example (most of the examples used are from a family or a bank database, they are simple examples to make the understanding of the refactoring easier);
- Application (a refactoring can require changes in the applications which access the database. This part brings an overview of what has to be done to adapt the applications but should not be limited to it); and
- Additional Notes.

A notation was developed to be used in the item Metadata in order to represent better the abstract data. The symbols used are listed in Table 4.1.

Table 4.1: Notation of the abstract data

| **Symbol** | **Meaning** |
|---|---|
| $\mathbf{N_A}$ | Node of type A |
| $\mathbf{p_i}$ | Property i |
| $\mathbf{T_i}$ | Datatype of property i |
| $\mathbf{\Phi}$ | Function |
| $\mathbf{\Phi: (T_i) \rightarrow T_N}$ | Transformation function from $T_i$ to $T_N$ |
| $\mathbf{N_A \vdash p_i}$ | $p_i$ is a property of $N_A$ |
| $\mathbf{R_x}$ | Relationship type |
| $\mathbf{R_x: N_A \rightarrow N_B}$ | Relationship between node type A and B |

This homogeneous representation is used for all refactoring rules as an orientation to their use. Since they are subdivided in topics, it is easier for the data modeler to find the refactoring which best suits the desired change and to verify the steps and actions required for its execution.

## 4.4 Expansion Refactoring

In the category Expansion Refactoring three refactoring rules were included: Split property of a node, Property becoming a node and Relationship becoming a node. All three refactoring rules expand the schema and hence the database, but, at the same time, they improve the data organization.

### 4.4.1 Split property of a node

This refactoring allows the data modeler to split a property of nodes from a determined node type into one or more properties (*goal*). It can be applied when there is a need to work with fragments of the value of a property (*motivation*). Therefore, it is required to split the property to use one part of its value alone. This refactoring is the inverse operation of *Merge properties*.

| | |
|---|---|
| *Inputs:* | • Target node type ($N_A$); |
| | • Property to be divided ($p_j$); |
| | • Name and type of the properties to be created ($p_N$ and $T_N$); |
| | • Transformation function defining the division rules of the property: $\Phi$: $(T_j) \rightarrow T_N$. |
| *Preconditions:* | • A transformation function able to deal with unexpected values. |
| *Postconditions:* | • All new properties have to be of the same data type. If necessary, this can be changed later through a refactoring to change data types of properties; |
| | • The identifications (keys) of the new properties have to be different from the property selected to be separated. |

*Implementation:*

1. Create functions $\Phi_N$ to separate the property given as input, defining the data type of the new property and how to proceed when the value of a property cannot be separated.
2. Create a synchronization function $f_t$ to run automatically in order to maintain properties (the old one and the new ones) updated
   a. When updating/deleting the split property, or the new ones, the function must be invoked to copy data from this property to the other ones. Also when inserting new data through applications using the old/new schema, all the properties have to be updated.

*Steps:*

1. Select node type;
2. Select a property ($p_j$) to split;
3. If preconditions are met:
   a. Create the required new properties in the node type selected;
4. [Transition] Use the split function $\Phi$ to generate and set the value of the new properties ($p_N$) for all instances of the selected node type;
5. [Transition] Set a date to remove the split property ($p_j$) from the database;
6. [Transition] Remove the old property ($p_j$) and the function $f_t$ that synchronizes the properties
   a. If necessary, keep the data from the old properties outside the database
7. If postconditions are true after the execution of all the steps
   a. All nodes (instances) from target node type will have their properties $p_j$ transformed in *n* properties.

*Metadata:* In Figure 4.4, an example with abstract data is shown, which can also be explained with the following formulation:

Given:

$N_A$

$\vdash p_i: T_i$

$\quad p_j: T_j$

$\Phi_N: (T_j) \rightarrow T_N$

Resulting Schema:

$N_A$

$\vdash p_i: T_i,$

$\quad p_k: T_k$ | *new*

$\quad \dots$

$\quad p_N: T_N$ | *new*

A property $p_j$ from a node type $N_A$ is selected to be split into other properties. The number of properties, their name and the rules to divide the selected property will be provided in the transformation function. After the transformation function is implemented and the property to be divided is selected, the new properties are added to the target node type and an initial value to all instances is set through the synchronization function. After all the values are set and the old property is not used anymore, it can be deleted from the database resulting in the desired schema.

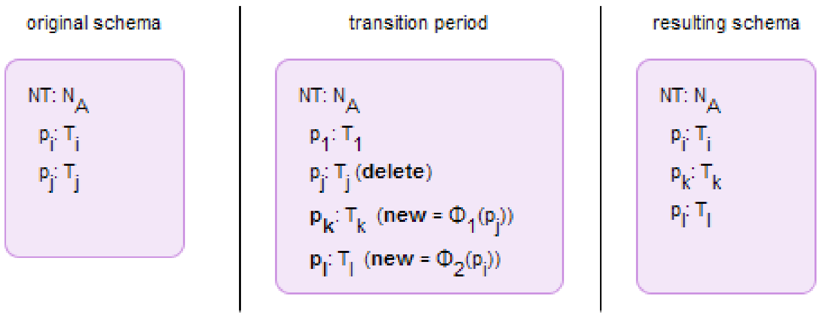

Figure 4.4: Abstract example of refactoring *Split property*

In the original schema of Figure 4.5, the property customer's name was storing the full name of the customer. Because of a user requirement, it was necessary to work with the middle and the last name of the customers and having this information in distinct properties could facilitate the querying information of nodes from this node type.

The refactoring creates new properties with the required values and removes the old property containing the complete value.

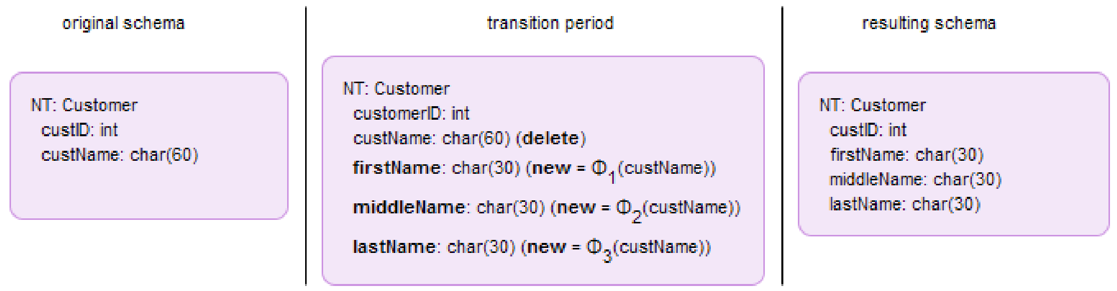

Figure 4.5: Refactoring *Split property* on metadata

*Example:* Figure 4.6 shows a practical application of the refactoring *split property*. In this case, three new properties were created separating the value of the property *custName*, which was storing the full name of the customer.

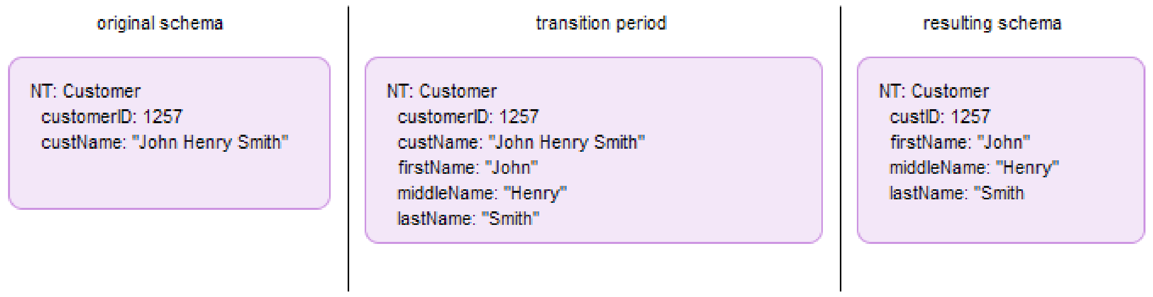

Figure 4.6: Refactoring *Split property* applied to an instance

*Application:* Change the references to the old property to the new properties. Applications have to remove code they were using before the refactoring.

*Additional Notes:* It is important to think about the treatment of unexpected data, which cannot be split using the transformation function; and how to proceed in the case that different instances have completely different values, becoming hard to set a unique way to separate the values and create the new properties.

The reverse refactoring of *Split property* is the *Merge properties*.

### 4.4.2 Property becoming a node

This refactoring allows replacing a property with a new node of a chosen type (*goal*). The purpose of this expansion refactoring is to increase the details of a property, turning it into a node (*motivation*). The metadata structure is modified evolving the property to a node, allowing it to have its own properties. Other parameters are:

*Inputs:*
- Target node type ($N_A$);
- Property to be transformed into a node ($p_k$);
- Relationship type between target node type and new node type ($R_X$);
- Name of the new node type [optional] (the key of the property can be used as the name if a new name is not provided);
- Name of the property of the new node type [optional].

*Preconditions:*
- The name of the new node type has to be different from the existing node types.

*Postconditions:*
- All nodes of the node type created have a relationship with a node from the same type of the one containing the transformed property.

*Implementation:*

1. Create two synchronization functions $f_1$ and $f_2$ to run automatically in order to maintain properties updated (the one from the selected node type and the one from the recent created node type).
   a. In any of the events update, insert or delete occurring in the new node type or in its property, the property selected to be transformed also has to change and the other way around. Each node type will have its own synchronization function.

*Steps:*

1. Select node type ($N_A$);

2. Select $p_k$ to be transformed into a node of a chosen type;
3. Provide a new relationship type ($R_X$) or select an existing one;
4. If desired, provide name of the new node type and the name of its property;
5. Evaluate preconditions;
6. [Transition] If preconditions are met:
   a. Create a new relationship type $R_X$ (if required);
   b. Create a new node type
   c. Create a connection between the selected node and the created node through the relationship type $R_X$.
7. [Transition] Create instances of the new node type and relationship between the instances of the new node type and the selected node type for each instance of the selected node type;
8. [Transition] Use the refactoring *move properties* to move the property selected to be transformed ($p_k$) to the new node type ($N_k$), changing its name later if required;
9. [Transition] Use the synchronization functions $f_1$ and $f_2$ to maintain properties updated;
10. [Transition] Set a date to remove the selected property($p_k$) from the database
11. [Transition] Remove the selected property($p_k$) and the function $f_t$ that synchronizes the selected property with the property of the new node type;
12. If postconditions are true after the execution of all the steps
    a. All nodes (instances) from the selected node type will have the selected property transformed into a new node type.

*Metadata:*

In Figure 4.7, an example with abstract data is shown, which can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $\vdash p_i$: $T_i$, | $\vdash p_i$: $T_i$, |
| $p_j$: $T_j$, | $p_j$: $T_j$ |
| $p_k$: $T_k$ | $N_k$ \| *new* |
| | $\vdash p_k$: $T_k$ \| *copy* |
| | $R_X$: $N_A \rightarrow N_k$ \| *new* |

As explained before in the steps, a new relationship type and a new node type are created. The new relationship will connect the target node type to the new node type and the property selected to become a node type will be moved to the new node type. The new node type, relationship type and property will be named according to the data modeler input or will remain the same as the old ones when it is possible. After all the steps are completed, the selected property will be removed and transformed into a new node type.

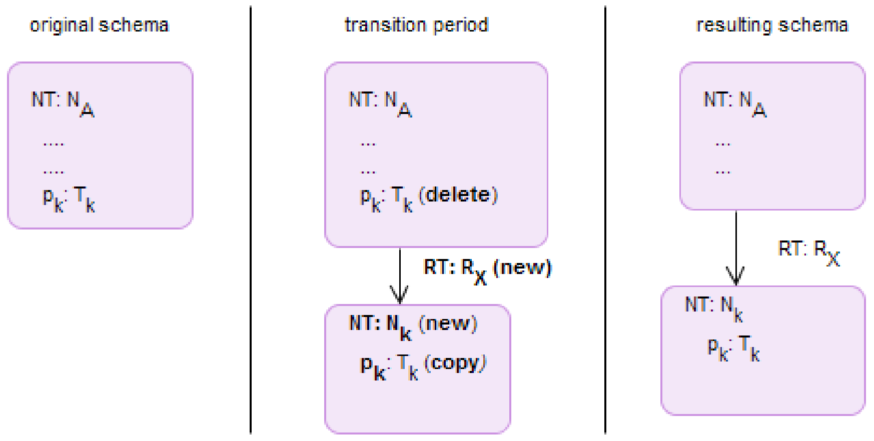

Figure 4.7: Abstract example of refactoring *Property becoming a node*

In the schema of Figure 4.8, there is a new user requirement to add more information to the property location of the node type *Wedding*. The data modeler decides then to create a new node type using the value of the property location. This refactoring would also be useful in order to normalize the database and avoid duplicated data if many nodes were using the same location. In this case, after the refactoring is completed, duplicate data should be removed from the database and their relationships moved to the same node. The performance should be evaluated before trying to normalize a graph database because it can bring future scalability problems.

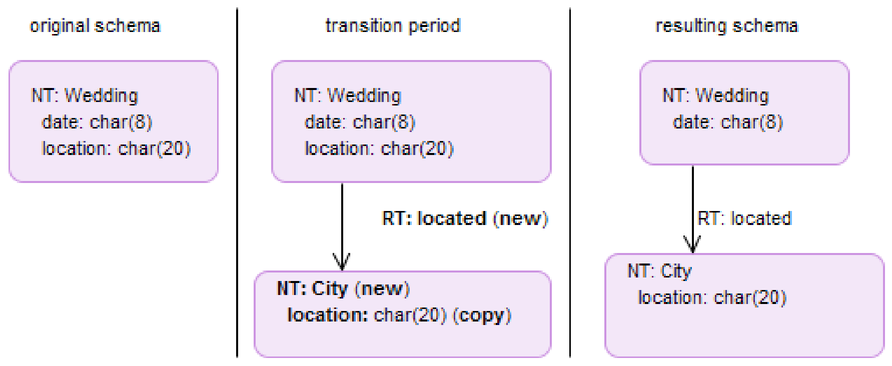

Figure 4.8: Refactoring *Property becoming a node* on metadata

*Example:* The data example in Figure 4.9 shows a different schema from Figure 4.8. The property selected to be transformed is the *address* property of the node type *Customer*. The value of this property can contain different values and after finishing this refactoring, it is interesting to apply the refactoring *split property*.

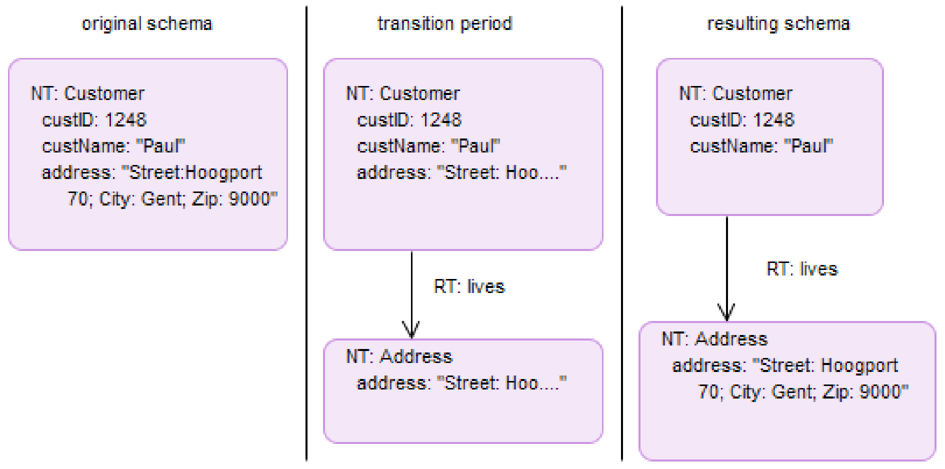

Figure 4.9: Refactoring *Property becoming a node* applied to an instance

*Application:* Applications have to change in order to work with the new node type instead of the old property.

*Additional Notes:* A special case is if the property transformed stores complex or some kind of structured data, like XML. In these cases, after this refactoring, it is important to apply the refactoring *split property* to the new node in order to create more properties and make it easier to query the data, add more detail or make the new properties *non-nullable*. The refactoring *split properties* can only be applied after this refactoring is completed, because, as explained in the beginning of this chapter, two refactorings cannot occur at the same time to ensure the consistency of the operations.

In the beginning of this chapter it was said that all refactorings involving creation of nodes, should also include the creation of a specific *id*. If the data modeler decides it before the creation of the instances, then the surrogate key can be easily introduced together with this refactoring. It can also be done later with the refactoring *introduce surrogate key* which is not included in this catalog because it is very similar to the similarly named refactoring for relational databases.

### 4.4.3 Relationship becoming a node

This refactoring allows transforming all connections of a relationship type in instances of a node type, keeping the existent connections and moving the properties of the relationship type to the node type (*goal*). It can be required before adding more details and connections to a relationship (*motivation*).

*Inputs:*

- Relationship type ($R_X$) to be transformed;
- Name of the new node type ($N_X$);
- Name of the first new relationship type ($R_Y$);
- Name of the second new relationship type ($R_Z$).

*Preconditions:*

- A valid relationship type selected.
- All instances of the selected relationship type can be modified.

*Postconditions:*

- Properties of the new node type have to be the same as the properties of the transformed relationship type;
- Direction of the new relationships ($R_Y$ and $R_Z$) should be the same in all instances;

*Implementation:*

1. Create synchronization functions $f_t$ to run automatically, in order to:
   a. maintain the properties updated (properties from the relationship and properties of the new node type);
   b. maintain data updated, so if a relationship is removed from an instance of the database, the corresponding new node type and its relationships should also be removed and the other way around.

   The function should be invoked in the case of any event update, insert or delete and also when insert new data through applications.

*Steps:*

1. Select a relationship type ($R_X$);
2. Provide the required inputs (name of the new node type and its new relationships ($N_X$, $R_Y$ and $R_Z$));
3. Evaluate preconditions;
4. If preconditions are met:
   a. The new node will be connected to the nodes it was connecting before through relationships of types $R_Y$ and $R_Z$. If these relationship types do not exist in the database, create them using the input data;
   b. Create the new node type that will replace the relationship.
5. [Transition] Create the new nodes and relationships in all instances;
6. [Transition] Use the refactoring *move properties* to move properties from the relationship to the new node type;
7. [Transition] Enable the synchronization functions;
8. [Transition] Set a date to remove the old relationship type and its instances from the database;
9. [Transition] Remove the old relationship type and the synchronization functions;
10. If postconditions are true after the execution of all the steps
    a. Resulting schema will contain a new node type and two new relationship types connecting them instead of the selected relationship type and all data in the database will be modified to meet the resulting schema.

*Metadata:*

In Figure 4.10, an example with abstract data is shown, which can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $N_B$ | $N_B$ |
| $R_X$: $N_A \rightarrow N_B$ | $N_X$ \| *new* |
| ⊦ $p_i$: $T_i$, | ⊦ $p_i$: $T_i$, \| *copy* |
| $p_j$: $T_j$ | $p_j$: $T_j$ \| *copy* |
| | $R_Y$: $N_A \rightarrow N_X$ \| *new* |
| | $R_Z$: $N_X \rightarrow N_B$ \| *new* |

A relationship type ($R_X$) is selected and, using the input parameters, new node types and relationship types are created. After the refactoring, the resulting schema will have the new node type, new relationship types and the properties moved from the old relationship to the new node type.

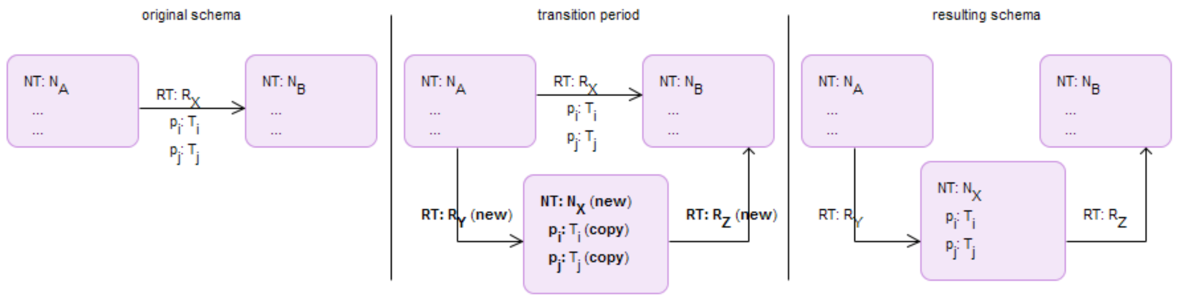

Figure 4.10: Abstract example of refactoring *Relationship becoming a node*

In the metadata example from Figure 4.11, there is a need to add connections to the relationship *married*. The refactoring creates a new node type *Wedding*, with an incoming and an outgoing relationship. The properties of the relationship become properties of the node created. After this refactoring, it is possible to create associations from the new node to other nodes (see for instance the data example in the additional notes).

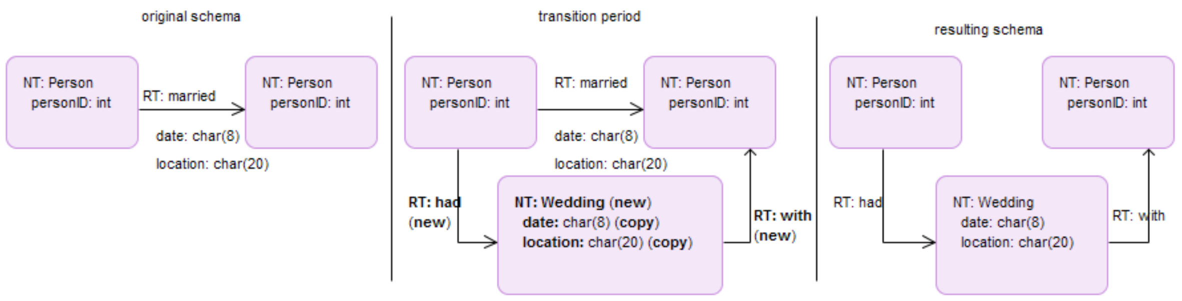

Figure 4.11: Refactoring *Relationship becoming a node* on metadata

*Example:* Figure 4.12 shows how this refactoring would change the data from the graph database.

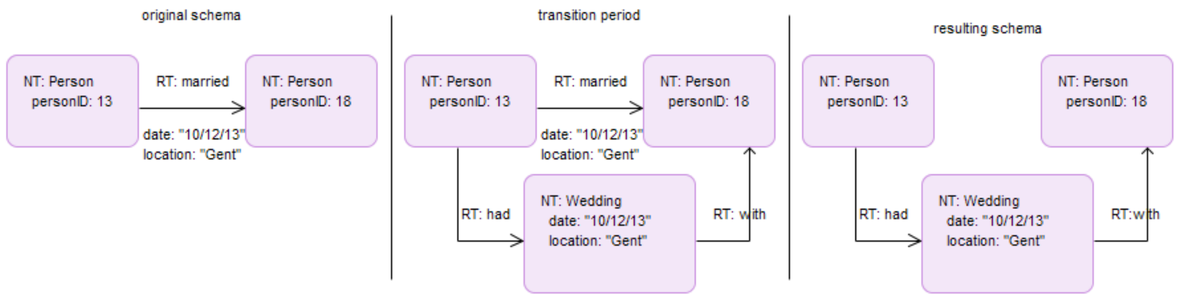

Figure 4.12: Refactoring *Relationship becoming a node* applied to an instance

*Application:* All references to the old relationship should be analyzed and the required modifications should be done to use the nodes from the new node type instead of the old relationship types.

*Additional Notes:* After the Refactoring more properties and connections can be added to the new schema, as in the example of Figure 4.11, a connection with guests that attended the wedding could be created increasing the level of detail of this node type.

## 4.5 Reduction Refactoring

The category Reduction Refactoring is a subdivision of the Structural Refactoring, as the Expansion Refactoring. Three rules are listed in this section, with their importance recognized because of the changes made in the nodes and properties of the graph database schema to improve its structure.

### 4.5.1 Merge properties of a node

This refactoring allows the data modeler to merge properties of a single node type into a new property (*goal*). During the evolution of Graph Database, different data modelers may have added identical properties with different names because the schema is not available or two or more properties may contain additional information and their usage is for the same purpose (*motivation*). The cases that do not require different properties motivate the merge properties refactoring.

*Inputs:*

- Target node type ($N_A$);
- Properties to be merged ($p_j$ and $p_k$);
- Name and type of the property to be created ($p_l$ and $T_l$);
- Transformation function to create the new property: $\Phi$: $(T_j, T_k) \rightarrow T_l$.

*Preconditions:*

- Properties ($p_j$ and $p_k$) have to be in the same node type;
- Properties ($p_j$ and $p_k$) have to be from the same data type ($T_j$==$T_k$) or the transformation function should transform them;
- The identification (key) of the new property ($p_k$) has to be different from the other properties selected to be merged.
- Two properties have to be indicated to the transformation function in each operation.

*Postconditions:*

- Properties $p_j$ and $p_k$ are removed from the database.

*Implementation:*

1. Create a function $\Phi$ to merge the properties
   b. The merge function will tell how the properties will be merged, including the direction of the merge and the resulting data type.

2. Create a synchronization function $f_t$ to run automatically in order to maintain properties (the old ones and the new one – $p_j$, $p_k$ and $p_l$) updated
   c. When updating/deleting one merged property the function must be invoked to copy data from this property to the old properties and also the other way around. Besides that, when inserting new data trough applications using the old/new schema, all the properties have to be updated.

*Steps:*

1. Select nodes of node type $N_A$;
2. Select $p_j$ and $p_k$;
3. Provide name (key) and the data type of $p_l$;
4. Evaluate preconditions;
5. If preconditions are met:
   a. Include the new property ($p_l$) in the node type provided as input
6. [Transition] Use the merge function $\Phi$ to generate and set the value of the new property ($p_l$) to all instances of the selected node type
7. [Transition] Set a date to remove the merged properties($p_j$ and $p_k$) from the database
8. [Transition] Remove the old properties($p_j$ and $p_k$) and the function $f_t$ that synchronizes the properties
   a. If necessary, keep the data from the old properties outside the database
9. If postconditions are true after the execution of all the steps
   a. All nodes (instances) from target node type will have their properties $p_j$ and $p_k$ merged in the property $p_l$.

*Metadata:*

In Figure 4.13, an example with abstract data is shown, which can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $\vdash p_i$: $T_i$, | $\vdash p_i$: $T_i$, |
| $p_j$: $T_j$, | $p_l$: $T_l$ \| *new* |
| $p_k$: $T_k$ | |
| $\Phi$: $(T_j, T_k) \rightarrow T_l$ | |

First, a new property $p_l$ is added to the target node type. Then, the transformation function is used to set values for this new property and a drop date is set for the old properties. The resulting schema will contain only the merged property.

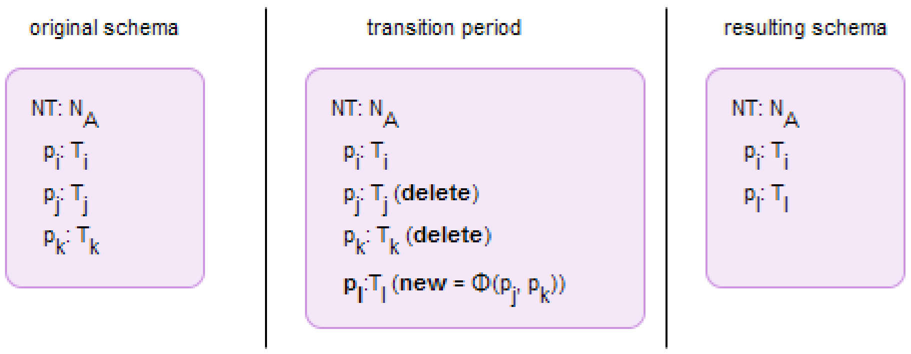

Figure 4.13: Abstract example of *Merge properties* refactoring

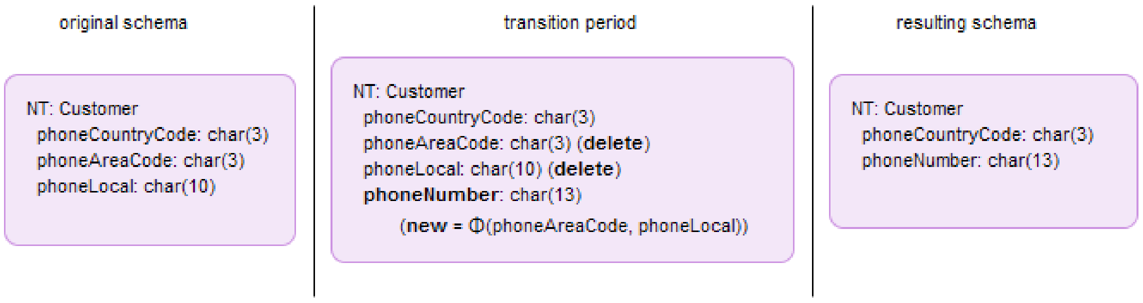
original schema
transition period
resulting schema
NT: Customer
phoneCountryCode: char(3)
phoneAreaCode: char(3)
phoneLocal: char(10)
NT: Customer
phoneCountryCode: char(3)
phoneAreaCode: char(3) (delete)
phoneLocal: char(10) (delete)
phoneNumber: char(13)
(new = Φ(phoneAreaCode, phoneLocal))
NT: Customer
phoneCountryCode: char(3)
phoneNumber: char(13)

Figure 4.14: *Merge properties* refactoring on metadata

In the original schema of Figure 4.14, the customer's phone number is stored using three different properties, but *phoneAreaCode* and *phoneLocal* are always used together. The resulting schema will then contain only one property containing the information of both old properties because, since they are always used together, there is no reason to keep both properties separate. The refactoring included a new property during the transition period called *phoneNumber* which is created through the transformation function and in this case is the concatenation of the two old properties and, after the new property is updated, the old ones are removed.

*Example:* The data example shows a practical application of the merge properties refactoring. In this case, two strings were concatenated in an order determined by the merge function, resulting in the final property *phoneNumber*.

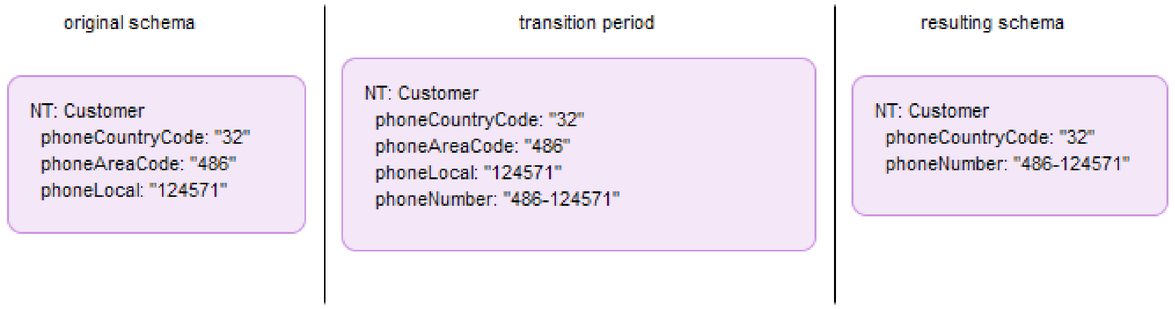

Figure 4.15: *Merge properties* refactoring applied to an instance

*Application:* Change the references to the old property to the new property. Applications have to remove code they were using before the refactoring for merging properties.

*Additional Notes:* Before starting this refactoring, it is important to verify if it is necessary to preserve the properties separated for future use. After the merge of the properties, it can be difficult to separate the data. Still, if it is necessary to separate the data, it is possible to use the reverse refactoring *Split property*.

### 4.5.2 Delete a property of a node

This refactoring allows removing a property from an existing node type (*goal*). After some evolutions, a database may contain properties that are not used anymore. In this situation it is important to remove properties of the corresponding node type for cleanness of the model and to avoid inappropriate use (*motivation*). It is necessary to decide what to do with the existent data before removing the property.

| | |
|---|---|
| *Inputs:* | • Target node type ($N_A$); |
| | • Property to be deleted ($p_j$); |
| *Preconditions:* | • Property is not being used to calculate other properties in the database. |
| *Postconditions:* | • Property removed from database without being used by any application. |

*Implementation:* There is no need to create support code for this refactoring.

*Steps:*

1. Select target node type ($N_A$);
2. Select a property($p_j$) to be removed;
3. [Transition] Set a date to remove the selected property ($p_j$);
4. [Transition] Check if the property is being used in any other place before the removal (index, functions, calculated properties, …) and, if it is, stop the refactoring;
5. [Transition] Remove the selected property ($p_j$);
   a. If necessary, keep the data from the removed properties outside the database
6. If postconditions are true after the execution of all the steps
   a. All nodes (instances) from target node type will not have the selected property ($p_j$) anymore.

*Metadata:*

In Figure 4.16, an example with abstract data is shown, which can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $\vdash p_i: T_i,$ | $\vdash p_i: T_i$ |
| $p_j: T_j$ | |

A property $p_j$ is selected to be removed from the target node type $N_A$. During the transition period a date to remove this property is set and all required changes are made. After the refactoring, the selected property is removed from the schema of the target node type.

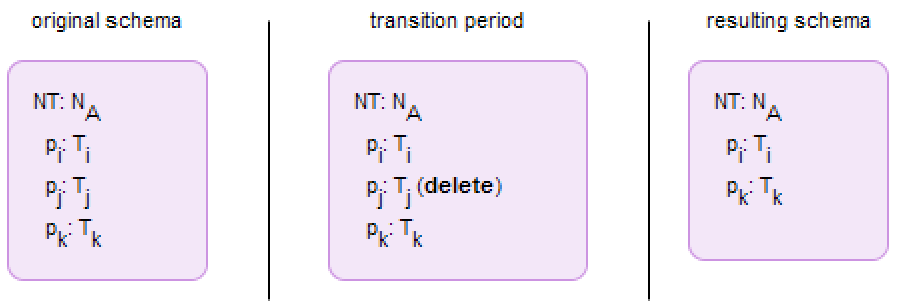

Figure 4.16: Abstract example of refactoring *Delete a property of a node*

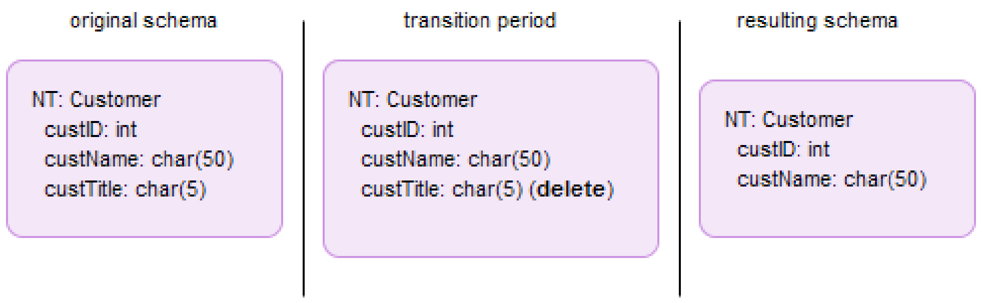

Figure 4.17: Refactoring *Delete a property of a node* on metadata

In the metadata of Figure 4.17, the original schema has a property *custTitle* that is not used anymore. During the transition period, the drop date will be set and developers will be alerted about this change, but the property will still exist in the database. Before the final removal, it is useful to rename the property in order to test if the applications will keep running after the deletion. After the refactoring, the resulting schema will be clean without the unused property *custTitle*.

*Example:* The data example in Figure 4.18 shows a practical application of the refactoring *delete a property from a node*. The property that was not used anymore was removed from all instance nodes after the refactoring.

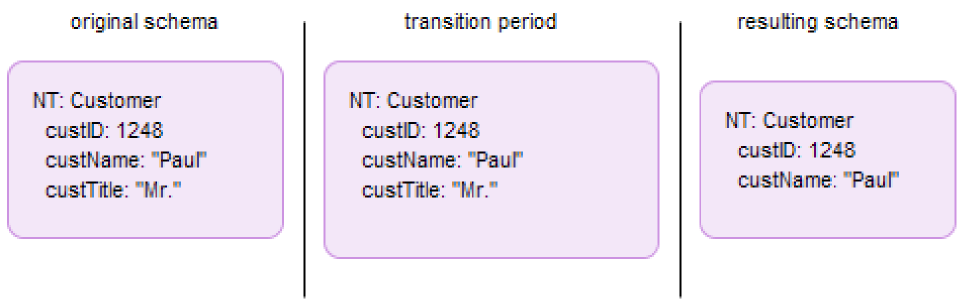

Figure 4.18: Refactoring *Delete a property of a node* applied to an instance

*Application:* Remove all the references to the removed property on the applications.

*Additional Notes:* A very similar refactoring can be derived from this refactoring: *delete a property of a relationship type*.

### 4.5.3 Merge Nodes

This refactoring allows the contraction of two node types intimately connected (*goal*). Two node types are so intimately connected that they could be represented together as another node type, with a distinct name (*motivation*).

| | |
|---|---|
| *Inputs:* | • Target nodes type ($N_A$ and $N_B$);<br>• New node type ($N_{AB}$);<br>• Existent relationships in the merged nodes that will be kept in the new node (by standard all the relationships will be kept, but the user should be able to select which ones he wants to keep). |
| *Preconditions:* | • Nodes have to be intimately connected (minimum and maximum multiplicities 1:1) |
| *Postconditions:* | • New node type ($N_{AB}$) containing all properties existent in the merged nodes;<br>• Nodes of the merged node types do not exist in the database. |

*Implementation:*

1. Create a function Φ to create the new node type, moving the properties from the old node types and changing their names before adding them to the new node, when required;
2. Create synchronization functions $f_t$ to run automatically in order to maintain properties (belonging to the old node types and the new one) updated
   a. When update/delete/insert a property in the new or the old node types, a function must be invoked to copy data from one property to another. Also when inserting new data trough applications using old/new schema, all the properties have to be updated.

*Steps:*

1. Select nodes type ($N_A$ and $N_B$) to merge;
2. If preconditions are met:
3. [Transition] Create a new node type ($N_{AB}$) that will be the resulting node;
4. [Transition] Move properties of both nodes selected to be merged to the new node, using the function Φ to treat properties;
5. [Transition] Move all the relationships or the ones selected by the user from the old nodes to the new node;
6. [Transition] Support code to maintain new and old schema updated
7. [Transition] Set a date to remove the old nodes from the database;
8. [Transition] Remove the old nodes and the functions $f_t$ that synchronizes it
9. If postconditions are true after the execution of all the steps
   a. All nodes (instances) from selected node types will be merged into a new node type.

*Metadata:* In Figure 4.19, an example with abstract data is shown, which can also be explained with the following formulation:

Given:

$N_A$

⊦ $p_i$: $T_i$

$N_B$

⊦ $p_j$: $T_j$

$R_X$: $N_A \rightarrow N_B$

⊦ $p_k$: $T_k$

Resulting Schema:

$N_{AB}$ | *new*

⊦ $p_i$: $T_i$, | *copy*

$p_j$: $T_j$ , | *copy*

$p_k$: $T_k$ | *copy*

A node type $N_A$ and one node type $N_B$ connected by a relationship $R_X$ are selected to be merged into a new node type. The properties of both nodes and of the relationship between them will be copied to the new node type $N_{AB}$. There will be a function to treat properties which have the same key and synchronization functions to maintain the old and the new schema updated. Unless the user specifies it different, all connections from the merged nodes will be moved to the new node ($N_{AB}$). After the new node type is created and all instances of the selected node type are changed, a date to drop the merged nodes will be set and the old schema can later be removed from the database. Figure 4.20 shows how this refactoring would be applied to the schema. *Customer* and *Account* are intimately connected and could be merged into the same node type.

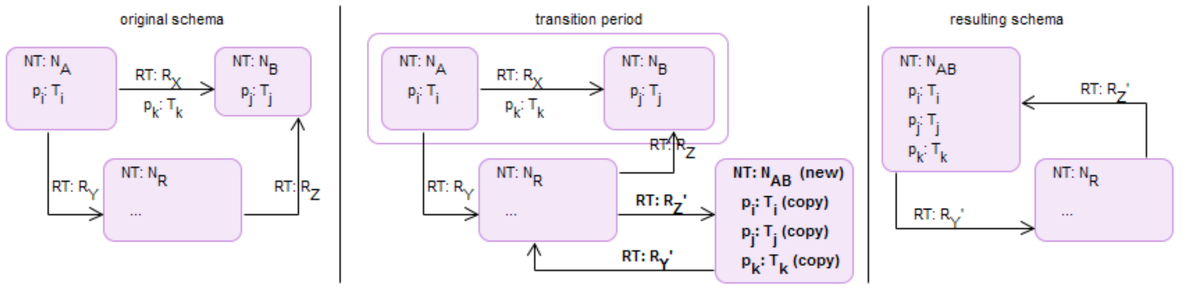

Figure 4.19: Abstract example of refactoring *Merge nodes*

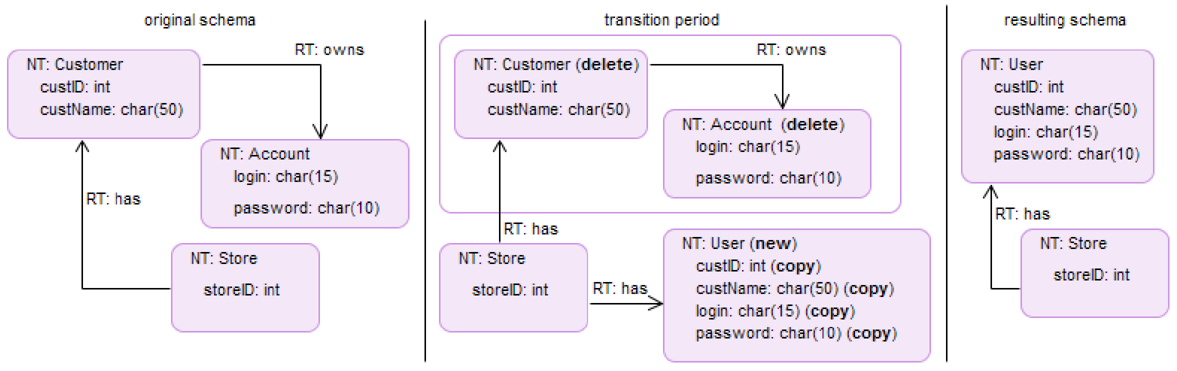

Figure 4.20: Refactoring *Merge Nodes* on metadata

*Example:* In the example, it is not necessary to have two different nodes to store the data from a user. The instances of node type *Customer* could be merged with instances of node type *Account* into a new instance of node type *User*. All nodes customer connected to account through a relationship has will be transformed in a node of node type user.

Figure 4.21 shows an example of this refactoring, where the relationship existent between the merged node *Customer* and the node *Store* was copied to the nodes of type *User*. The relationship type *has* between the merged nodes do not have to be maintained.

Before starting the refactoring, it is necessary to evaluate and ensure that the old information will not be required again, since the details of the old schema will be lost after this refactoring.

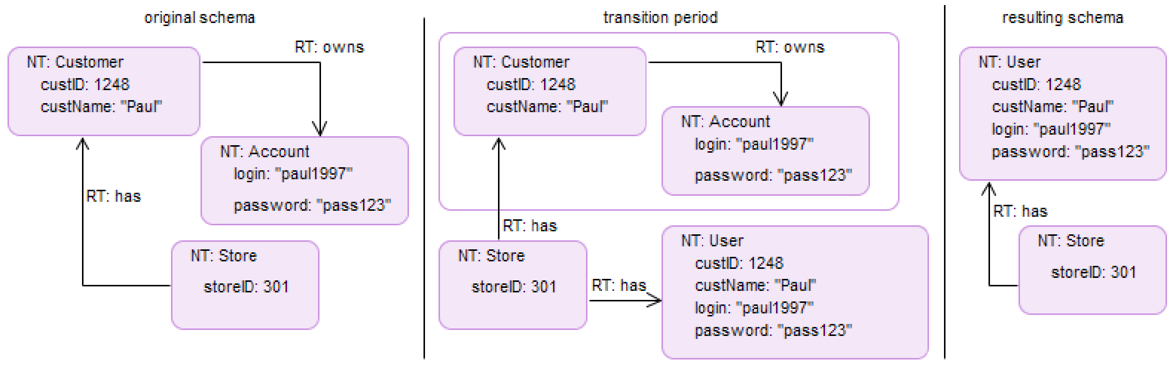

Figure 4.21: Refactoring *Merge Nodes* applied to an instance

*Additional Notes:* If nodes selected to be merged contains the same properties, the transformation function will rename those properties even if their information are not valuable anymore. After this refactoring, it is important to evaluate the properties copied and apply the refactorings *merge properties* or *delete properties* if necessary.

## 4.6 Improvement Refactoring

Improvement Refactorings improve the logic of the database. In this category the rules bring small changes to the database schema that can improve query performance and its results. The refactoring rules presented here are: Introduce calculated property, Shorten path, Invert direction of a relationship and Move a property of a node to another node.

### 4.6.1 Introduce calculated property

This refactoring introduces a new property based on calculations involving data of one or more node types (*goal*). There are cases in which applications make calculations based on values from the database each time it queries the data. In this case, the calculation is repeated in each query. This refactoring unifies the calculation formula to all applications and avoids repetition of calculation (*motivation*). This improves the application performance by providing prepopulated values for a given property derived from other properties.

*Inputs:*

- Target node type ($N_A$);
- Name and type of the property to be created ($p_{CALC}$ and $T_{CALC}$);
- Properties used to calculate the value of $p_{CALC}$;
- Transformation function to create the new property, given the properties that will be used: $\Phi: (T_N) \rightarrow T_{CALC}$.

*Preconditions:*

- Properties selected to calculate the value of the property have to belong to the target node type or to a node type that has a relationship with the target node type.

*Postconditions:*

- Calculated property has to be set according to the selected properties for the calculation;
- The identification (key) of the new property ($p_{CALC}$) has to be different from the other properties in the target node.

*Implementation:*

1. There is no need to create the function to create the calculated property because this function will be given as an input, but a function to keep the calculated property updated is required, which should be invoked for any event update, insert or delete occurring in the properties used to calculate $p_{CALC}$.

*Steps:*

1. Select target node type;
2. Select properties used to make the calculations;
3. Provide name(key) and the data type of $p_{CALC}$;
4. Evaluate preconditions;
5. If preconditions are met:
    a. Include the new property ($p_{CALC}$) in the target node type provided as input and in all instances of this type.
6. [Transition] Use the transformation function $\Phi$ to generate and set the values of the new calculated property ($p_{CALC}$) to all instances of the selected node type;
    a. The calculate function will keep active in the database, being used as batch process or as an automatic function triggered by an event.
7. If postconditions are true after the execution of all the steps
    a. All nodes (instances) from target node type will have a new calculated property $p_{CALC}$.

*Metadata:*

In Figure 4.22, an example with abstract data is shown, which can also be explained with the following formulation:

Given:

$N_A$

$\vdash p_k: T_k,$

$P_l: T_l$

$[N_B]$

$\vdash p_i: T_i,$

$p_j: T_j$

$\Phi: (T_N) \rightarrow T_{CALC}$

Resulting Schema:

$N_A$

$\vdash p_k: T_k,$

$p_l: T_{l,}$

$P_{CALC}: T_{CALC}$ | *new*

$[N_B]$

$\vdash p_i: T_i,$

$p_j: T_j$

A new property $p_{CALC}$ is added to the target node type and calculated according to the transformation function $\Phi$. In the abstract example, $N_B$ is optional because the calculated property can be computed using only properties from the same node type. A process to keep the calculated property updated has to be defined, for example, using the calculate function any time one of the properties used for the calculation is updated. The resulting schema will contain the calculated property.

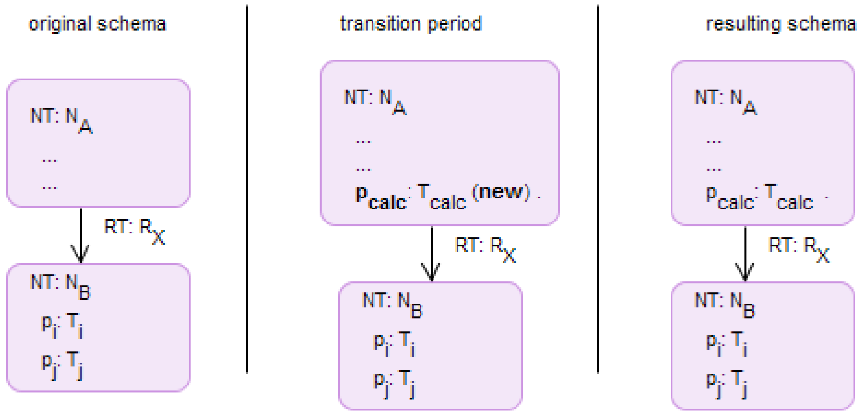

Figure 4.22: Abstract example of refactoring *Introduce Calculated Property*

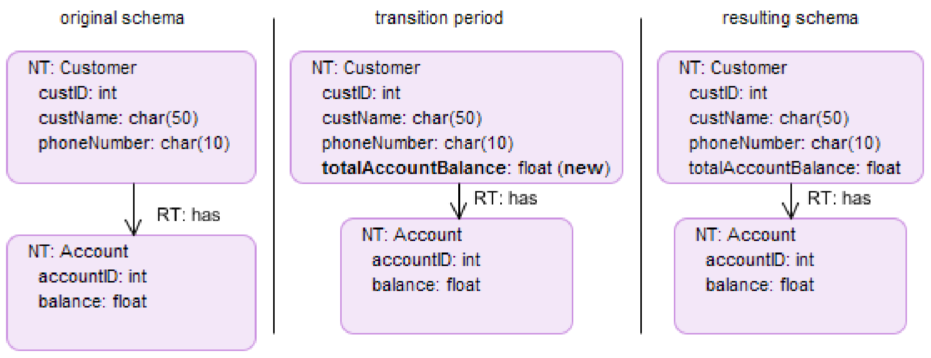

Figure 4.23: Refactoring *Introduce Calculated Property* on metadata

In the example, a customer can have many accounts and, before applying the refactoring, the applications would have to find his accounts and calculate the balance sum of all of them. After the refactoring and an automatic function to update the *totalAccountBalance*, the applications only have to query this new property.
The transition period it is only necessary to announce the changes to the users of this database, and it is when the new property is added to the nodes of a node type in order to store the calculation result and the code to update new properties (batch or trigger) is used.

*Example:* The data example in Figure 4.24 shows a customer that has more than one account. In order to calculate his *totalAccountBalance* the calculate function has to query all his accounts and sum the balance in all of them. After the refactoring, all nodes of this node type will have a new calculated property.

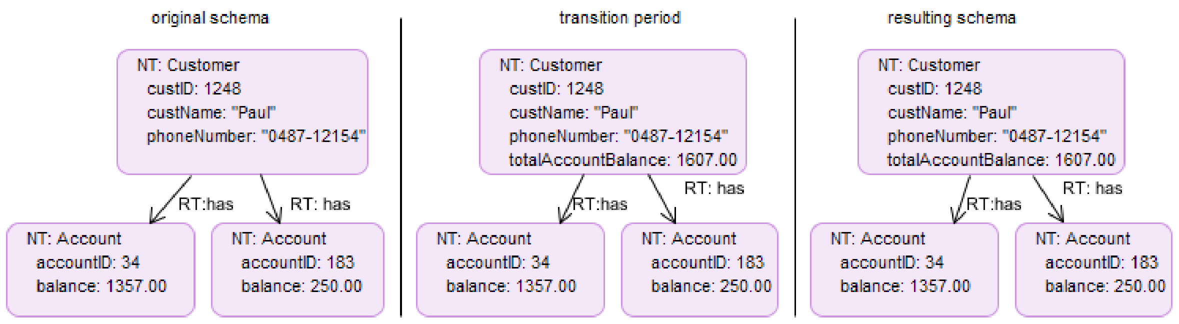

Figure 4.24: Refactoring *Introduce Calculated Property* applied to an instance

*Application:* Remove from the application methods that calculate the value before the refactoring. Instead, applications have to query the new calculated property.

*Additional Notes:* The transformation function Φ that calculates the value of the calculated property received as input can be used in two different ways: batch process that will calculate the property to all nodes or as an automatic function that calculates the value of the property after a determined event, such as an update.

### 4.6.2 Move property of a node (to a relationship or to another node)

This refactoring allows the migration of a property from a specified node type to a node from another type or to a relationship (*goal*). The creation of a new node type due to a database evolution can require the migration of a property from an old node type to a new one. This refactoring makes this reorganization allowing the new nodes to add properties containing coherent value (*motivation*).

*Inputs:*

- Property to be moved ($p_i$);
- Node type ($N_A$) that contains the property;
- Node type ($N_B$) or Relationship type ($R_X$) that will receive the property.

*Preconditions:* -

*Postconditions:*

- Property $p_i$ from $N_B$ (moved property) has to be from the same data type and store the same value than property pi from $N_A$ (original property);
- Moved property should not be similar to other properties of the new node type it belongs;
- The identification (key) of the moved property ($p_i$) has to be different from the other properties belonging to the new node type.

*Implementation:*

1. Create a synchronization function $f_t$ to run automatically in order to maintain properties (the one in the old node type and the moved ones) updated.
    a. The function should run automatically for any event update/insert/delete occurring in the property at its old or new location maintaining both properties updated with the same value.

*Steps:*

1. Select target node type ($N_A$);
2. Select a property($p_i$) to be moved;
3. Select node type ($N_B$) or relationship type ($R_X$) to where the property ($p_i$) will be moved;
4. Evaluate preconditions;
5. If preconditions are met:
    a. Create a new property ($p_i$) in the selected node/relationship type($N_B$/$R_X$);
    b. The new property will have the same name of the selected property. If the new location already contains a property with the same name, a prefix or a suffix can be added (e.g.: "$p_i$_1") and the name can be changed afterwards;
    c. It is also important try to find properties used together or similar ones and consider applying the refactoring *merge properties* after the current refactoring.
6. [Transition] Update the property at the new location with the existent data of the old location (it can be done applying the synchronization function $f_t$ manually to all nodes);
7. [Transition] Enable the synchronization function $f_t$;
8. [Transition] Set a date to remove the moved property ($p_i$) from the old location in the schema and all instances;
9. [Transition] Remove the old property($p_A$) and the synchronization function $f_t$;
10. If postconditions are true after the execution of all the steps
    a. All properties in the old location will be moved to the new locations.

*Metadata:*

In Figure 4.25, an example with abstract data is shown. The property $p_k$ is selected to be moved from node $N_A$ to $N_B$, which can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $\vdash p_i: T_i,$ | $\vdash p_i: T_i,$ |
| $p_j: T_j,$ | $p_j: T_j,$ |
| $p_k: T_k$ | $N_B$ |
| $N_B$ | $\vdash p_l: T_l,$ |
| $\vdash p_l: T_l,$ | $p_m: T_m$ |
| $p_m: T_m$ | $p_k: T_k$ \| *copy* |

The selected property ($p_k$) is created at the new location and removed from the old one. The examples are showing only properties moving from one node type to another node type, but properties can move between relationships and also from nodes to relationships.

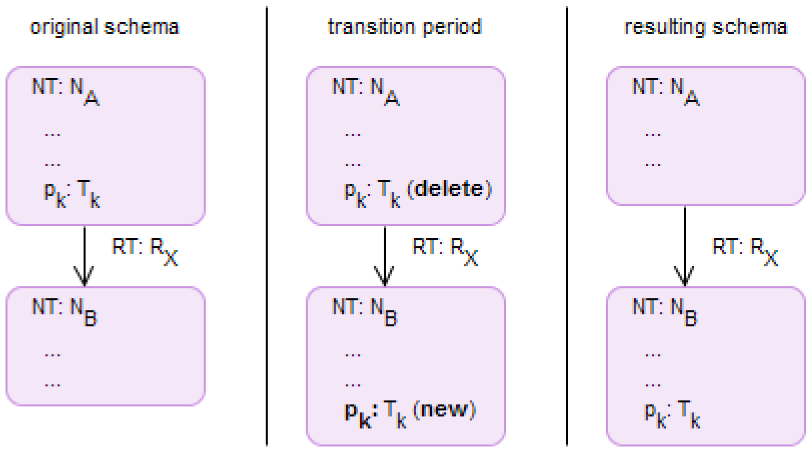

Figure 4.25: Abstract example of refactoring *Move property of a node*

In the metadata example of Figure 4.26, the property balance was stored in the node type *Customer*, but for some reason, for example, a change in the database that allows a customer to have multiple accounts, the property balance should be moved to the node type account. In order to keep both properties updated, synchronization functions that run automatically according to some event have to be developed and applied to both node types. The refactoring will be complete after the removal of the old property and the support code.

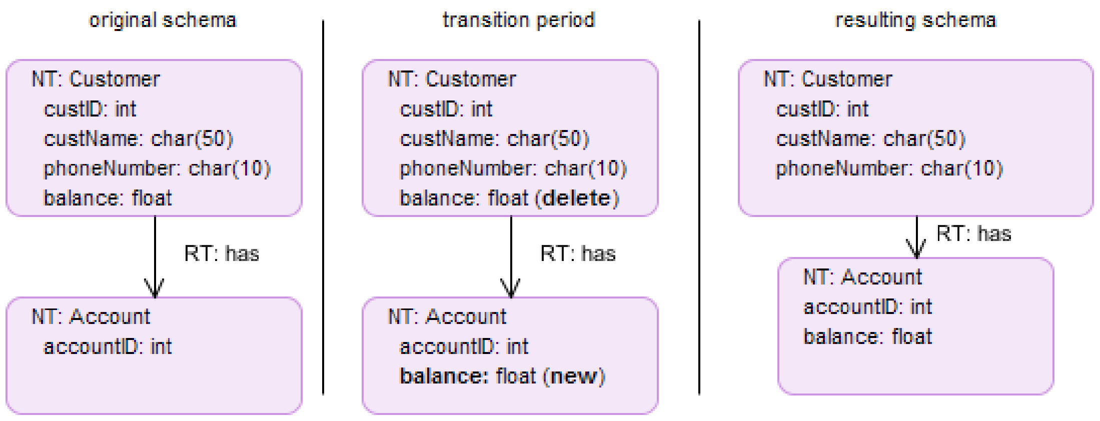

Figure 4.26: Refactoring *Move property of a node* on metadata

*Example:* The node *Customer* in Figure 4.27 was storing the balance of a customer before the refactoring. After the change in the database allowing a customer to have more than one account, the property *balance* had to be moved to the node type *Account*, since a customer can have a different balance in each account. During the transition period, the properties exist in both locations; and, after the refactoring, all the nodes of node type *Account* will have the property *balance*.

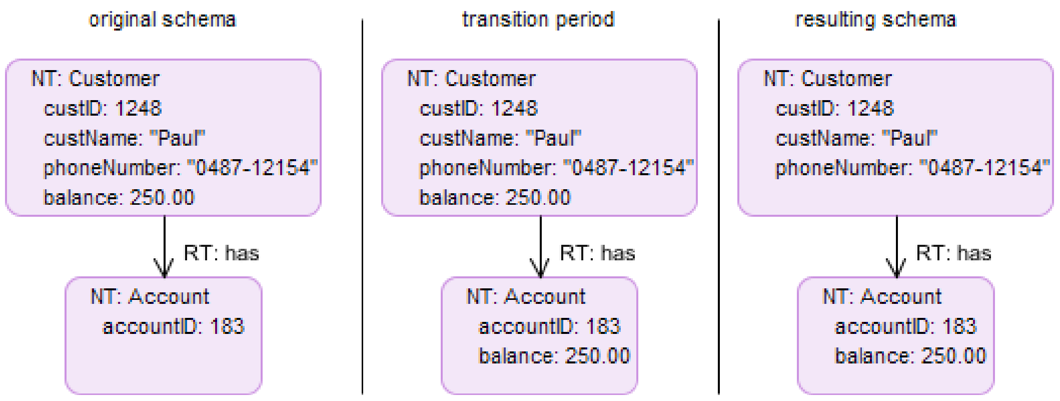

Figure 4.27: Refactoring *Move property of a node* applied to an instance

*Application:* All references to the property in the old node type have to be modified to refer to the node type that the property currently is.

*Additional Notes:* To remove the property of the old location, the refactoring *delete property* of a node type can be used to ensure that the property is not being used in any other location and avoid future problems. Again, if there is a similar property to the one moved at the new location, the refactoring *Merge Properties* has to be considered. The data modeler should recognize if the semantic of the property will be the same after moving the node before applying the refactoring.

### 4.6.3 Invert Direction of a relationship

The goal of this Refactoring is to switch the origin (initial point) and the destiny point of a selected edge, removing an old relationship type and create a new one with the same properties and connections, but with inverted direction (*goal*). A relationship direction can be wrong in the schema of a database, causing an error of consistency or a change in the business rules can happen. Both cases would require the inversion of the relationship direction (*motivation*).

*Inputs:*
- Relationship type ($R_X$);
- Properties of the relationship type that will be kept in the new direction (the user selects which ones will be kept).

*Preconditions:*
- The direction of the relationship type ($R_X$) has to be previously defined.

*Postconditions:* -

*Implementation:*
1. Create a synchronization function $f_t$ to run automatically in order to maintain properties of both relationships updated after any insert/update event. It is also important to think about what to do if the old relationship is deleted from the database: delete or keep the relationship type with the new direction.

*Steps:*
1. Select a relationship type ($R_X$);

2. Evaluate the preconditions;
3. [Transition] Create a new relationship type ($R_Y$) with the same properties and connections of the relationship type selected in 1, but with inverted direction.
4. [Transition] For every relationship of type $R_X$ create a new connection of type $R_Y$ connecting the same old nodes;
5. [Transition] Use the refactoring move properties to move properties from old relationships to the new ones according to the input data;
6. [Transition] Mark all relationships of type $R_X$ to be removed from the database.
7. Remove the relationships of type $R_X$.
8. If postconditions are true after the execution of all the steps
   b. All nodes that were connected by the relationship $R_X$ will be then connected by the relationship $R_Y$ in the inverse direction.

*Metadata:*

Figure 4.28 shows an example with abstract data, which can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $\vdash p_i$: $T_i$, … | $\vdash p_i$: $T_i$, … |
| $N_B$ | $N_B$ |
| $\vdash p_j$: $T_j$, … | $\vdash p_j$: $T_j$, … |
| $R_X$: $N_A \rightarrow N_B$ | $R_Y$: $N_B \rightarrow N_A$ \| *new* |
| $\vdash p_k$: $T_k$, … | $\vdash p_k$: $T_k$, … \| *copy* |

A new relationship type is created based on the relationship type selected to be inverted, with the same properties and connecting the same nodes, but with an inverted direction. After creating the new relationship to all required nodes, the relationship with the old direction can be removed.

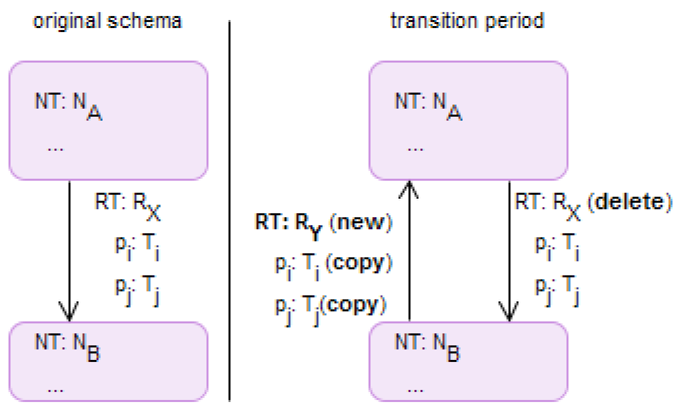

Figure 4.28: Abstract example of refactoring *Invert Direction of a Relationship*

In the example of Figure 4.29, there is a relationship type *likes* connecting a *Customer* to a *Product*. However the direction of the edge shows that the product is the one who likes the customer. Using this refactoring, it is possible to invert the direction of the arc and thus, the relationship type *likes* would have the origin in *Customer* instead of *Product*.

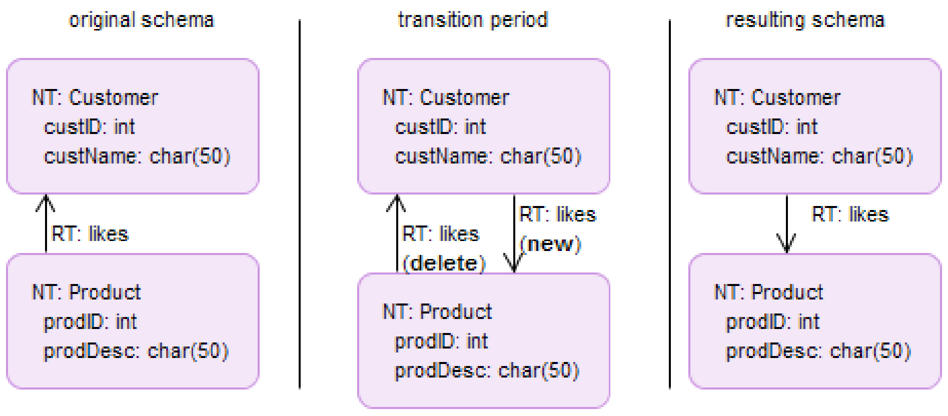

Figure 4.29: Refactoring *Invert Direction of a Relationship* on metadata

*Example:* The instances of the database would have exactly the same change from Figure 4.29.

*Application:* Applications working with this relationship should be reviewed and, if required, change to use the new schema.

*Additional Notes:* Old queries should be reviewed and, if necessary, the direction of their relationships modified. It is important to verify if this change will not imply in semantic changes to the database, otherwise, it cannot be applied.

### 4.6.4 Shorten Path

The goal of this refactoring is to create an edge (relationship) between two nodes, based on other relationships, in order to reduce the path between them (*goal*). It reinforces the data quality because the refactoring will allow multiple paths between two nodes and make some queries easier (*motivation*).

*Inputs:*

- Origin and Destiny Nodes ($N_C$ and $N_B$);
- Path between two nodes that can be provided as query containing the desired nodes and relationships between them;
- A Relationship type ($R_Z$) used to reduce the path;
- Its direction and which nodes it will connect in order to reduce the path [optional].

*Preconditions:*

- The path between origin and destiny nodes should be a valid path in the database;
- Relationship type $R_Z$ should exist in the database;
- The new relationship type ($R_Z$) should be different from the others described in the matched pattern.

*Postconditions:*

- Valid relationship between two nodes reducing the path and improving the quality of the database.

*Implementation:* There is no need to create support code for this refactoring.

*Steps:*

1. Select the relationship type provided as input or, if it does not exist, include the new relationship type in the database;
2. Query all nodes that meet the specific pattern provided as input;
3. Evaluate preconditions;
4. If preconditions are met:
   a. Connect the nodes matching the provided pattern using the relationship type created/select in step 1;
5. If postconditions are true after the execution of all the steps
   a. All nodes (instances) matching the provided pattern will have an extra relationship in order to reduce the path and improve queries.

*Metadata:*

In Figure 4.30, an example with abstract data is shown, where a $R_Z$ is created between nodes $N_C$ and $N_B$, since they meet a specified pattern. The abstract data can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $N_B$ | $N_B$ |
| $N_C$ | $N_C$ |
| $R_X$: $N_C \rightarrow N_A$ | $R_X$: $N_C \rightarrow N_A$ |
| $R_Y$: $N_A \rightarrow N_B$ | $R_Y$: $N_A \rightarrow N_B$ |
| | $R_Z$: $N_C \rightarrow N_B$ \| *new* |

The example in Figure 4.31 has three nodes of type person connected by two relationships *is_parent*. The path between two nodes could be shortened and instead of querying data using the middle node, a new connection could be added skipping it.

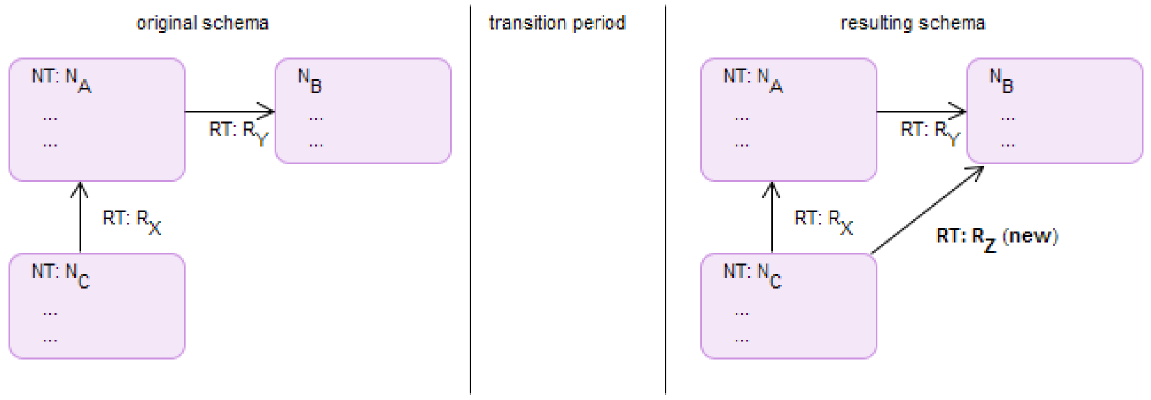

Figure 4.30: Abstract example of refactoring *Shorten Path*

In this example the two relationships are the same, but there are cases where the pattern will involve different relationships and the new relationship will be created based on them. For example a customer that rates a restaurant located inside a hotel can have a connection saying that he had been to this hotel shortening the path between the nodes hotel and customer. There is no need of a transition period because the only change will be the new relationship between two node types.
Through this refactoring, the connection between the two nodes is reinforced.

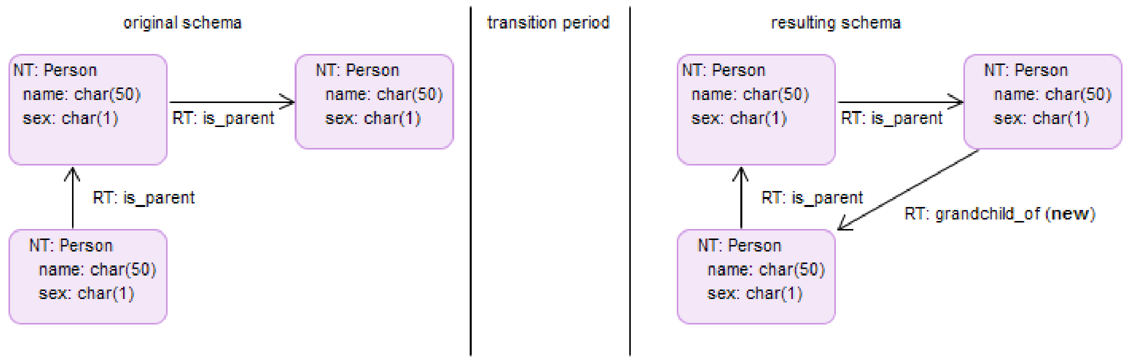

Figure 4.31: Refactoring *Shorten Path* on metadata

*Example:* The data example in Figure 4.32 shows a practical application of the refactoring *shorten path*.

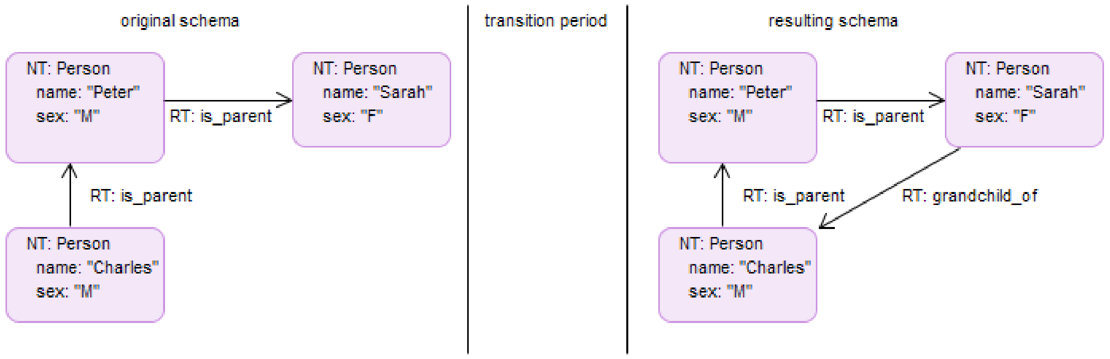

Figure 4.32: Refactoring *Shorten Path* applied to an instance

*Application:* The application can replace the queries meeting the pattern to retrieve the data using the new relationship.

*Additional Notes:* -

## 4.7 Data Quality Refactoring

In the data quality refactoring only two refactorings were introduced. Since this category aims to ensures the consistency and usage of the values stored within the database, the rules of this section make use of metadata *nullable* and *default*, to increase the number of rules in this category, new metadata, such as *unique*, should be introduced.

### 4.7.1 Introduce Default Values

This refactoring lets the database provide a default value for an existing property of a node type (*goal*). There are cases where a property has to exist and null values or values with no meaning cannot be used. To solve this issue, this refactoring sets a default value for a property, so it will always have a value (*motivation*). The default value has to be useful for all the applications using the database.

| | |
|---|---|
| *Inputs:* | • Target node type ($N_A$); |
| | • Property($p_i$) to be set with the default value; |
| | • Default value. |
| *Preconditions:* | • Default value has to be from the same data type as the property selected. |
| *Postconditions:* | • Default value should be applied to all corresponding properties, ensuring there is no property storing null values or values without meaning |

*Implementation:*

1. Create a function $f_t$ to apply the default value to the property when a new node from the target node type is created.

*Steps:*

1. Select node type ($N_A$);
2. Select a property($p_i$);
3. Provide the default value ("value") for the selected property;
4. Evaluate preconditions;
5. If preconditions are met:
   a. Include the constraint default value with the provided value to the selected property ($p_i$).
6. Update existent nodes of the selected node type in which the property is null or that do not have this property.
7. Keep the function $f_t$ active in the database for later updates;
8. If postconditions are true after the execution of all the steps
   a. All nodes (instances) of the target node type will contain a default value for the selected property and it will never be null.

*Metadata:*

In Figure 4.33, an example with abstract data is shown, which can also be explained with the following formulation:

Given:
$N_A$
$\vdash p_i$: $T_i$,
$p_j$: $T_j$

Resulting Schema:
$N_A$
$\vdash p_i$: $T_i$,
$p_j$: $T_j$ {default = ‘value’}

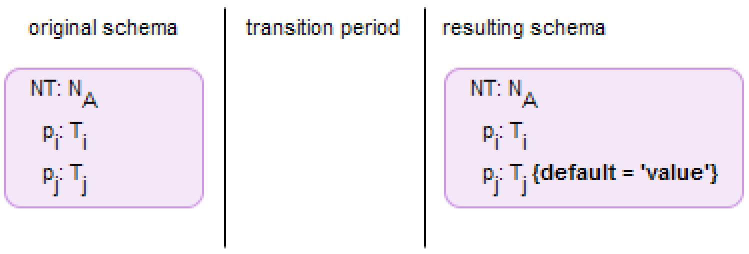

Figure 4.33: Abstract example of refactoring *Introduce Default Values*

The only change in the resulting schema of Figure 4.33 is the creation of a new constraint containing the default value for the selected property $p_j$.

In the example of Figure 4.34, the node type *Customer* has a property storing the status of a customer. After the refactoring, in the case of creating a new node without provide the value for the property *status*, the default value ‘Active’ will be applied.

This refactoring does not need a transition period, it is sufficient to alter the metaschema setting a default value to the property.

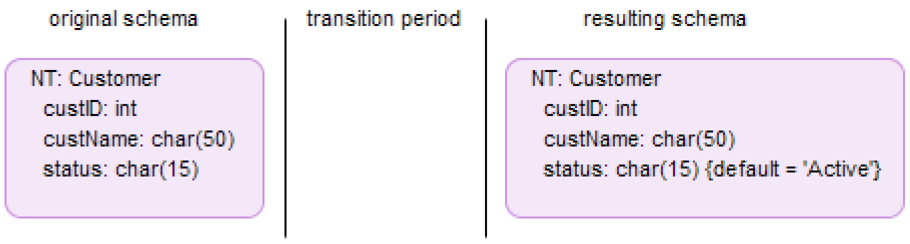

Figure 4.34: Refactoring *Introduce Default Values* on metadata

*Example:* The original data of Figure 4.35 did not have the status of the customer, after the refactoring, this node contains a property *status* with the default value for it.

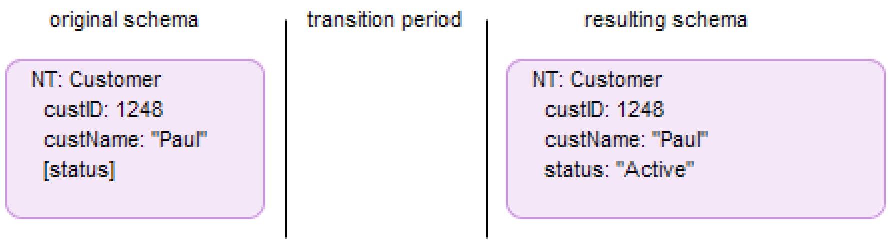

Figure 4.35: Refactoring *Introduce Default Values* applied to an instance

*Application:* If an application has code to treat null values, this treatment has to change to use the default value.

*Additional Notes:* Not all graph database systems allow setting a default value for a property and that is the reason an alternative notation, without the need of a constraint, is used here to show that a property will receive a default value.

### 4.7.2 Make properties non-nullable

This refactoring changes an existing property such that it does not accept any null values (*goal*). The refactoring is used to reinforce a business rule that requires one property to be non-null or when it is desired to remove the code treating properties which were not provided by the application (*motivation*). It can be applied to properties of a node or of a relationship type.

| | |
|---|---|
| *Inputs:* | • Target node type ($N_A$); |
| | • Property($p_j$) to be set as *not null*. |
| *Preconditions:* | - |
| *Postconditions:* | • Selected property cannot be null in all nodes in the database. |

*Implementation:* -

*Steps:*

1. Select target node type ($N_A$);

2. Select a property($p_j$);
3. Update all existing nodes of the selected node type that does not have this property to a valid value using a determined value or the refactoring *introduce default value*
4. Include the constraint not null to the select property;
5. If postconditions are true after the execution of all the steps:
   a. All nodes (instances) belonging to the target node type will contain the non nullable property.

*Metadata:*

In Figure 4.36, an example with abstract data is shown, which can also be explained with the following formulation:

| Given: | Resulting Schema: |
|---|---|
| $N_A$ | $N_A$ |
| $\vdash p_i$: $T_i$, | $\vdash p_i$: $T_i$, |
| $p_j$: $T_j$ | $p_j$: $T_j$ {not null} |

The only change in the resulting schema of Figure 4.34 is the creation of a *not null* constraint for the selected property $p_j$.

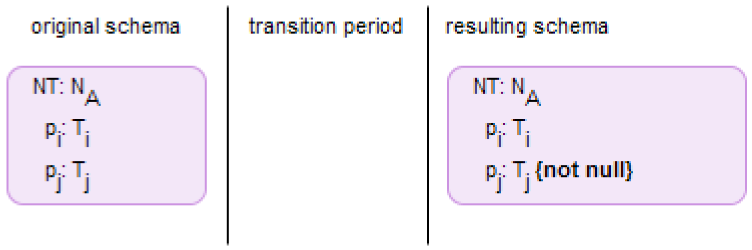

Figure 4.36: Abstract example of refactoring *Make Property non-nullable*

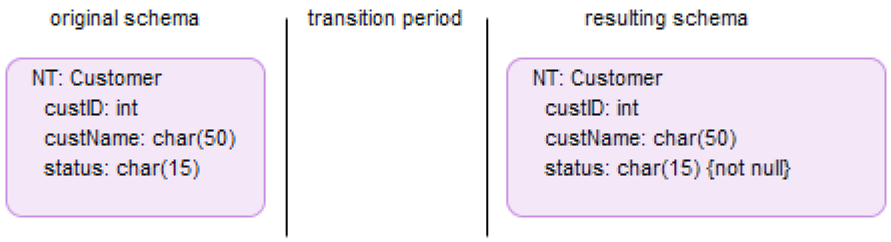

Figure 4.37: Refactoring *Make Property non-nullable* on metadata

Before the refactoring, some applications could insert a customer without providing the status value. After the refactoring, the property *status* has a constraint *not null*, requiring this information from all applications.
To perform this refactoring it is necessary to add some kind of constraint to the property nullable in the optional schema and set it as *not null*. Then, whenever an application inserts or updates a node/relationship containing a non-nullabe property, the value of this property should be provided. One useful technique is to assign a default value using the refactoring: *introduce default values* to all properties that became non-nullable.

*Example:* The data example in Figure 4.38 shows a node that did not have the property *status* before the refactoring, since it was not required. After the refactoring to make the property non-nullable, the property *status* was added to all the instances of the node type *Customer* storing the default value provided before.

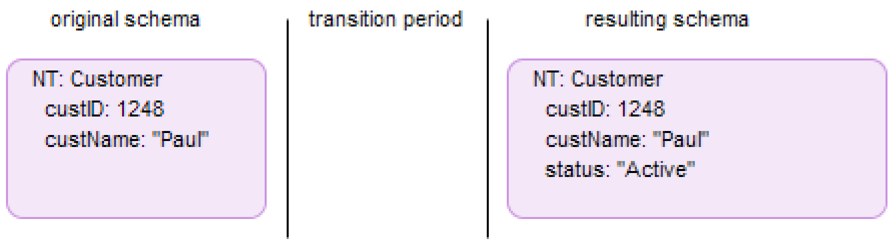

Figure 4.38: Refactoring *Make Property non-nullable* applied to an instance

*Application:* Applications should also provide a valid value for this property because after this refactoring, the property will have to exist and cannot contain null values.

*Additional Notes:* Not all graph databases allow setting a property as *non-nullable* and that is the reason why an alternative notation, without the need of a constraint, is used here to show that a property cannot be null.

## 4.8 Final Considerations

This chapter brought a catalog of refactoring rules for graph databases. An overview of the refactoring rules created is showed in Table 4.2.

Table 4.2: List of Refactoring Rules

| Expansion Refactoring | Improvement Refactoring |
|---|---|
| Split property of a node | Introduce calculated property |
| Property becoming a node | Move property of a node |
| Relationship becoming a node | Invert Direction of a relationship |
| | Shorten Path |
| **Reduction Refactoring** | **Data Quality Refactoring** |
| Merge properties of a node | Introduce Default Values |
| Delete a properties of a node | Make properties non-nullable |
| Merge Nodes | |

The refactorings were developed to be as simple as possible and, if there is a complicated case that does not meet the preconditions of a refactoring and you still want to apply the refactoring, other refactorings have to be applied to resolve and simplify the cases.

In very complex cases or cases with high coupling, it is not recommended to apply a refactoring and that is the reason why some situations were not covered in this document. In cases extremely complex the refactoring should be prevented.

Although a formal implementation to validate the rules was not executed, the rules can be useful for data modelers looking for an initial orientation on how to evolve a graph database which uses the optional schema. Furthermore, the creation of a catalog with best practices encourages the growing of this catalog with new rules.

# 5. CONCLUSION

This work provides support for applying refactoring rules in graph database enabling an organized evolution of the graph database. The rules can be useful for data modelers looking for a primary orientation on how to evolve a graph database using an optional schema without compromising the flexibility of it and enabling an organized evolution of graph databases. The rules created are generic and can be adapted to any graph database.

In past works of the group (FONSECA & CAMOLESI Jr., 2015) similar results were presented, which included the specification guideline, the classification of the refactoring rules and initial refactoring rules examples. This line of research is still new and the catalog does not end with the rules listed here. The refactoring rules set has some operations, but new operations can emerge considering that graph database management systems are still being improved and new rules can become interesting for the new versions that will be released soon.

## 5.1 Future Work

The work presented in this document could be extended with the development of a refactoring tool applied to existent engines which would allow a user to recognize the existent metadata in a database, make changes in the metadata and refactor the data linked to it.

Another activity would be to grow the catalog, adding new refactoring rules and improving the description of the refactoring's execution process, already listed in this document, based on user experiences. For the creation of the new rules, the researchers can make use of the specification guideline developed in this work. They also have to ensure that the new refactoring rules do not subvert the integrity and consistency of the current refactoring rules.

# REFERENCES

AKRAWI, A. **Social Network System Design**. Dissertation. Royal Institute of Technology. Stockholm, Sweden. 2010.

AMBLER, S. W. & SADALAGE, J. **Refactoring databases: Evolutionary database design**. Addison-Wesley Professional. 1st Edition. 2007.

ANGLES, R. **A Comparison of Current Graph Database Models**. In 28th IEEE International Conference on Data Engineering Workshop (ICDEW*)*, pages 171-177. 2012.

BRUHN, D. **Comparison of Distribution Technologies in Different NoSQL Database Systems**. Institute of Technology (KIT). Karlsruhe, Germany. 2011.

CHODOROW, K. **MongoDB: The Definitive Guide**. O'Reilly Media, Inc. 2nd Edition. 2013.

DATE, C. J. **An Introduction to database systems**. Pearson/Addison-Wesley. 8th Edition. 2004.

ELMASRI, R.; NAVATHE, S.B. **Fundamentals of Database Systems**. Addison-Wesley Publishing Company. 6th edition. 2010.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional. 1st Edition. 1999.

FONSECA, A. M.; CAMOLESI Jr.; L. Refactoring Rules for Graph Databases. In World Conference on Information Systems and Technology (WorldCIST'15). /* no prelo*/

GANTZ, J.; REINSEL, D. **The digital Universe in 2020: Big Data, bigger digital shadows, and biggest growth in the far east**. IDC iView. 2012. Available at: http://idcdocserv.com/1414. Last Access: 10 may 2014.

HAN, J.; HAIHONG, E.; LE G.; DU J. **Survey on NoSQL database**. In 6th IEEE International Conference on Pervasive Computing and Applications (ICPCA), pages 363-366. 2011.

LAKSHMAN, A.; MALIK, P. **Cassandra - A Decentralized Structured Storage System**. In ACM SIGOPS Operating Systems Review, pages 35-40. 2010.

LÓSCIO, B. F.; OLIVEIRA, H. R. **NoSQL no desenvolvimento de aplicações Web colaborativas**. In VIII Simpósio Brasileiro de Sistemas Colaborativos. 2011.

MONGODB. Available at: http://www.mongodb.org/display/DOCS/Home. Last Access: 10 may 2014.

NEO4J. **Neo4j: The World's Leading Graph Database**. Available at: http://neo4j.org/. Last Access: 15 dec 2014.

OREND, K. **Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer**. Dissertation. Technische Universität München, Germany. 2010.

PLUGGE, E.; MEMBREY, P.; HAWKINS. T. **The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing**. Apress. 2010.

ROBINSON, I.; WEBBER, J. AND EIFREM, E. **Graph Databases**. O'Reilly Media. 2013.

RUFLIN, N.; BURKHART, H.; RIZZOTTI, S. **Social-data storage-systems**. In Databases and Social Networks. ACM, pages 7-12. 2011.

SADALAGE, P. J.; FOWLER, M. **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. Addison-Wesley Professional. 2012.

STRAUCH, C. **NoSQL Databases**. Lecture, Selected Topics on Software Technology. Stuggart Media University. 2011.

TIWARI, S. **Professional NoSQL**. John Wiley & Son, Inc. 2011.

WARDEN, P. **Big Data Glossary**. O'Reilly Media, Inc. 2011.

WEBER, S. **NoSQL Databases**. University of Applied Sciences HTW Chur, Switzerland. 2010.