Imperial College London
Department of Computing

GraphAware:
Towards Online Analytical Processing in Graph Databases

by

Michal Bachman

Submitted in partial fulfilment of the requirements for the
MSc Degree in Computing (Distributed Systems) of Imperial College London
September 2013

Abstract

Graph databases are becoming increasingly popular as an alternative to relational databases for managing complex, densely-connected, semi-structured data. Whilst primarily optimised for online transactional processing, graph databases would greatly benefit from online analytical processing capabilities. Since relational databases were introduced over four decades ago, they have acquired online analytical processing facilities; this is not the case with graph databases, which have only drawn mainstream attention in the past few years.

In this project, we study the problem of online analytical processing in graph databases that use the property graph data model, which is a graph with properties attached to both vertices and edges. We use vertex degree analysis as a simple example problem, create a formal definition of vertex degree in a property graph, and develop a theoretical vertex degree cache with constant space and read time complexity, enabled by a cache compaction operation and a property change frequency heuristic.

We then apply the theory to Neo4j, an open-source property graph database, by developing a Relationship Count Module, which implements the theoretical vertex degree caching. We also design and implement a framework, called GraphAware, which provides supporting functionality for the module and serves as a platform for additional development, particularly of modules that store and maintain graph metadata.

Finally, we show that for certain use cases, for example those in which vertices have relatively high degrees and edges are created in separate transactions, vertex degree analysis can be performed several orders of magnitude faster, whilst sacrificing less than 20% of the write throughput, when using GraphAware Framework with the Relationship Count Module.

By demonstrating the extent of possible performance improvements, exposing the true complexity of a seemingly simple problem, and providing a starting point for future analysis and module development, we take an important step towards online analytical processing in graph databases.

Keywords: Graph Databases, Neo4j, Analytics, OLAP, Property Graph, Vertex Degree, GraphAware.

Acknowledgements

First and foremost, I would like to thank Dr. Yike Guo for his willingness to be my supervisor and for his time and valuable guidance, especially at the outset of this project.

A huge thank you also goes to Dr. William Knottenbelt for his enthusiasm, his willingness to go the extra mile for his students, and for his invaluable help in the final stages of this project.

It is my great pleasure to acknowledge Dr. Jim Webber, who planted the idea of graph database analytics into my head in the first place and supported me throughout the project, with a contagious passion for graphs. I hope you enjoy your holiday reading!

I am indebted to my friend, Adam "Tarra" George, for being the grammar and spelling police, taking his time to carefully read through the report a number of times, and making the effort to fully understand it. Ta, m'duck!

I would like to thank my parents for teaching me that education is one's most valuable asset and supporting me in this endeavour, both emotionally and financially. Now, my education is finally complete to my father's standards!

Most importantly, I would like to thank my fiancée, Daniela, who is still proofreading the report as I am writing these lines, for her love, patience, and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Graph databases and the graph data model are becoming increasingly popular as a complement of, and sometimes an alternative to, traditional relational databases and the relational data model, giving rise to a growing desire for analysis of data modelled as graphs.

In contrast to relational databases and the relational data model, different graph databases employ different kinds of graph-based data models. This project focuses on the data model used by Neo4j, a popular open-source online transactional processing (OLTP) graph database. It is a directed graph with labelled edges, where both vertices and edges have an arbitrary number of arbitrary key-value pairs called attributes or properties. This model is referred to as the property graph model.

Numerous software systems and frameworks exist for large-scale graph processing. These solutions typically take a static graph, in one form or another, and perform an offline, batch-like analytical operation on the graph outside of the database, often utilising map-reduce frameworks.

Most relational databases posses online analytical processing (OLAP) features, which have also been extensively studied in academia. In-database online analytical processing for graph databases, however, has not received much academic attention, which is especially true for OLTP property graph databases. Perhaps as a consequence, today's graph databases typically provide no OLAP capabilities.

Graph databases are optimised for local graph traversals, yet many useful graph algorithms perform global graph operations. An analytical query that runs in time linearly proportional to the number of vertices and/or edges might take hours in very large graphs. Such a query could hardly be called online or real-time.

One does not even need to go as far as performing global graph operations to find poorly performing analytical queries. The degree of a vertex, for example, is a simple, yet powerful and frequently needed graph measure. In a property graph with labelled directed edges, the degree is not just a simple edge count, since one must take into account edge labels and attributes. Inspecting every edge when querying for a vertex degree, an algorithm with time complexity linearly proportional to the total number of edges at the vertex, can be sufficiently slow to be disqualified from being considered real-time.

1.2 Aims

The ultimate objective of this project is to advance the field of computing towards OLAP on primarily OLTP graph databases with the property graph data model. Whilst one can imagine a virtually unlimited number of different types of analytical queries on a graph database, the aim is an in-depth investigation of efficient querying for vertex degrees in property graphs, attempting to generalise the outcomes into a form useful for further research, perhaps of other kinds of analytical queries. Neo4j is used for validating and evaluating the outcomes of the research.

1.3 Contributions

The contribution of this project is twofold. First, a framework for Neo4j has been created, enabling convenient development of *modules* that store and maintain pre-computed information about the graph. Second, the problem of finding vertex degrees in a property graph has been studied in depth and an efficient solution for vertex degree analysis has been found and implemented as a framework module. The remainder of this section describes both contributions in more detail.

1.3.1 GraphAware Framework

A framework for Neo4j, called *GraphAware*, has been designed, fully implemented, tested and evaluated. Adding no significant overhead, the framework enables convenient development of *modules* that store pre-computed information about the graph, typically in the graph itself, and keep that information up to date. The framework has a number of utilities that support such development and boasts three key features:

Module Lifecycle Management First of all, modules that wish to be made aware of about-to-be-committed graph mutations register themselves with the framework, which in turn fully manages their lifecycle: startup, (re-)initialisation, configuration and change detection thereof, runtime work delegation, exception handling, and shutdown. At runtime, the framework informs the modules about graph mutations right before they are committed to the database, allowing modules to react to these changes by, for instance, performing additional mutations.

Graph Snapshots Secondly, a *graph snapshot* mechanism atop Neo4j transaction event API has been introduced, allowing for easy analysis of graph mutations. Modules can inquire about created, deleted, and changed vertices and edges, as well as traverse two versions of the graph: the “old" graph, i.e., a snapshot of the graph as it was before the transaction started, and the “new" graph, i.e., the future version of the graph as it will be after the transaction commits.

Decorators and Representations Finally, a set of base classes has been created, simplifying the development of custom vertex and edge representations and decorators. The utility of a string representation of an edge (including its attributes), for instance, has been demonstrated when storing some pre-computed information about such an edge. Decorators, on the other hand, have been used to increase performance by providing additional caching.

1.3.2 Efficient Vertex Degree Analysis

The problem of finding a vertex degree in a property graph has been studied in depth. Apart from contributions stemming from that study, such as a formal definition of vertex degree in a property graph, an efficient solution for caching vertex degrees on vertices themselves with constant space and read time complexity has been found, resulting in the following additional contributions.

Compaction Algorithm It is possible and, in fact, not uncommon that some attributes take on a different value for each edge. An example of that could be a timestamp attribute with a millisecond-precision value indicating, when the edge was created. Caching the number of edges with such attributes would require space linearly proportional to the number of edges. A *compaction algorithm* and a *Property Change Frequency* heuristic have been devised, allowing the space requirement to be constant, whilst keeping the resulting information loss as small as possible.

General-to-Specific Ordering In part, this has been enabled by the introduction of partial general-to-specific ordering of edges with attributes and the development of a self-generalising edge represen-

tation.

Vertex Degree Module Finally, a vertex degree analysing module for the GraphAware Framework, called the *Relationship Count Module*, has been designed, implemented, tested, and evaluated, putting the ideas introduced above into practice. It has been demonstrated to increase the speed of vertex degree analysis in some scenarios by several orders of magnitude, for as little penalty as 20% of edge write throughput. This, per se, is already a useful outcome, as it can be readily used when efficient vertex degree analysis is required in Neo4j. However, as an additional contribution, these results give an indication of the possible performance improvements and penalties for future module developers. Last but not least, the module serves as a reference implementation of a GraphAware module.

1.4 Report Structure

The rest of the report is organised as follows. Chapter 2 introduces the background for the research and states the problem. Chapter 3 is a brief survey of related work. Chapter 4 analyses the problem introduces a number of own definitions, theorems, proofs, and theoretical solutions. Chapter 5 describes the design and implementation of the software that puts the theory into practice, while Chapter 6 presents evaluation of the work. Chapter 7 comments on the evaluation and suggests topics for further research in this area.

Appendix A serves as a reference of the symbols used throughout the report and especially in Chapter 4. Appendices B and C provide a user guide for the GraphAware Framework and the Relationship Count Module, respectively.

Chapter 2

Background

2.1 Graph Databases

In the last decade, the nature of data stored and processed by computer systems has changed in a number of ways. First of all, the sheer volume of data mankind produces, stores, and processes is growing very quickly. Secondly, data is becoming more complex and less structured. Finally, data is becoming increasingly inter-connected.

At the same time, the architecture of database-backed software systems has evolved from database-centric, where the database serves as an integration point between system components, towards service-oriented, where system components integrate on API level, allowing for different kinds of data stores in a single system.

Both of the trends outlined above gave rise to a new category of database management systems (DBMS) called NoSQL. Sadalage and Fowler [22] provide an excellent overview of these and a discussion of the definition, concluding that NoSQL refers to "an ill-defined set of mostly open-source databases, mostly developed in the early 21st century, and mostly not using SQL".

Sadalage and Fowler [22] further identify four broad categories of NoSQL databases: key-value, document, column-family, and graph. The first three differ from relational database management systems (RDBMSs) by using "aggregate data models", i.e., operating on aggregates - units of data with more complex structures than relations and tuples. Graph databases, on the other hand, have a substantially different data model; they work with simple records and complex interconnections.

RDBMSs use the relational data model, first proposed by Codd [8], with relations, attributes, and tuples. These are explained in detail in the vast amount of literature published on database systems, e.g. Connolly and Begg [9] or Ullman et al. [24]. On the other hand, *graph data model*, in its simplest

from, operates with the notion of vertices and edges, familiar from graph theory. Graph theory is an extensively studied field of discrete mathematics, dating back to at least mid-18th century [11].

Semantically, the graph data model is richer than the relational data model. Whilst relationships between entities have to be inferred in the relational model, they are modelled explicitly in a graph. This has significant implications, both in that graph data is naturally multi-dimensional and that the structure of the graph, in addition to its "contents" provides insightful information that can be analysed.

There are two ways graph data structures are typically represented in computer systems: as a collection of adjacency lists, or as an adjacency matrix [10]. Systems assuming sparse graphs, i.e., graphs where the number of edges is much smaller than the square number of vertices, prefer the former, because it is more compact. Graphs are said to be stored and/or processed *natively*, if one of the aforementioned representations, or an extension thereof, is used.

Graph database management systems (GDBMSs), or *graph databases* in short, allow real-time storage, retrieval and modification of complex, semi-structured, highly-connected data, modelled as a graph [21]. This ability is also referred to as online transactional processing (OLTP). Some graph databases, such as Neo4j, store data natively as a graph, while others transform the data from/to another model (e.g. relational). Either way, the *exposed* model, i.e., the data structures users interact with, is what all graph databases have in common.

In contrast to OLTP graph databases, *graph compute engines* are technologies designed to perform computations on large data sets, represented as graphs in one way or another, and may or may not include a data storage capability. These technologies are primarily concerned with batch, potentially parallel and/or distributed data analysis [21]. Graph compute engines are explicitly and completely out of scope of this project.

2.2 Basic Definitions

Let us review some basic definitions from graph theory that will be used throughout the report. All definitions in this section are taken or adapted from Diestel [11], unless indicated otherwise.

Graph A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq (V \times V)$. The elements of V are *vertices* or *nodes* of the graph G and the elements of E are its *edges* or *relationships*. Because literature on graph theory typically talks about vertices and edges, while Neo4j and its APIs operate on nodes and relationships respectively, these terms are used interchangeably throughout the report.

Order The *order* of a graph G, denoted $|G|$, is the number of vertices of G. Its number of edges is denoted $||G||$.

Incident Vertex A vertex v is *incident* with an edge e if $v \in e$; then e is an edge *at* v. An edge $\{x, y\}$ can be written as xy.

Vertex Degree The set of all edges in E at a vertex v is denoted $E(v)$. The degree of a vertex v, denoted as $d(v)$, is the number $|E(v)|$ of edges at v.

Subgraph Given two graphs $G = (V, E)$ and $G' = (V', E')$, G' is a *subgraph* of G, written as $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. Furthermore, if $G' \subseteq G$ and G' contains all the edges $xy \in E$ with $x, y \in V'$, then G' is an *induced subgraph* of G.

Path A *path* is a non-empty graph $P = (V, E)$ of the form
$V = \{x_0, x_1, \ldots, x_k\}, E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$. The number of edges in a path is its *length*.

Directed Graph A *directed graph* $G = (V, E, init, ter)$ is a graph $G = (V, E)$ together with two maps $init : E \to V$ and $ter : E \to V$ assigning every edge $e \in E$ an *initial vertex* $init(e)$, also called the *tail*, and a *terminal vertex* $ter(e)$, also called the *head*. The edge e is said to be *directed from* $init(e)$ *to* $ter(e)$. There may be several edges between the same two vertices. If $init(e) = ter(e)$, the edge e is called a *loop*.

Vertex Degree Revisited Rather than about a vertex degree, we talk about *indegree* and *outdegree* in a directed graph. Indegree, denoted $d_{in}(v)$, is defined as the number $|E_{in}(v)|$ of edges at v, where $E_{in}(v) \subseteq E(v)$ such that $\forall e_{in} \in E_{in}(v), ter(e_{in}) = v$. Similarly, outdegree, denoted $d_{out}(v)$, is defined as the number $|E_{out}(v)|$ of edges at v, where $E_{out}(v) \subseteq E(v)$ such that $\forall e_{out} \in E_{out}(v), init(e_{out}) = v$. Note that a loop adds one to both the outdegree and indegree of its vertex. Thus, in a graph with a single vertex and a single edge, which is a loop at that vertex, the indegree of the vertex is one and so is its outdegree[1].

[1] the definitions in the last paragraph are our own

2.3 Property Graph

Property Graph Data Model A *graph*, i.e., pair $G = (V, E)$, s.t. $E \subseteq (V \times V)$ as defined by Diestel [11], is not a very powerful data model, because it does not provide any facilities for actual data storage. Hence, graph databases typically employ extensions of this model, one of which is referred to as the *property graph model*, which we formally define in Chapter 4. Informally, property graph model, as used by Neo4j, is a set of vertices and directed, labelled edges, where both vertices and edges can have an arbitrary number of arbitrary attributes. Attributes have a key, uniquely identifying the attribute, and a value. Vertices and edges in a property graph are commonly referred to as *property containers*. It is the combination of vertices, directed, labelled edges, and attributes that is used for modelling a real-world domain.

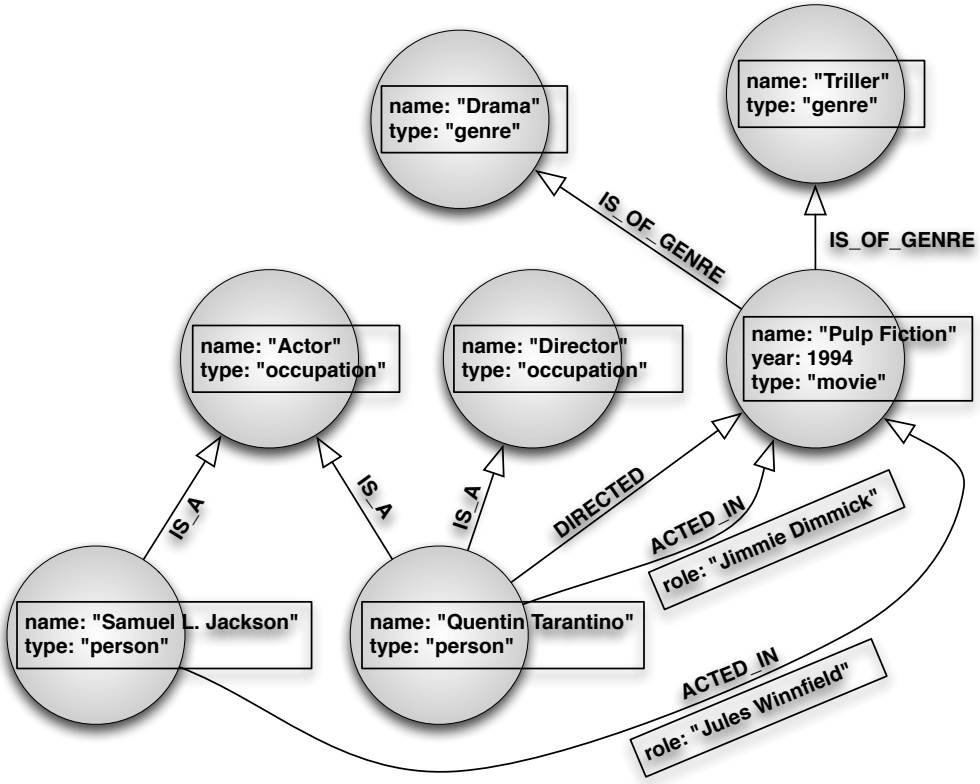

Figure 2.1: Example Domain Modelled as a Property Graph

Example Property Graph Figure 2.1 shows a small subgraph of a hypothetical movie database modelled as a property graph. Vertices, depicted as circles, represent real-world entities, in this case movies, genres, actors, directors, etc. Directed edges, depicted as arrows labelled with capitalised character strings, represent relationships between the entities. Labels give edges a semantic meaning in the model. For example, the edge labelled `ACTED_IN` means that the initial vertex of the edge represents an actor that acted in a movie, which is represented by the terminal vertex of the edge. Finally, rectangular boxes represent attributes attached to vertices and edges, enriching the model with additional informa-

tion. Depending on the application, this information might be absolutely necessary to uniquely link the vertex/edge to its real-world meaning (e.g. movie or actor name), or nice-to-have extra information, which could otherwise be looked up elsewhere (e.g. movie year or actor role). The attributes consist of an attribute name (key) and its value, separated by a colon in the figure.

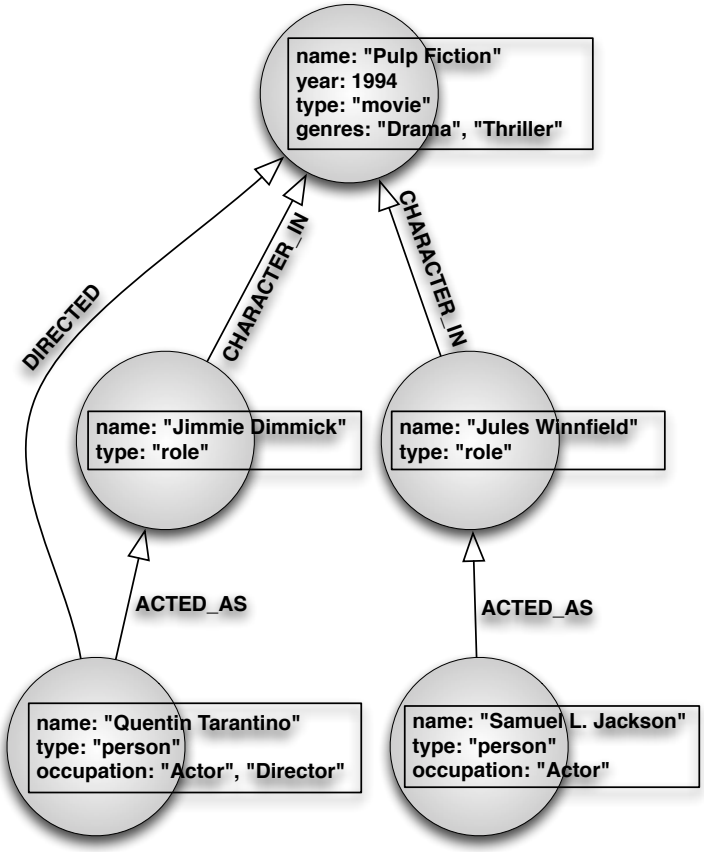

Figure 2.2: Alternative Movie Database Domain Model

Modelling Options As in the relational data model, there is no single correct way of modelling a certain domain in a property graph. To illustrate the point, consider the graph presented in Figure 2.2. It captures exactly the same information as the graph presented above in Figure 2.1, with different choices on how to model certain aspects of the domain. For instance, movie genres are modelled as attributes on movie vertices, whilst names of movie characters are vertices instead of edge attributes. When modelling data for a graph database, the choices made depend on a number of different factors, comprehensive survey of which is beyond the scope of this discussion. One, perhaps the most important consideration, however, is what kinds of queries will be executed against the data.

Querying a Property Graph The most natural property graph query consists of finding a starting vertex and exploring the graph by traversing selected edges and the vertices they connect, discovering interesting information along the way until a terminating condition is met. The actual algorithm used

for the traversal, for example breadth-first or depth-first search, depends on the goal of the query and can be decided by the developer or, perhaps, a built-in query planner. When using a database that stores data natively as graphs, the performance of such a local query is independent of the size of the remaining (unexplored) part of the graph. In contrast, all but the most trivial graph queries would require the use of a join operator in a relational database, the performance of which degrades with increasing data size (shown, for example, by Ordonez and García-García [18]).

Example Query Let us present an example of a typical graph query on the movie database from Figure 2.1. The purpose of the query is to find genres of all the movies an actor has acted in. Such a query would be created as a traversal of all outgoing edges with labels `ACTED_IN` or `IS_OF_GENRE`, starting from the actor vertex. All vertices reached by the traversal, such that the traversed path length from the starting vertex to the reached vertex is equal to 2, would correspond to individual elements in the result set of the query. Figure 2.3 illustrates such a query for Samuel L. Jackson. The path traversed is shown by a dashed red line and the vertices forming the result set by a solid red line. Note that the total number of movies in the database, which Samuel L. Jackson has not acted in, should not have any performance implications, as these vertices would not be visited by the traversal.

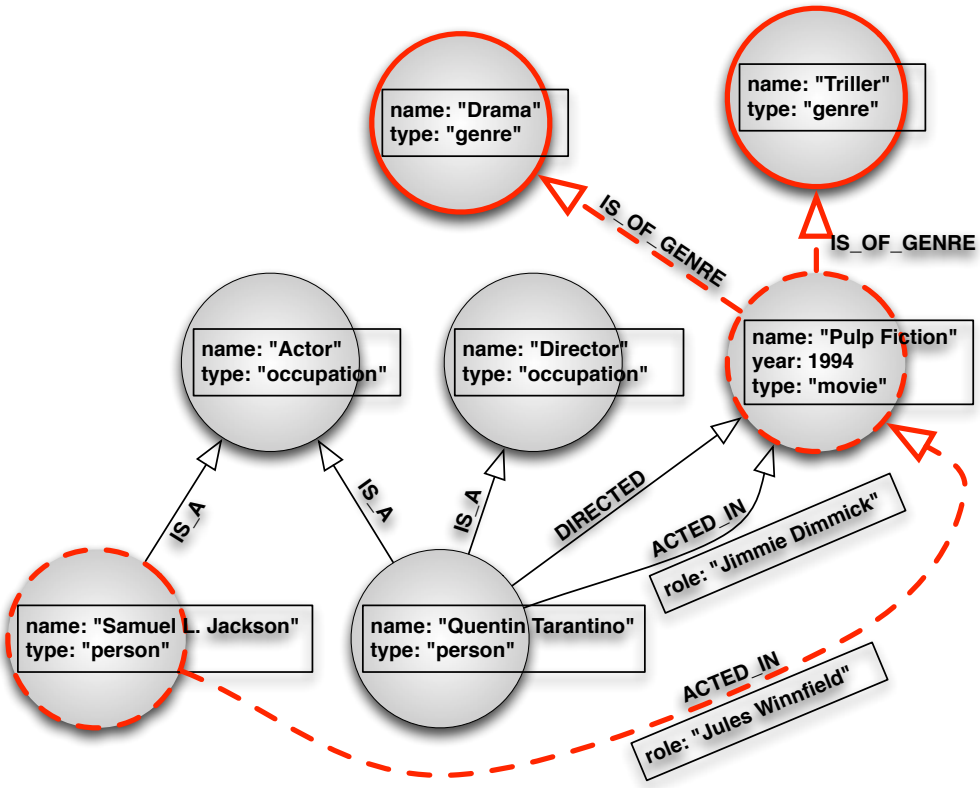

Figure 2.3: Example Graph Database Query

2.4 Neo4j

Neo4j is an open-source graph database running on the Java Virtual Machine (JVM) that processes and stores data natively as property graphs. Like many NoSQL databases, it is schema-free. This means that, unlike in relational databases, data does not have to adhere to any pre-defined structure. Nodes, relationships, and properties can be created completely arbitrarily, with the exception that relationships must have a start node and an end node at all times, which is enforced by the database.

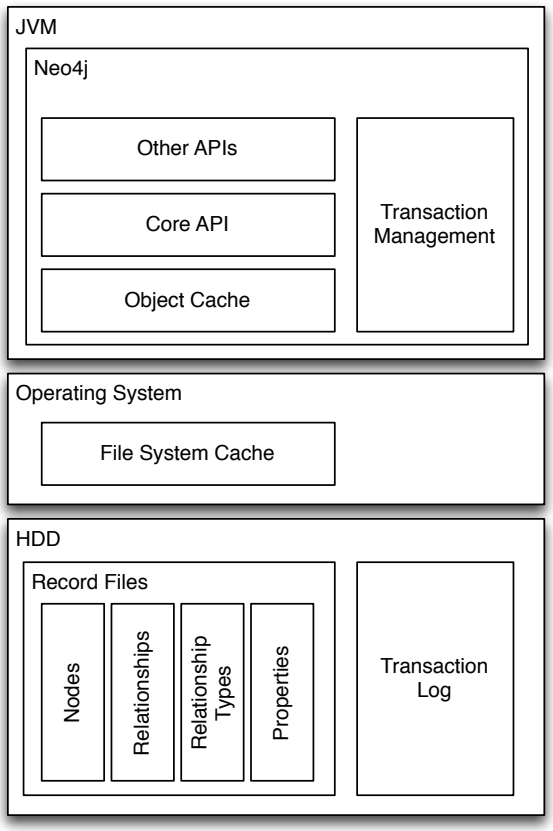

Figure 2.4: Neo4j High-Level Architecture

Architecture Figure 2.4 shows the high-level architecture of Neo4j. On top of the core Java API, a number of other APIs for graph data storage and manipulation are provided, including a REST API and other JVM language bindings. Through the object cache, which holds materialised portions of the dataset as Java objects, and the file system cache, which caches parts of the filesystem to speed-up I/O operations, data is persisted to and read from hard disk(s). As an orthogonal concern, transaction management, discussed shortly, is provided. Data is physically stored in a number of separate record files, four of which are also depicted in Figure 2.4. Three of these files are shown in detail in Figure 2.5[2].

[2] (both figures inspired by Robinson et al. [21] and modified for purposes of this report, additional details obtained by inspecting the Neo4j codebase)

Node Store Nodes are stored as fixed-size 9-byte records, allowing speedy lookup by ID, by calculating the offset in the node store. Each node record consists of a single bit flag (irrelevant for this report), a 35-bit pointer to the relationship store (ID of the first relationship of the node), and a 36-bit pointer to the property store (ID of the first property of the node).

Relationship Store Similarly, relationships are stored as fixed-size 33-byte records with pointers to the start and end nodes in the node store, a relationship type pointer, next and previous relationships for both nodes, and the first property of the relationship.

Property Store Properties in Neo4j are key-value pairs, where keys are Java strings and values can be of any Java primitive type, an array of primitives, a string, or an array of strings. Properties for both nodes and relationships are 41-byte records in the property store: 36-bit pointers to the next and previous property records of the property container, and up to four 8-byte property blocks. Figure 2.5 also shows the details of the second property block. 4 bits, denoted “t", are used as a pointer to the relationship type store and identify the type of the relationship. 3 bytes, denoted “k", are used to look up the property key in a store that links key IDs to their string representations (not depicted). The value of the property is stored either in the same block, denoted “v", if it is up to 4 bytes in size, in the remaining property blocks, if it is up to 24 bytes in size, or in a separate store with dynamic size (not depicted), if larger. In the last case, the 4 bytes denoted “v" serve as a pointer to the dynamic store and the remaining property blocks can be used for other properties.

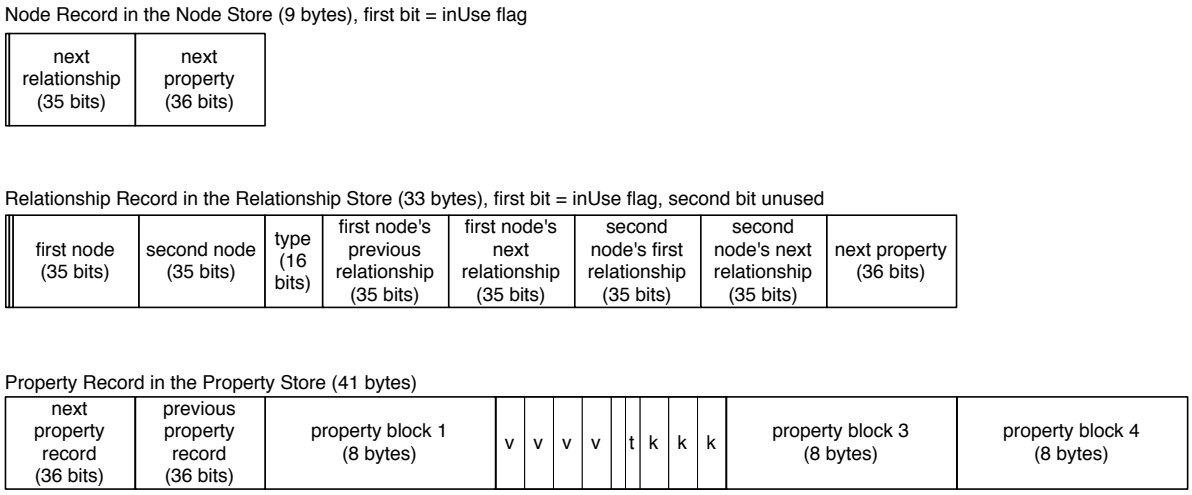

Figure 2.5: Neo4j Data Storage

Index-Free Adjacency While this graph representation is not exactly the same as a traditional adjacency list, it allows for rapid graph traversals by merely following pointers, instead of having to look them up in a global index. Robinson et al. [21] refer to this as *index-free adjacency*.

Transactions In order for a database to be reliable and remain in a consistent state in the presence of concurrency and software and hardware failures, database management systems should provide support for transactions [9]. Neo4j not only provides this support, but, unlike many other NoSQL databases, requires that every mutating operation be run in the context of a transaction. Thus, logical units of work are guaranteed to be atomic, consistent, isolated, and durable (ACID). Atomic operations either happen entirely, or not at all. The database remains in a consistent state at all times; for example, when a node is deleted but still has relationships, the database will reject such a mutation and rollback the entire transaction, because it would cause the presence of an orphaned relationship were it to commit. Changes performed by transactions that have not yet been committed are not visible to other, concurrently running transactions. Finally, once a transaction is committed, the changes are persisted permanently to a durable medium (hard disk).

Transaction Event API For this project, one important aspect of Neo4j transaction handling is programmatic access to the transaction lifecycle and the ability to inspect about-to-be-committed and just-committed graph mutations.

2.5 Online Analytical Processing

OLTP The primary purpose of relational and, indeed, graph databases is typically online transactional processing; that is, efficient execution of a large number of relatively simple transactions [9]. These include create, read, update, and delete operations (CRUD).

OLTP in RDBMS In relational databases, all CRUD operations are typically specified using declarative Structured Query Language (SQL), an implementation of relational algebra. Read queries are implemented as searches in relations (tables) and result in "temporary" relations with projected attributes and selected tuples. When multiple relations serve as input to a read query, a set-based operation called *join* is typically involved. The presence of this operation is one of the distinguishing characteristics of relational databases and it is often a performance bottleneck.

OLTP in GDBMS There is no single standard query language for graph databases. CRUD operations are typically performed using a query language specific to the database (or multiple databases), or using a RESTful or programming-language-specific API. In native graph databases, read queries are implemented as traversals through the underlying property graph; they are local to a starting vertex or a set of starting vertices. Read queries in graph databases typically result in either a set of vertices, a

set of edges, or a set of paths. That is to say, an OLTP read query in a graph database does not typically result in an induced subgraph.

OLAP In contrast to OLTP, online analytical processing (OLAP) is the "dynamic synthesis, analysis, and consolidation of large volumes of multi-dimensional data"[9]. It is hard to define a clear boundary between what is still an OLTP operation and what is already OLAP, but OLAP typically involves multi-dimensional view of aggregate data for the purpose of analysis. The distinction is even less obvious when considering different data models. What might be a straightforward shallow breadth-first search in a GDBMS (hence considered OLTP) could be a very expensive multi-join operation in RDBMS (thus qualifying as OLAP). On the other hand, a simple sum on an attribute of all tuples in a single relation might still be considered OLTP in RDBMS, but would be treated as OLAP in GDBMS, because it involves a graph global query, for which the data model is not optimal.

OLAP in RDBMS Although primarily designed for OLTP, many RDBMSs have acquired OLAP capabilities over the years. This is exemplified through the addition of OLAP-specific operators, such as CUBE and ROLLUP, to the ANSI SQL standard [9]. Various benchmarks and metrics have been proposed, an example of which is the APB-1 from the OLAP Council [9]. Multidimensional operations in RDBMS OLAP typically involve *roll-up*, *drill-down*, *slicing and dicing*, and *pivoting*. For business reasons, there is a clear trend towards real-time or near-real-time analytics [9].

OLAP in GDBMS Although graphs have been studied for substantially longer period of time than RDBMSs have been around, today's graph databases lack native OLAP support. In fact, it is a frequently cited concern about graph database adoption[3].

Structural Analytics Because of the explicit relationship modelling in graph data models, there is an additional category of analytical processing, which we refer to as *structural analytics*. As opposed to OLAP, which looks at aggregated data held by vertices and perhaps also edges, this kind of analytical processing deals with qualitative and quantitative aspects of the actual structure of the graph/network.

2.6 Problem Definition

An analytical query, which requires inspecting many nodes and relationships, is not something native graph databases are optimised for. Therefore, the time such queries take can be prohibitively long.

[3]source: on-line Q/A forums, author's own consulting experience, conversations with industry professionals

While local traversals can be executed in microseconds, finding the sum of a single property across a million nodes in the graph can take between 400 milliseconds (entire graph in object cache) to 10 seconds (nothing in caches). Contrast that with MySQL, a relational database, in which the sum of a single column in a table with one million rows takes about 250 milliseconds when data is read from disk, i.e., 40 times less[4].

Before we start considering analytical and graph-global queries, let us look at a simple problem, which is in a way similar to the one described above: vertex degrees. Due to the layout of graph data on disk, shown in Figure 2.5, a query asking to count all relationships of a given type and, perhaps, with a specific property value, requires inspecting every single relationship. On a node with one million relationships, this query exhibits similar performance characteristics to the one described in the previous paragraph.

Yet counting the number of relationships for a node, optionally constrained by type, direction, and properties, is a very useful and often needed operation, both in its own right and as a primitive building block for other, more complex and even analytical queries. Consider, for example, a simple system that allows users to rate movies, using a `RATED` relationship and a property called `rating`, ranging from 1 (hated it) to 5 (loved it). To see how popular a movie is, some or all of its `RATED` relationships have to be tallied. This involves, as mentioned already, traversing all of the movie's relationships.

In graph analytics, and especially social network analysis, a number of *centrality* measures has been studied, where centrality is the relative importance of a vertex in a graph. The simplest one of these measures is called *degree centrality* and is defined as the degree of a vertex [13], providing another example why it might be useful.

Disregarding the option of computing approximate results for a moment, the only way to prevent accessing large amounts of global graph data is pre-computing and/or localising the information. Since the graph is being constantly mutated in an OLTP environment, this pre-computed and/or localised data must be kept in sync with the actual data in the graph.

Another option would be employing probabilistic sampling, in order to arrive to an approximate result. However, traversing random relationships from a vertex requires knowing, at the very least, how many there are in total. As already explained, this requires inspecting every single one of them, which is equivalent to finding the vertex degree.

Based on the discussion above, this project sets out to explore, whether it is possible to pre-compute and localise interesting information in a constantly changing property graph in an OLTP graph database, to prototype a solution, and to evaluate the performance thereof. It aims at using vertex degrees as an

[4] (measurements performed on a quad-core MacBook Air with 8 GB RAM and an SSD disk. These are for illustration only, not comprehensive benchmarks)

example problem for its relative simplicity and wide applicability, but intends to generalise the results, so that they are useful for other, more complicated, use cases, such as OLAP and structural analytics in graph databases.

Chapter 3

Related Work

This chapter provides a brief chronological overview of the literature published on OLAP in graph databases, which is relatively scarce, and on structural graph analytics.

3.1 OLAP in Graph Databases

3.1.1 SNAP and k-SNAP

Tian et al. [23] recognise the fact that most existing graph summarisation methods use simple statistics, results of which can be difficult to interpret. They also emphasise the fact that many algorithms are based purely on vertex connectivities, largely ignoring their attributes, which is also true for edges. They talk about the need of OLAP-style operations, such as roll-up and drill-down, on user defined attributes and edges.

The authors introduce and formally define two operations for graph summarisation, *SNAP* (Summarisation by Grouping Nodes on Attributes and Pairwise Relationships) and *k-SNAP*, both of which produce a graph as a result. SNAP produces a summary graph through a homogeneous grouping of the input graph's nodes, based on user-selected attributes and relationships.

The SNAP operation groups nodes into completely homogeneous disjoint groups with respect to the selected attributes and infers relationships between these groups, such that if there is an inferred relationship between two groups (in the resulting graph), each node in one group must have a relationship of that type with at least one node in the second group (in the original graph). While such a relationship inference is definitely useful for some use cases, it seems very restrictive. The authors do recognise this fact in the sense that noisy or incomplete data would result in a large number of groups.

An alternative approach, k-SNAP, allows users to specify the number of resulting groups, relaxing the homogeneity requirement for relationships by not requiring that every node participates in a group relationship. k-SNAP is proven to be NP-complete in the same paper. An efficient algorithm is proposed for the SNAP operation, as well as two heuristic methods for approximating k-SNAP results.

While SNAP and k-SNAP are interesting graph summarisation techniques, their applicability for OLAP-style graph analytics is questionable. Their primary purpose is creating groups of nodes, based on the (non)existence of relationships between those groups, using node attributes to constrain the grouping. Roll-up and drill-down is achieved by controlling the number of groups created. In traditional OLAP, however, roll-up and drill-down is performed along data dimensions to achieve a coarser-grained and finer-grained view, respectively. For example, one might want to see sales figures for the whole company, drill-down into summaries for branches, departments, and so on. SNAP and k-SNAP do not operate with this notion of data dimensions at all. Moreover, not much can be said about the groups produced by these operations, apart from the (non)existence of specified relationship types amongst them. No quantitative information is available about those relationships (how many there actually are, how they are distributed across the nodes etc.).

3.1.2 Graph OLAP Framework

Chen et al. [7] recognise the need for OLAP-like operations on graph data sources and the fact that existing OLAP technologies are not sufficient, since they do not consider explicit relationships between data tuples.

The authors develop a theoretical graph OLAP framework, which presents a multi-dimensional and multi-level view over graphs, where hierarchies can be associated with dimensions. One important characteristic of their approach is that their starting point is not a single graph representing a domain; instead, they use a set of graphs representing collaboration patterns among authors working in a given field, where each graph in the set is a snapshot of a collaboration network for a given year and conference. In other words, their graphs have very limited dimensionality; the dimensions (such as conference name and year) are represented as labels of entire graph snapshots. This model is significantly different from a typical graph database model, where all dimensions would be captured within a single graph. For instance authors, papers, conferences, and years could be modelled as nodes and `WRITTEN_BY` (author), `WRITTEN_IN` (year), and `WRITTEN_FOR` (conference) as relationships.

Aggregation is achieved by overlaying the graph snapshots for particular dimension(s). As in the previous paper, the result of this aggregation is a graph. They apply aggregation functions, such as `COUNT`, to relationships, but not to nodes.

As a further motivation, the authors present an example of finding a maximum flow between two cities, connected by different means of transport with different transportation capacities (represented as weights on relationships). While they present this problem as a case of OLAP, it would fall under structural analytics in our categorisation. Assuming the lack of any further constraints, such as maximum hop-count, this is an NP-hard problem and involves all the vertices and edges in the graph. Another example is later presented, where the authors perform a roll-up-like operation on the author collaboration network and compute an aggregated graph that represents author collaborations grouped by institutions, again applying the `COUNT` operator on relationships.

As mentioned above, dimensions are represented by properties/labels of different graph snapshots, which could be thought of as the actual graphs being just cells in a relational table. Perhaps due to this modelling decision, the authors divide graph OLAP into informational OLAP (I-OLAP), i.e., aggregation of graph snapshots across labels, and topological OLAP (T-OLAP), i.e., aggregation within the individual graphs. The rest of their paper focuses on I-OLAP, while T-OLAP is examined by Qu et al. [19].

I-OLAP The authors go on to define an I-Aggregated graph as a graph computed from a set of network snapshots, whose informational dimensions are of identical values, by applying an aggregation function to node and relationship attributes. They require each snapshot to contain the same set of nodes, which is a major limitation of this framework, because it will rarely be the case in any real-world application. They also define a roll-up, drill-down, and slice/dice operation on the aggregated graph. As in traditional RDBMS OLAP, they classify the aggregate measures into *distributive*, *algebraic*, and *holistic*, depending on whether they need all the underlying data points (holistic), or whether an aggregated measure (distributive) or a set thereof (algebraic) for a subset of the data is sufficient.

Following this division, the authors devise optimisations for efficient computation of aggregated measures based on pre-computed materialised views (for distributive and algebraic measures). They also prove that for some measures, such as centrality, the computation is hard.

T-OLAP In a subsequent paper, Qu et al. [19] show efficient computation of T-OLAP based on two characteristics of computed measures, *distributiveness* and *monotonicity*. Distributiveness allows them to use pre-aggregated networks to compute even higher level summaries (roll-up), while monotonicity helps pruning the search space when executing a query. For example, when looking for authors with a minimum number of published papers, this number cannot be greater than the total number of papers published by authors of the same institution combined. Thus, for instance, if an institution has 10 published papers and we're looking for authors with at least 11, all authors belonging to the institution can safely be ignored.

Similarly to our research, the authors investigate pre-computing information and materialising views over data. However, they assume a static graph, and therefore do not consider the effects of graph mutations that inevitably happen in an OLTP database. Moreover, as already mentioned, dimensions are represented as graph snapshot labels, rather than nodes and relationships, as would be the case in a graph database.

3.1.3 Graph Cube

In the last and latest work on this topic, Zhao et al. [26] also recognise the need for OLAP on graphs and introduce the concept of a *Graph Cube* by combining characteristics of multi-dimensional graphs with existing data cube technologies. The Graph Cube model is essentially a set of all possible aggregations of the underlying multi-dimensional network.

The authors define two types of queries on a graph cube: a *cuboid query*, resulting in a specific aggregation of the multi-dimensional network, and a *crossboid query* (for cross-cuboid), which is a query with more than one cuboid involved.

In terms of implementation, the authors identify three possibilities: full materialisation, no materialisation, and partial materialisation, the first two (quite naturally) taking too much space and time, respectively, in any but the most trivial networks.

Again, the paper does not consider the effects of a changing graph. Moreover, the model used is an undirected graph and has neither labels nor properties on relationships. Finally, informational dimensions are represented as attributes on nodes, while the option of representing them as relationships to nodes is not investigated.

3.2 Structural Analytics

In contrast to OLAP, the topic of structural analytics on (mostly static) graphs has been well studied. Many of the structural measures are very useful in the context of graph databases, too. For example, finding a balanced minimum cut of a graph, which is known to be an NP-complete problem, could help clustering a graph database across multiple machines.

3.2.1 Centrality

Due to the recent rise in the amount and popularity of social networking sites, measures for quantifying a node's importance, also known as centrality, within a network and efficient algorithms for exact or

approximate computation thereof have received increased attention. These, however, are applicable to a wide range of other domains, including, for instance, route planning. The following paragraphs briefly introduce a few of the many publications on this topic, in chronological order.

A good summary and a (re)introduction of 9 different centrality measures (primarily for social networks) is provided by Freeman [13]. Brandes [4] introduces a new algorithm for one type of centrality, called *betweenness centrality*, which runs in $O(nm)$ and $O(nm + n^2 log n)$ time for unweighted and weighted graphs, respectively - an improvement from $O(n^3)$, the fastest known algorithm until then. White and Smyth [25] introduce algorithms for estimating relative importance in a graph, i.e., which nodes are important with respect to a given set of nodes. Newman and Girvan [17] use betweenness measures for community detection in graphs, achieved by iteratively removing edges with high betweenness centrality. All the literature discussed thus far only considers shortest path between nodes for computing centrality measures. Brandes and Fleischer [6] use a current-flow variant of closeness centrality, which takes into account all paths between nodes, and proposes algorithms and approximation schemes for betweenness computations.

All papers in the last paragraph treat betweenness as a global graph property, i.e., most (and often all) vertices have to be accessed to compute the measure. Everett and Borgatti [12] experiment with the relationship between ego network betweenness (i.e., a local property) and betweenness of the node in the entire network. They find that, while there is no formal connection, there is a high correlation in many real-life and randomly generated graphs. Rattigan et al. [20] use *structure indices* of the network to efficiently approximate betweenness and closeness centrality. The results are quite impressive: path length between any two nodes can be estimated in constant time and betweenness centrality estimation accesses less than 5% of the nodes explored by breadth-first search.

With adaptive random sampling, Bader et al. [2] approximate betweenness centrality using a reduced number of single-source shortest path computations for nodes with high centrality and show that it is an improvement over the algorithm presented by Brandes [4]. Geisberger et al. [15] propose an approximation of betweenness centrality computed using only a single (“canonical”) shortest path. Finally, Brandes [5] provides a concise review of different shortest-path betweenness centrality variants and argues that they are all computable with simple variants of the same algorithm, introduced by Brandes [4] himself.

3.2.2 Other Interesting Measures

The literature on centrality measures presented above is just a small sample of the research published on that single topic. Furthermore, there are a number of other interesting measures on networks, such

as similarity between vertices, which have also been studied quite extensively. Many different research projects could be devised for every one of these measures in the context of OLTP graph databases.

3.3 Summary

The first four papers presented above are the only publications on the topic of OLAP in graph databases. There are problems that have not been addressed by any of the papers, which should be researched before a real-world graph database (and specifically Neo4j) acquires OLAP capabilities.

First of those problems is the fact that the database is primarily OLTP. In other words, the OLAP framework must be able to deal with a changing graph. Secondly, there are different ways of modelling the same domain in a graph. Some users, for example, might choose to put a genre of a movie on movie nodes as attributes, while others might create a node for each genre and connect movies to it by an `IS_OF_GENRE` relationship, as illustrated in Figures 2.1 and 2.2. No attention has been paid to this fact whatsoever. Finally, none of the research presented above deals with the property graph model, which is richer (more complex) than any of the authors' models. The consequences of directed, labelled, property-containing relationships between pairs of nodes (also with properties) have not been investigated.

On the other hand, some of the research uses the notion of graph indices and/or pre-computed graph measures to achieve the desired results. This is in line with the approach taken by this project.

Of the vast amount of research on centrality and other measures, only a small fraction deals with dynamic graphs. An interesting opportunity for further research would, therefore, be the topic of efficient computation of these measures in the presence of updates in the graph. More specifically, it would be beneficial to find out, whether already pre-computed measures, cached values, and/or indices, which are kept up to date in changing graphs, can be used for calculating these measures. Such a research could build on top of this project, but is outside of scope for the moment.

Chapter 4

Theory

This project uses vertex degrees as an example analytical problem. More specifically, it asks the question whether it is possible to pre-compute and localise some metadata about the graph, keep it up to date whilst the graph is being mutated, and use it to efficiently answer inquiries about vertex degrees.

In this chapter, we first create a formal definition of the property graph model. We then formally define vertex degree in a property graph and study it in detail. Finally, we develop a theoretical solution for vertex degree caching with constant space and read time complexity.

When reading this chapter, it might be useful to refer to Appendix A, which provides a reference of all symbols used throughout.

4.1 Basic Definitions

Let us begin with a few own definitions that are useful for the remainder of this chapter. Other definitions are presented as needed.

As mentioned previously, a property graph is a directed graph with labelled edges. Loops are allowed and there can be multiple edges between the same pair of vertices, even if they are identical in terms of direction, label, and properties. Each property container (i.e., vertex or edge) can have an arbitrary number of arbitrary key-value pairs called attributes. If an attribute with a specific key exists for a property container, the value of such an attribute is defined. Otherwise, it is undefined. Formally,

Definition 4.1. A directed property graph with labelled edges is a set $G = (V, E, init, ter, label, attrp)$, such that $(V \cap E) = \emptyset$, $init : E \to V$, $ter : E \to V$, $label : E \to \Lambda$, and $attrp : C \times A \nrightarrow B$, where the elements of V are *vertices* or *nodes* of the graph G, the elements of E are its *edges* or *relationships*, i.e., pairs $(init(e), ter(e))$, the elements of Λ are edge *labels* or *types*, $C = (V \cup E)$ is the set of *property*

containers, the elements of A are *attribute* or *property keys* of property containers in C, the elements of B are *attribute* or *property values* of property containers in C, $init(e)$ is the *start node* or *initial vertex* of $e \in E$, $ter(e)$ is the *end node* or *terminal vertex* of e, and $label(e)$ is the label of e. *attrp* is a partial function on $C \times A$, which maps a set $\{c, \alpha\} \in (C \times A)$ to $\beta \in B$, if *defined* for c and α. Otherwise, it is *undefined* ($attrp(c, \alpha) \downarrow$). When there is no ambiguity, such a graph will be referred to as a *property graph* or simply *graph*.

For convenience, we define a new function, this time a total one, for mapping property containers and property keys to property values. To achieve that, we define a value that represents a non-existing property. Formally,

Definition 4.2. *attr* is a total function on $C \times A$, which maps each set $\{c, \alpha\} \in (C \times A)$ to $\beta \in (B \cup \{UNDEF\})$. $attr(c, \alpha) = UNDEF$ if and only if $attrp(c, \alpha) \downarrow$. Otherwise, $attr(c, \alpha) = attrp(c, \alpha)$.

The property set of a property container is the set of all properties (defined and undefined) on that property container. Formally,

Definition 4.3. For a property container $c \in C$, the *property set* $\Pi(c)$ is defined as $\Pi(c) \subseteq (A \times (B \cup UNDEF))$, such that $\forall\{\alpha, \beta\} \in \Pi(c), \beta = attr(c, \alpha)$.

In a directed graph, we can talk about the *direction* of an edge with respect to an incident vertex. Such a direction can either be *OUT*, or *IN*[1]. In the special case of a loop, we define the direction as *LOOP*. Formally,

Definition 4.4. Direction δ is a partial map $\delta : (E \times V) \nrightarrow (\Delta \cup \{LOOP\})$, where $\Delta = \{IN, OUT\}$. The map is defined as $\delta(e \in E, v \in e) = OUT$ if and only if $(init(e) = v) \wedge (ter(e) \neq v)$, $\delta(e, v) = IN$ if and only if $(ter(e) = v) \wedge (init(e) \neq v)$, and $\delta(e, v) = LOOP$ if and only if $(init(e) = v) \wedge (ter(e) = v)$. Otherwise, $\delta(e, v) \downarrow$ (i.e., in all other cases, e not an edge at v, so the function is undefined).

4.2 Vertex Degrees in Property Graphs

One of the goals of the project is to come up with a way to localise information about vertex degrees and keep them up to date, so that they can be determined quickly without the need to inspect every single edge at a vertex. First, however, we must take a deeper look at what vertex degrees actually mean in a property graph.

[1] We use *OUT* and *IN* for brevity in this chapter. Neo4j uses `OUTGOING` and `INCOMING`, respectively.

Example 4.5. Consider the graph shown in Figure 4.1, a modified version of the graph from Figure 2.1, which now also captures how people rated different movies. These ratings are expressed through `RATED` relationships with a `rating` property, taking on values from 1 (hated it) to 5 (loved it), and a timestamp in milliseconds since 1/1/1970, representing when the rating was created. The total outdegree of the vertex representing Pulp Fiction is now 2 and indegree is 6, but that information is of little semantic use.

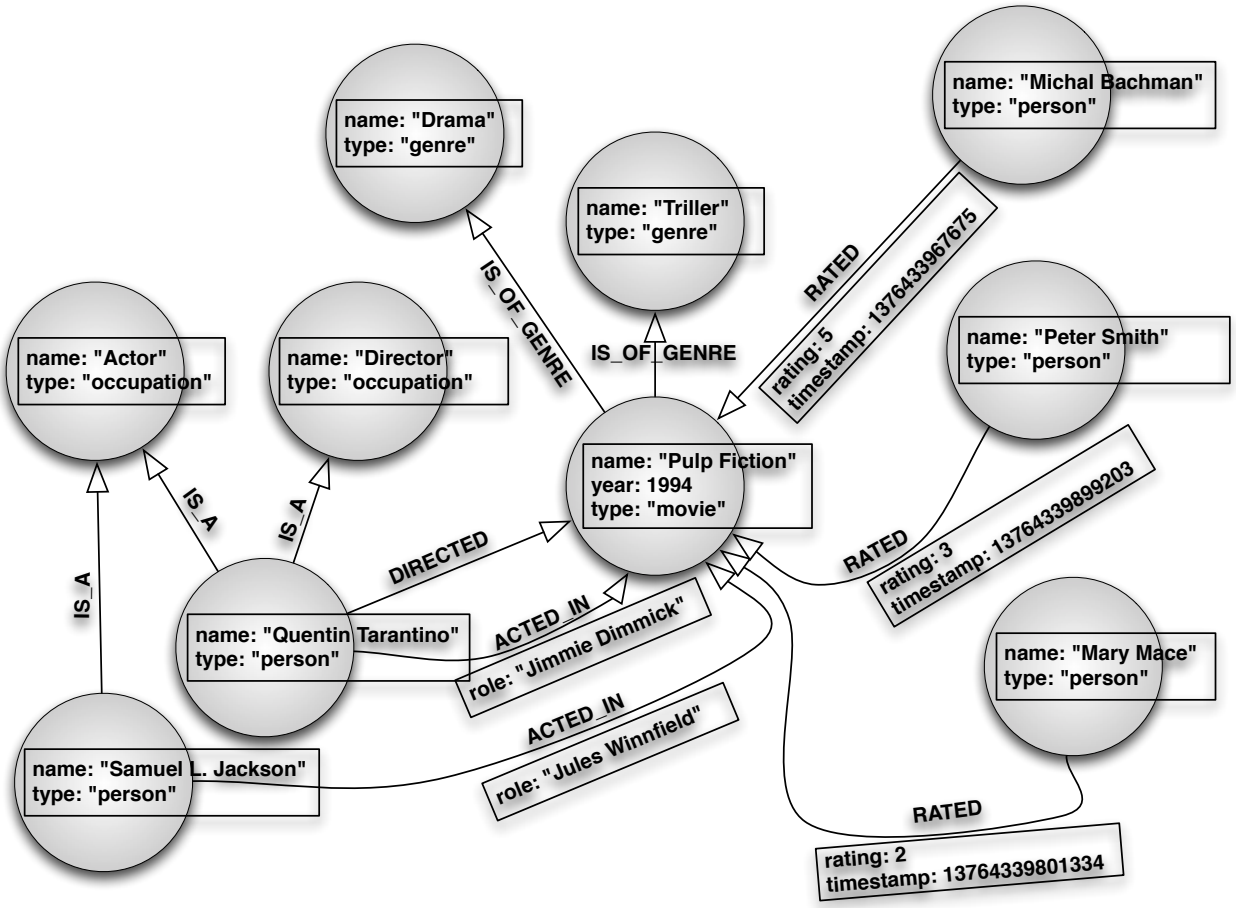

Figure 4.1: Movie Database with User Ratings

Property graph edge labels typically represent informational dimensions. As apparent from Figure 4.1, Pulp Fiction's incoming relationships of type `ACTED_IN` convey a completely different information from incoming relationships of type `RATED`. Thus, in order for the in/outdegree to have any informational value, it needs to be measured with respect to a relationship type.

Asking for the indegree of a movie with respect to relationships of type `RATED` can tell us, how many times the movie has been rated. However, it says nothing about how popular the movie is. Asking for the number of `RATED` relationships with the `rating` property greater than 3, on the other hand, can tell us how many people actually liked the movie.

4.2.1 Properties Description

Therefore, in order to gain useful insight by counting the number of edges at a vertex, we must specify the direction, type, and (optionally) properties to be taken into account when counting. Let us start with the properties and call their specification the properties description. Formally,

Definition 4.6. A *properties description* ϕ is map $\phi : A \to \Gamma$, assigning every property key $\alpha \in A$ a value $\gamma \in \Gamma$, where each element of Γ, also called a *property constraint*, is a predicate on $\beta = attr(c, \alpha)$, i.e., the value of property with key α on property container c, such that $\gamma(\beta) \to \{true, false\}$. A predicate indicating that the value of the property can take on any value, denoted ?, is formally defined as $\forall \beta, ?(\beta) = true$. Sometimes, we refer to ? as "wildcard".

Example 4.7. Using the movie graph from Figure 4.1 again, if we were to attempt counting a movie node's incoming relationships of type `RATED` with property `rating` grater than 3 and property `timestamp` smaller than 1376433967675, we could construct a properties description ϕ_1 as follows (ignoring any node properties in the model for brevity):
$\phi_1 = \{rating \to (\beta > 3), timestamp \to (\beta < 1376433967675), role \to ?\}$.
If we were to construct a properties description for a relationship that does not have the `timestamp` property, we could do so by defining
$\phi_2 = \{rating \to (\beta > 3), timestamp \to (\beta = UNDEF), role \to ?\}$.

We say that a property container matches a properties description if every property constraint in the properties description is satisfied by the properties of the property container. A property constraint for a property is satisfied, if the predicate (constraint) assigned to the property key returns *true* for the property value on the property container. Formally,

Definition 4.8. Property container c *matches properties description* ϕ, denoted as $matches(c, \phi)$, if and only if $(\forall \alpha \in A, \gamma(\beta) = true)$, where $\gamma = \phi(\alpha)$ and $\beta = attr(c, \alpha)$.

Example 4.9. Let the set of all property keys be $A = \{rating, timestamp, role\}$. Let c be a property container with a single property `rating` set to 2. Let ϕ be a properties description, defined as $\phi = \{rating \to (\beta < 3), timestamp \to (\beta = UNDEF), role \to ?\}$. Then:

$\alpha_1 = rating$,
$\gamma_1 = \phi(\alpha_1) = \phi(rating) = (\beta_1 < 3)$,
$\beta_1 = attr(c, \alpha_1) = attr(c, rating) = 2$,
$\gamma_1(\beta_1) = \gamma_1(2) = (2 < 3) = true$.

$\alpha_2 = timestamp$,
$\gamma_2 = \phi(\alpha_2) = \phi(timestamp) = (\beta_2 = UNDEF)$,
$\beta_2 = attr(c, \alpha_2) = attr(c, timestamp) = UNDEF$,
$\gamma_2(\beta_2) = \gamma_2(UNDEF) = (UNDEF = UNDEF) = true$.

$\alpha_3 = role$,
$\gamma_3 = \phi(\alpha_3) = \phi(role) = ? = true$ (by definition).

Since $(\forall \alpha \in A, \gamma(\beta) = true)$, we can say that $matches(c, \phi)$, i.e., property container c matches properties description ϕ.

4.2.2 Relationship Description

Now that we have defined properties description, we can use it as a building block for defining a relationship description.

Definition 4.10. A *relationship description* ψ is a set $\psi = (\delta_\psi, \lambda_\psi, \phi)$, where $\delta_\psi \in \Delta$ is a direction, $\lambda_\psi \in \Lambda$ is a relationship type, and ϕ is a properties description.

Example 4.11. Using the movie graph from Figure 4.1 once again, if we were to attempt counting a movie node's incoming relationships of type `RATED` with property `rating` grater than 3 and property `timestamp` smaller than 1376433967675, we could construct a description ψ_1 as follows: $\psi_1 = (IN, RATED, \{rating \rightarrow (\beta > 3), timestamp \rightarrow (\beta < 1376433967675), role \rightarrow ?\})$. If we were to construct a description of a relationship that does not have the `timestamp` property, we could do so by defining $\psi_2 = (IN, RATED, \{rating \rightarrow (\beta > 3), timestamp \rightarrow (\beta = UNDEF), role \rightarrow ?\})$.

We now define what it means that a relationship matches a relationship description. Informally, a node's relationship matches a relationship description if their types are equal, their directions match, and the relationship's properties match the properties description on the relationship description. Directions match if they are equal (the relationship's direction is determined with respect to a node), or if the direction of the relationship is *LOOP*. Formally,

Definition 4.12. Direction $\delta(e, v) \in (\Delta \cup \{LOOP\})$ of a relationship e at node v matches a direction $\delta_\psi \in \Delta$, denoted as $matches(\delta(e, v), \delta_\psi)$, if and only if $(\delta(e, v) = \delta_\psi) \vee (\delta(e, v) = LOOP)$.

Definition 4.13. Relationship e at node v *matches a description* $\psi = (\delta_\psi, \lambda_\psi, \phi)$, denoted as $matches(e, \psi, v)$, if and only if $(\lambda_\psi = label(e)) \wedge matches(\delta(e, v), \delta_\psi) \wedge matches(e, \phi)$.

Example 4.14. Let the set of all property keys be $A = \{rating, timestamp, role\}$. Let e be an outgoing relationship of type `RATED` at node v with a single property `rating` set to 2. Let ψ be a relationship description, defined as $\psi = (IN, RATED, \{rating \rightarrow (\beta < 3), timestamp \rightarrow (\beta = UNDEF), role \rightarrow ?\})$. Then:

$\phi = \{rating \rightarrow (\beta < 3), timestamp \rightarrow (\beta = UNDEF), role \rightarrow ?\}$, and $matches(e, \phi)$ (Example 4.9),
$\lambda_\psi = RATED$, $label(e) = RATED$, thus $\lambda_\psi = label(e)$,
$\delta_\psi = IN$, $\delta(e, v) = OUT$, thus $\neg matches(\delta(e, v), \delta_\psi)$.

Since $\neg matches(\delta(e, v), \delta_\psi)$, we can say that $\neg matches(e, \psi, v)$, i.e., relationship e at node v does not match relationship description ψ.

4.2.3 Vertex Degree

Finally, we define the degree of a node v in a property graph as

Definition 4.15. The vertex degree $d(v, \psi)$ of node v with respect to relationship description ψ is the number $d(v, \psi) = |E(v)|$ of relationships at v such that $\forall e \in E(v)$, $matches(e, \psi, v)$.

A note on loops is in order. We have only defined indegrees and outdegrees, but no such thing as “loopdegrees". This is because loops are a rare special case in real-life graph models. Since the property graph model used by Neo4j allows loops, we must not ignore them. As described later in Section 4.4, we indeed do account for them to the extent that they add one to the outdegree and one to the indegree of the vertex with the loop, as already mentioned in Section 2.2. However, since *LOOP* is not a valid direction in relationship descriptions, no capability is provided for counting loops only, which is consistent with the non-existence of “loopdegree" of a vertex.

4.3 General-to-Specific Ordering

4.3.1 Properties Description Ordering

Next, we define general-to-specific ordering of properties descriptions, inspired by Mitchell [16]. Given properties descriptions ϕ and ϕ', ϕ is more general or equal to ϕ' if an only if any property container that matches ϕ' also matches ϕ. Formally,

Definition 4.16. For any two properties descriptions ϕ and ϕ', *ϕ is more general than or equal to ϕ'*, written as $\phi \geq_g \phi'$, if and only if $\forall e \in E, matches(e, \phi') \rightarrow matches(e, \phi)$. *$\phi$ is (strictly) more general*

than ϕ', written as $\phi >_g \phi'$, if and only if $(\phi \geq_g \phi') \wedge (\phi \neq \phi')$. Finally, ϕ' *is (strictly) more specific than* ϕ if and only if $\phi >_g \phi'$.

The $\geq_g$ relation is reflexive, anti-symmetric, and transitive, and defines a partial order over any set Φ of properties descriptions, forming a lattice, example of which is shown in Figure 4.2. The example assumes only 2 property keys in A, i.e., $A = \{rating, timestamp\}$. As proven below, the most specific properties descriptions ϕ_{maxs} are those with all constraints in the form of $(\beta = x)$, where x can be any value (integer, string, etc.), or *UNDEF*. The most general properties description ϕ_{maxg} is one where all property constraints are wildcards, i.e., $\forall \alpha \in A, \phi(\alpha) \rightarrow ?$, since any property container matches that description.

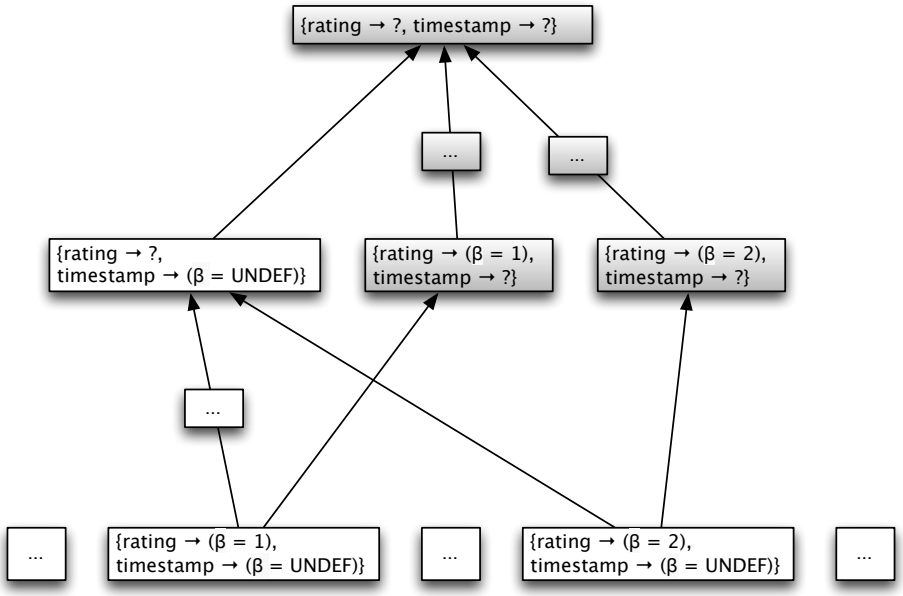

Figure 4.2: Part of a Lattice of Properties Descriptions

If we allow any predicates to act as property constraints, the set Φ' of all properties descriptions more general than a given properties description ϕ is infinite. This is illustrated by the grey boxes in Figure 4.2 and in a more detail in Figure 4.3. Formally,

Theorem 4.17. $\exists\{\phi, \Phi'\}$ *such that* $\forall \phi' \in \Phi'$, $(\phi' \geq_g \phi) \wedge (|\Phi'| = \infty)$.

Proof. We prove by induction. Let γ be a property constraint, assigned by ϕ to a property key α, i.e., $\gamma = \phi(\alpha)$. Let γ take the form of $\gamma(e, \alpha) = (\beta < x)$, where $\beta = attr(e, \alpha)$ and $x \in \mathbb{R}$. Let $\gamma'(e, \alpha) = (\beta < x')$, where $x' > x$. If ϕ' is obtained by replacing γ with γ' in ϕ, then $\phi' \geq_g \phi$, since $x' > x > \beta$. There are infinitely many $x' \in \mathbb{R}$, such that $x' > x$. Thus, there are infinitely many obtainable γ' and, consequently, ϕ', making $|\Phi'| = \infty$. □

A few theorems related to maximally specific and maximally general properties descriptions are now presented and proven.

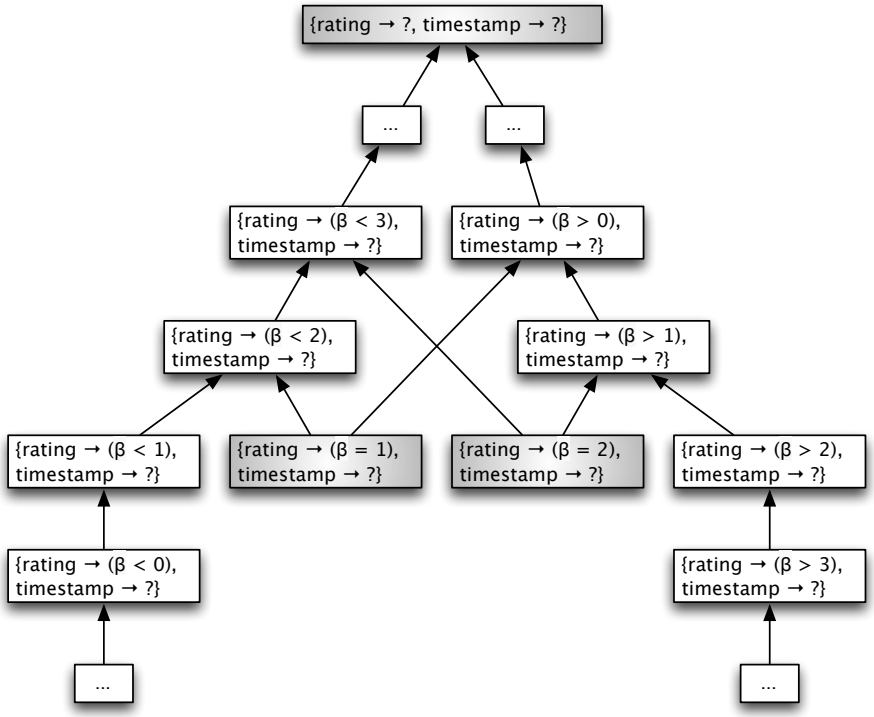

Figure 4.3: Detailed View on Part of an Infinite Lattice of Properties Descriptions

Theorem 4.18. *For each property container $c \in C$, there is exactly one maximally specific matching properties description $\phi_{maxs}(c)$. Formally, $\forall c \in C, \exists \phi_{maxs}(c)$, such that matches$(c, \phi_{maxs}(c))$ and $\forall \phi \neq \phi_{maxs}(c)$, such that matches$(c, \phi)$, $\phi >_g \phi_{maxs}(c)$.*

Proof. We proof by construction. Let $\phi_{maxs}(c)$ for a property container c be constructed as $\forall \alpha \in A$, $\phi_{maxs}(c, \alpha) \rightarrow \gamma$, where $\gamma(\beta) = (\beta = attr(c, \alpha))$. First, note that $matches(c, \phi_{maxs}(c))$ by Definition 4.8. For any $\alpha \in A$, the predicate $\gamma(\beta)$ is only *true* for a single value of β, which is the value of property α on property container c. To construct a ϕ from $\phi_{maxs}(c)$ by replacing γ with γ', there are three options. First, $(\gamma(\beta) = true) \leftrightarrow (\gamma'(\beta) = true)$, but then $\gamma = \gamma'$ and $\phi = \phi_{maxs}(c)$. Second, $\gamma'(\beta) = true$ for fewer values of β than γ, but since $\gamma(\beta) = true$ only for a single value of β, then $\forall \beta \ \gamma'(\beta) = false$, thus $\neg matches(c, \phi)$. Third, $\gamma'(\beta) = true$ for more values of β than γ, but then $\phi >_g \phi_{maxs}(c)$. Thus $\forall c \in C, \exists \phi_{maxs}(c)$, such that $matches(c, \phi_{maxs}(c))$ and $\forall \phi \neq \phi_{maxs}(c)$, such that $matches(c, \phi)$, $\phi >_g \phi_{maxs}(c)$. □

Theorem 4.19. *The most specific properties descriptions of two property containers are equal if and only if the property sets of the property containers are equal. Formally, $\forall c_1, c_2 \in C$, $(\phi_{maxs}(c_1) = \phi_{maxs}(c_2)) \leftrightarrow (\Pi(c_1) = \Pi(c_2))$.*

Proof. We prove directly. $\forall c_1, c_2 \in C$, $\phi_{maxs}(c_1) = \phi_{maxs}(c_2)$ if and only if $\forall \alpha \in A$, $\phi_{maxs}(c_1, \alpha) = \phi_{maxs}(c_2, \alpha)$, which is true if and only if $\forall \alpha \in A$, $attr(c_1, \alpha) = attr(c_2, \alpha)$ by construction of ϕ_{maxs} from previous proof. Finally, $attr(c_1, \alpha) = attr(c_2, \alpha)$ $\forall \alpha \in A$ if and only if $\Pi(c_1) = \Pi(c_2)$. □

Theorem 4.20. *A property container matches the most specific properties description of another container if and only if their property sets are equal. Formally,* $\forall c_1, c_2 \in C$, $matches(c_1, \phi_{maxs}(c_2)) \leftrightarrow (\Pi(c_1) = \Pi(c_2))$.

Proof. We prove directly. $\forall c_1, c_2 \in C$, $matches(c_1, \phi_{maxs}(c_2))$ if and only if $\forall \alpha \in A$, $\gamma_2(attr(c_1, \alpha)) = true$, where $\gamma_2 = \phi_{maxs}(c_2, \alpha)$. This is the case if and only if $\gamma_2 = (\beta = attr(c_1, \alpha))$, which is true if and only if $attr(c_2, \alpha) = attr(c_1, \alpha)$, by construction of ϕ_{maxs} from the proof of Theorem 4.18. Since $\forall \alpha \in A$, $attr(c_2, \alpha) = attr(c_1, \alpha)$, then $\Pi(c_1) = \Pi(c_2)$. □

Theorem 4.21. *The most general properties description is* ϕ_{maxg}*, such that* $\forall \alpha \in A, \phi_{maxg}(\alpha) \rightarrow ?$.

Proof. We prove directly. Any property container matches a properties description ϕ_{maxg}, such that $\forall \alpha \in A, \phi_{maxg}(\alpha) \rightarrow ?$, by definition of ?. Since for any $c \in C$, $true \rightarrow matches(c, \phi_{maxg})$, $\forall \phi \in \Phi$, where Φ is the set of all properties descriptions, $matches(c, \phi) \rightarrow matches(c, \phi_{maxg})$, thus $\phi \leq_g \phi_{maxg}$. □

Finally, we define mutually exclusive properties descriptions, informally as such properties descriptions that have no chance of matching a single property container at the same time. Another way to put this is that they do not have a common descendant in the lattice of property descriptions. Formally,

Definition 4.22. Properties descriptions ϕ_1 and ϕ_2 are *mutually exclusive*, denoted as $mutex(\phi_1, \phi_2)$, if and only if $\nexists c$, such that $matches(\phi_1, c) \wedge matches(\phi_2, c)$.

4.3.2 Relationship Description Ordering

We can now define similar general-to-specific ordering on relationship descriptions. Since each relationship description has a single relationship type and direction, the ordering depends purely on the properties descriptions. Formally,

Definition 4.23. Relationship description $\psi = (\delta_\psi, \lambda_\psi, \phi)$ *is more general or equal to* a relationship description $\psi' = (\delta'_\psi, \lambda'_\psi, \phi')$, denoted as $\psi \geq_g \psi'$, if and only if $\forall e \in E$ and $\forall v \in e$, $matches(e, \psi', v) \rightarrow matches(e, \psi, v)$. Equivalently, $\psi \geq_g \psi'$, if and only if $(\lambda_\psi = \lambda'_\psi) \wedge (\delta_\psi = \delta'_\psi) \wedge (\phi \geq_g \phi')$ (from Definitions 4.13 and 4.16). Again, ψ *is (strictly) more general than* ψ', written as $\psi >_g \psi'$, if and only if $(\psi \geq_g \psi') \wedge (\psi \neq \psi')$. Finally, ψ' *is (strictly) more specific than* ψ if and only if $\psi >_g \psi'$.

Under the condition that constraints in properties descriptions are allowed to be arbitrary predicates, the lattice formed by relationship description general-to-specific ordering is, again, infinite, since it is based on the lattice formed by properties descriptions, proven above to be infinite. We now present a few useful theorems for relationship descriptions, which are directly related to the theorems proven for properties descriptions.

Theorem 4.24. *For each relationship e at a node $v \in V$, which is not a loop, there is exactly one maximally specific matching relationship description of the form $\psi_{maxs}(e,v) = (\delta_\psi, \lambda_\psi, \phi) =$*
$= (\delta(e,v), label(e), \phi_{maxs}(e))$.

Proof. We prove directly. From Definition 4.13, $matches(e, \psi, v)$ if and only if $\delta_\psi = \delta(e,v)$ (for non-loop edges), $\lambda_\psi = label(e)$, and $matches(\phi, e)$. From Theorem 4.18 and proof thereof, the most specific ϕ such that $matches(\phi, e)$, is $\phi_{maxs}(e)$. Thus, $\psi_{maxs}(e,v) = (\delta(e,v), label(e), \phi_{maxs}(e))$. □

Theorem 4.25. *For each relationship e at a node $v \in V$, which is a loop, there are exactly two maximally specific matching relationship descriptions of the forms $\psi_{maxs1}(e,v) = (IN, label(e), \phi_{maxs}(e))$ and $\psi_{maxs2}(e,v) = (OUT, label(e), \phi_{maxs}(e))$.*

Proof. We prove directly. From Definition 4.13, $matches(e, \psi, v)$ if and only if $matches(\delta_\psi, \delta(e,v))$, $\lambda_\psi = label(e)$, and $matches(\phi, e)$. From Theorem 4.18 and proof thereof, the most specific ϕ such that $matches(\phi, e)$, is $\phi_{maxs}(e)$. From Definition 4.12, $\forall \delta \in \Delta$, $matches(\delta, LOOP)$. Since $\Delta = \{IN, OUT\}$, there are exactly two maximally specific matching relationship descriptions of the forms $\psi_{maxs1}(e,v) =$ $(IN, label(e), \phi_{maxs}(e))$ and $\psi_{maxs2}(e,v) = (OUT, label(e), \phi_{maxs}(e))$. □

From this point onward, as a consequence of this result and in the interest of simplicity, each relationship which is a loop is treated as two different relationships, one incoming and the other one outgoing.

Theorem 4.26. *The most specific relationship descriptions of two relationships at the same node are equal if and only if the directions, labels, and property sets of the relationships are equal. Formally, $\forall e_1, e_2 \in E(v)$, $(\psi_{maxs}(e_1,v) = \psi_{maxs}(e_2,v)) \leftrightarrow ((\delta(e_1,v) = \delta(e_2,v)) \wedge (label(e_1) = label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2)))$.*

Proof. We prove directly.

$(\psi_{maxs}(e_1,v) = \psi_{maxs}(e_2,v)) \leftrightarrow ((\delta(e_1,v) = \delta(e_2,v)) \wedge (label(e_1) = label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2)))$

$((\delta(e_1,v), label(e_1), \phi_{maxs}(e_1)) = (\delta(e_2,v), label(e_2), \phi_{maxs}(e_2))) \leftrightarrow ((\delta(e_1,v) = \delta(e_2,v)) \wedge (label(e_1) =$ $label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2)))$ (from Theorem 4.24)

$((\delta(e_1,v) = \delta(e_2,v)) \wedge (label(e_1) = label(e_2)) \wedge (\phi_{maxs}(e_1) = \phi_{maxs}(e_2))) \leftrightarrow ((\delta(e_1,v) = \delta(e_2,v)) \wedge$ $(label(e_1) = label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2)))$ (logical re-arrangement)

$(\phi_{maxs}(e_1) = \phi_{maxs}(e_2)) \leftrightarrow (\Pi(e_1) = \Pi(e_2))$, which is *true* (from Theorem 4.19) □

Theorem 4.27. *A relationship matches the most specific relationship description of another relationship if and only if their directions, labels, and property sets are equal. Formally,*
$\forall e_1, e_2 \in E(v)$, $matches(e_1, \psi_{maxs}(e_2,v), v) \leftrightarrow ((\delta(e_1) = \delta(e_2)) \wedge (label(e_1) = label(e_2)) \wedge (\Pi(e_1) =$ $\Pi(e_2)))$.

Proof. We prove directly.

$matches(e_1, \psi_{maxs}(e_2, v), v) \leftrightarrow ((\delta(e_1) = \delta(e_2)) \wedge (label(e_1) = label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2)))$.

$((\delta(e_1) = \delta(e_2)) \wedge (label(e_1) = label(e_2)) \wedge matches(e_1, \phi_{maxs}(e_2))) \leftrightarrow ((\delta(e_1) = \delta(e_2)) \wedge (label(e_1) = label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2)))$.

Since $matches(e_1, \phi_{maxs}(e_2)) \leftrightarrow (\Pi(e_1) = \Pi(e_2))$ (from Theorem 4.20),

$((\delta(e_1) = \delta(e_2)) \wedge (label(e_1) = label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2))) \leftrightarrow ((\delta(e_1) = \delta(e_2)) \wedge (label(e_1) = label(e_2)) \wedge (\Pi(e_1) = \Pi(e_2)))$. □

Finally, we define mutually exclusive relationship descriptions, informally as such relationship descriptions that have no chance of matching a single relationships at the same time. Another way to put this is that they do not have a common descendant in the lattice of relationship descriptions. Formally,

Definition 4.28. Relationship descriptions $\psi_1 = (\delta_{\psi 1}, \lambda_{\psi 1}, \phi_1)$ and $\psi_2 = (\delta_{\psi 2}, \lambda_{\psi 2}, \phi_2)$ are *mutually exclusive*, denoted as $mutex(\psi_1, \psi_2)$, if and only if $\nexists e$, such that $matches(\psi_1, e) \wedge matches(\psi_2, e)$. It follows directly from Definitions 4.13 and 4.22 that $mutex(\psi_1, \psi_2) \leftrightarrow ((\delta_{\psi 1} \neq \delta_{\psi 2}) \vee (\lambda_{\psi 1} \neq \lambda_{\psi 2}) \vee (mutex(\phi_1, \phi_2)))$.

4.4 Naive Caching

It should now be possible to determine the number of relationships at a node by creating a relationship description, inspecting every single relationship at the node, and counting those that match the description. As mentioned before, however, this could be too costly. Instead, we would like to cache that information, so it can be found faster. One way of caching such information would be creating the most specific description of the relationship and storing the number of relationships matching that description for each node.

Definition 4.29. $K(v)$ is a set, called the *degree cache* of vertex v. Each element $k \in K(v)$ of this set, called a *cached degree*, is a tuple $(\psi \in \Psi, n \in \mathbb{N})$, where Ψ is the set of all possible relationship descriptions and $\mathbb{N}$ is the set of all natural numbers[2]. For a vertex with no edges, the cache is empty, i.e., $|K(v)| = 0$.

Creating Relationships When a relationship e is created[3] at node v, a cached degree $k \in K(v)$ is found such that $matches(e, \psi \in k, v)$. The cached degree $k = (\psi, n)$ is removed from $K(v)$ and replaced

[2] defined as any positive integer and 0

[3] as mentioned previously, a relationship that is a loop is now treated as two different relationships, one outgoing, the other one incoming

by $k' = (\psi, n+1)$. In other words, the cached degree is incremented by 1. If no such k exists in $K(v)$, the most specific description $\psi_{maxs}(e, v)$ is created for e, and $k = (\psi_{maxs}(e, v), 1)$ is inserted into $K(v)$. By inserting elements into the cache this way, we claim two statements to be true.

Theorem 4.30. *There are no two elements $k_1 \in K(v)$ and $k_2 \in K(v)$, such that $\psi_1 \in k_1 = \psi_2 \in k_2$.*

Proof. We prove by contradiction. Assume that to cache $K(v)$ for vertex v, already containing cached degree k_1, a second element k_2 such that $\psi_2 \in k_2 = \psi_1 \in k_1$, has been inserted. Then ψ_2 must be the most specific relationship description for some relationship $e_2 \in v$, i.e., $\psi_2 = \psi_{maxs}(e_2)$. When $\psi_1 = \psi_2$, then it must be the case that $matches(e_2, \psi_1, v)$. Then $n_1 \in k_1$ would have been incremented by 1 instead of k_2 being inserted, which contradicts the assumption. □

Theorem 4.31. *There is always at most 1 cached degree $k \in K(v)$ for any $e \in E(v)$, such that $matches(e, \psi \in k, v)$.*

Proof. We prove by contradiction. Let cache $K(v)$ for vertex v contain two elements $k_1, k_2 \in K(v)$. Let ψ_1 be the most specific relationship description of some relationship $e_1 \in v$, such that $\psi_1 \in k_1 = \psi_{maxs}(e_1, v)$. Similarly, let ψ_2 be the most specific relationship description of some relationship $e_2 \in v$, such that $\psi_2 \in k_2 = \psi_{maxs}(e_2, v)$. Now assume that $\exists e_3 \in v$, such that $matches(e_3, \psi_1, v) \wedge matches(e_3, \psi_2, v)$. Using Theorem 4.27, this means that $(\delta(e_3, v) = \delta(e_1, v) = \delta(e_2, v)) \wedge (label(e_3) = label(e_1) = label(e_2)) \wedge (\Pi(e_3) = \Pi(e_1) = \Pi(e_2))$. Then, by construction of maximally specific relationship description from Theorem 4.24, $\psi_1 = \psi_2$. This is a contradiction to Theorem 4.30. □

Deleting Relationships When a relationship is deleted at node v, a cached degree $k \in K(v)$ is found such that $matches(e, \psi \in k, v)$. The cached degree $k = (\psi, n)$ is removed from $K(v)$ and replaced by $k' = (\psi, n-1)$, if $n > 1$ (otherwise it is just removed). In other words, the cached degree is decremented by 1 or removed when already equal to 1. k must exist, since it was created when the first relationship with description $\psi_{maxs}(e, v)$ was created and incremented every time thereafter, when another relationship with the same description was created.

Changing Relationships Finally, when a relationship is changed, the same operations take place as if the old version of the relationship was deleted and the new version created. This way, the degree cache is up to date at all times, holding the correct vertex degrees.

Querying Cache To determine the degree of a vertex v, one would first construct a relationship description ψ_{query} of the relationships to count. The degree of vertex v with respect to description

ψ_{query} is then the sum of the values of all cached degrees in cache $K(v)$, such that the relationship description of the cached degree is more specific or equal to ψ_{query}. Formally,

Theorem 4.32. *The degree of a vertex $v \in V$ with respect to relationship description ψ_{query} is*

$$d(v, \psi_{query}) = \sum_{k \in K(v), s.t. \psi_{query} \geq_g \psi \in k} n \in k.$$

Proof. We prove directly. By Definition 4.23 and Theorem 4.24, $matches(e, \psi, v) \longleftrightarrow (\psi \geq_g \psi_{maxs}(e, v))$, where e is an edge at vertex v. Given the construction of cache $K(v)$, i.e., the fact that cached relationship counts are up to date with actual counts, the definition of vertex degree from Definition 4.15 is equivalent to the vertex degree definition from Theorem 4.32 above. □

Space Complexity There is an apparent problem with this approach to caching. In the worst case, when a property for each relationship takes on a different value (as in the case of timestamps, for instance), the space complexity of this caching is $O(||G||)$, i.e., linearly proportional to the number of edges in the graph. This clearly defeats its very purpose, because we already have such a cache in the graph, the vertices themselves.

4.5 Constant Space Complexity Caching

Whilst the number of property values $\beta \in B$ for a property key $\alpha \in A$ can theoretically be of order $O(||G||)$, as mentioned above, the number of relationship types $|\Lambda|$ in any real-life graph is not typically greater than 10 or 20. This is because the types represent real kinds of relationships between entities, such as *is in*, *is of type*, *likes*, *follows*, and so on. A graph model with hundreds or thousands of relationship types would be incomprehensible and, consequently, of little value, since the purpose of these types is semantic meaning for humans. Moreover, in a system that uses a graph database, the relationship types are typically decided by designers of the system, rather than derived from user input. Thus, the number of relationship types in the graph can usually be approximated reasonably accurately up front.

4.5.1 Cache Compaction

We propose a relationship description generalisation operation, a compaction threshold τ, and a cache compaction operation, which results in $|K(v)| \leq \tau$, effectively bringing the space complexity of the degree cache to $O(1)$.

The idea behind decreasing the space requirement for the degree cache is storing the number of relationships at a vertex using a more general relationship description than their most specific one. From Theorem 4.17 and subsequent discussion, we know that the number of relationship descriptions more general than a given relationship description is infinite. Bearing in mind that “value is greater than 3” is as valid a property constraint as “value starts and ends with a capital letter" (for a string-typed value), picking a reasonable generalisation would be very hard without imposing any limitations.

We propose the following limited generalisation operation, which produces a more general relationship description of a given relationship description by replacing one of its property constraints, not already equal to ?, by ?. Formally,

Definition 4.33. *Generalisation* of a relationship description $\psi = (\delta_\psi, \lambda_\psi, \phi)$ produces a relationship description $\psi' = (\delta_\psi, \lambda_\psi, \phi')$, such that $\phi'(\alpha_?) = ?$ and $\forall \alpha \in (A \setminus \{\alpha_?\})$, $\phi'(\alpha) = \phi(\alpha)$, where $\phi(\alpha_? \in A) \neq ?$. Such an operation is written as $\psi' = gen(\psi)$. The most general relationship description of relationship e at node v is $\psi_{maxg}(e, v) = (\delta(e, v), label(e), \phi_{maxg}(e))$ and since $\forall \alpha \in A$, $\phi_{maxg}(e, \alpha) = ?$, $gen(\psi_{maxg}(e, v)) = \emptyset$. That is to say, the most general relationship description cannot be generalised any further.

The generalisation set of a relationship description is the set of all more or equally general relationship descriptions, recursively produced by the generalisation operation defined above. Formally,

Definition 4.34. The set $\Psi_g(\psi)$, called *generalisation set* of relationship description ψ, is the set of all ψ_g, such that $\forall \psi_g \in \Psi_g(\psi)$, $\psi_g = gen(\psi'_g \in (\{\psi\} \cup \Psi_g(\psi)))$.

Since the number of property constraints that can be replaced by ? is bounded above by $|A|$, the set Ψ_g, partially ordered using the general-to-specific ordering, has a finite number of elements. An example lattice, formed by such a set, is shown in Figure 4.4. A is assumed to be $A = \{rating, time, role\}$.

We further propose a *compaction operation*, which results in a reduction of the space requirement for the degree cache of a vertex. In plain language, the operation works as follows. First, a generalisation set for each cached degree’s relationship description is produced and all elements in all the generalisation sets are inserted into a new set, called generalisation superset. The superset is then ordered, based on some criteria discussed later, from best to worst generalisation. The first generalisation in such an ordered superset, which is more general or equal to at least 2 cached degree relationship descriptions, is then selected and a new cached degree is formed out of it. The value of the newly cached degree is the sum of the cached degrees, whose relationship descriptions were more or equally specific to the relationship description of the newly cached degree. These are then removed from the cache. The pseudocode in Algorithm 4.1 illustrates the compaction operation on cache $K(v)$ for vertex v.

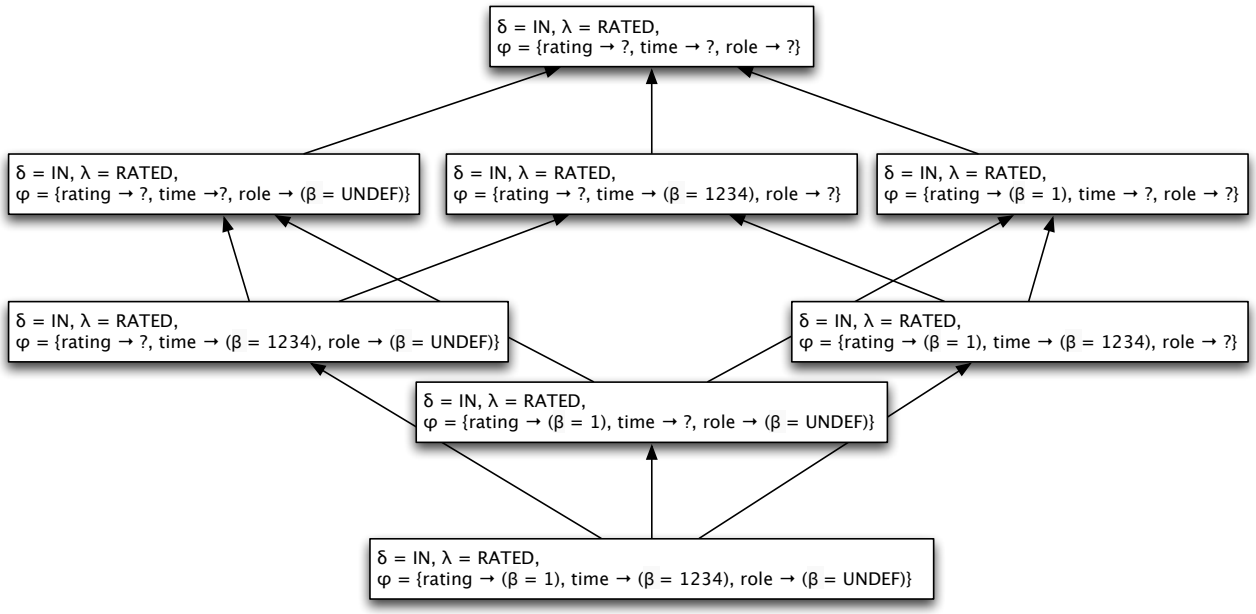

Figure 4.4: Lattice Formed by the Generalisation Set of a Relationship Description

Algorithm 4.1 Compaction Operation

function COMPACT($K(v)$)
  $\Psi_{sg} = \emptyset$ ▷ generalisation superset
  for k in $K(v)$ **do** ▷ populate generalisation superset
    $\Psi_{sg} = \Psi_{sg} \cup \Psi_g(\psi \in k)$
  end for
  $\Psi_{sg} = PCF(\Psi_{sg}, K(v))$ ▷ sort from best to worst, using a PCF function introduced later
  for $\psi_{candidate}$ in Ψ_{sg} **do**
    $i = 0$ ▷ number of cached degrees that will be compacted out by this candidate
    for k in $K(v)$ **do**
      if $\psi_{candidate} \geq_g \psi \in k$ **then**
        $i = i + 1$
      end if
    end for
    if $i > 1$ **then**
      $n_{new} = 0$ ▷ value of the new cached degree
      for k in $K(v)$ **do**
        if $\psi_{candidate} \geq_g \psi \in k$ **then**
          $n_{new} = n_{new} + n \in k$
          $K(v) = K(v) \setminus k$ ▷ remove a cached degree
        end if
      end for
      $k_{new} = (\psi_{candidate}, n_{new})$
      $K(v) = K(v) \cup \{k_{new}\}$ ▷ add a new cached degree
      return $K(v)$ ▷ exit
    end if
  end for
end function

With the compaction operation defined and explained, we now state and prove a few related theorems.

Theorem 4.35. *If $|K(v)| > 2|\Lambda|$, then the compaction operation results in $K(v)' = COMPACT(K(v))$, such that $|K(v)'| < |K(v)|$.*

Proof. We prove directly. Assume that $|K(v)| > 2|\Lambda|$ and $K(v)' = COMPACT(K(v))$. There are $|\Delta||\Lambda| = 2|\Lambda|$ distinct pairs ($\delta \in \Delta, \lambda \in \Lambda$). Since $|K(v)| > 2|\Lambda|$, then there must be $k_1, k_2 \in K(v)$ and $\psi_1, n_1 \in k_1, \psi_2, n_2 \in k_2$, such that $(\psi_1 \neq \psi_2) \wedge (\delta_{\psi_1} = \delta_{\psi_2}) \wedge (\lambda_{\psi_1} = \lambda_{\psi_2})$. Then for an imaginary edge e at v, such that $\delta(e,v) = \delta_{\psi_1} = \delta_{\psi_2}$ and $\lambda(e) = \lambda_{\psi_1} = \lambda_{\psi_2}$, and its most general relationship description $\psi_{maxg}(e,v)$, it must be the case that $(\psi_{maxg}(e,v) \neq \psi_1) \wedge (\psi_{maxg}(e,v) \neq \psi_2)$.

This is because if that was not the case, either $(\psi_{maxg}(e,v) = \psi_1) \wedge (\psi_{maxg}(e,v) = \psi_2)$, but then $\psi_1 = \psi_2$, which has been established to be false earlier in this proof. Alternatively, $(\psi_{maxg}(e,v) = \psi_1) \vee (\psi_{maxg}(e,v) = \psi_2)$, which means $(\psi_1 \geq_g \psi_2) \vee (\psi_2 \geq_g \psi_1)$ (from the Definition 4.33 of $\psi_{maxg}(e,v)$), but then the more specific of the two would have been removed on line 19 of the compaction operation. Consequently, it must be the case $(\psi_{maxg}(e,v) \geq_g \psi_1) \wedge (\psi_{maxg}(e,v) \geq_g \psi_2)$, which means k_1 and k_2 would be removed on line 19 and replaced by $k_{new} = (\psi_{maxg}(e,v), n_1 + n_2)$ on lines 22 - 24, resulting in $|K(v)'| < |K(v)|$. □

We now re-state an earlier theorem (Theorem 4.30) and prove that it holds even after the cache has gone through a compaction operation.

Theorem 4.36. *There are no two elements $k_1 \in K(v)$ and $k_2 \in K(v)$, such that $\psi_1 \in k_1 = \psi_2 \in k_2$.*

Proof. We prove directly. Theorem 4.30 proves that this is the case when no compaction has occurred. After a compaction operation on cache $K(v)$, this is also true, because if $\psi_1 = \psi_2$, then $(\psi_1 \geq_g \psi_2) \wedge (\psi_2 \geq_g \psi_1)$, which means one of them would be removed on line 19 of Algorithm 4.1, because line 17 would evaluate to *true*. □

Clearly, when the cache has gone through a compaction, it has lost its ability to provide vertex degrees with respect to all possible relationship descriptions. Specifically,

Definition 4.37. Cache $K(v)$ for vertex v is *unable to provide* vertex degree of vertex v with respect to a relationship description ψ_{query}, if and only if $\exists k \in K(v)$, such that $(\psi \in k \not\leq_g \psi_{query}) \wedge \neg mutex(\psi, \psi_{query})$.

Example 4.38. Let $k \in K(v)$ be a cached degree in cache $K(v)$ for vertex v and $\psi \in k$ a relationship description in k, of the form $\psi = (IN, RATED, \{rating \rightarrow (\beta = 3), year \rightarrow ?\})$. Let $\psi_{query} = (IN, RATED, \{rating \rightarrow (\beta = 3), year \rightarrow 2013\})$ be a relationship description, intended to count all

incoming relationships of type RATED at vertex v with the rating property equal to 3, and the year property equal to 2013. Because ψ, however, could represent incoming RATED relationships with rating equal to 3 and year equal to *any value*, such a vertex degree cannot be determined. Note that $(\psi >_g \psi_{query}) \wedge \neg mutex(\psi, \psi_{query})$, thus $(\psi \not\leq_g \psi_{query}) \wedge \neg mutex(\psi, \psi_{query})$.

Example 4.39. Let $k \in K(v)$ be a cached degree in cache $K(v)$ for vertex v and $\psi \in k$ a relationship description in k, of the form $\psi = (IN, RATED, \{rating \rightarrow (\beta = 3), year \rightarrow ?\})$. Let $\psi_{query} = (IN, RATED, \{rating \rightarrow ?, year \rightarrow 2013\})$ be a relationship description intended for counting the vertex' degree. Again, such a vertex degree cannot be determined, this time because nothing can be said about the two descriptions' mutual order and because they are not mutually exclusive. To see why, consider an incoming relationship of type RATED with the rating property equal to 3, and the year property equal to 2013. Both descriptions would match such a relationship, thus they are not mutually exclusive. Formally, $(\psi \not\geq_g \psi_{query}) \wedge (\psi \not\leq_g \psi_{query}) \wedge \neg mutex(\psi, \psi_{query})$, thus $(\psi \not\leq_g \psi_{query}) \wedge \neg mutex(\psi, \psi_{query})$.

The example above reveals an interesting fact. Theorem 4.31 is no longer valid, when the cache has gone through a compaction operation. We replace Theorem 4.31 with a new theorem:

Theorem 4.40. *There can be more than 1 cached degree $k \in K(v)$ for an edge $e \in E(v)$, such that $matches(e, \psi \in k, v)$.*

Proof. We prove by construction. Let $e_1, e_2 \in E(v)$ be two edges at vertex v, such that $\psi_{maxs}(e_1, v) = (IN, RATED, \{rating \rightarrow (\beta = 2), year \rightarrow (\beta = 2013)\})$ and $\psi_{maxs}(e_2, v) = (IN, RATED, \{rating \rightarrow (\beta = 3), year \rightarrow (\beta = 2012)\})$. Then $\psi_1 = (IN, RATED, \{rating \rightarrow ?, year \rightarrow (\beta = 2013)\})$ and $\psi_2 = (IN, RATED, \{rating \rightarrow (\beta = 3), year \rightarrow ?\})$ are valid generalisations of $\psi_{maxs}(e_1, v)$ and $\psi_{maxs}(e_2, v)$, respectively. Note that $(\psi_1 \not\geq_g \psi_2) \wedge (\psi_2 \not\geq_g \psi_1)$, thus it is possible that for some $K(v)' = COMPACT(K(v))$, $\psi_1 \in k_1, \psi_2 \in k_2$, where $k_1 \in K(v)'$ and $k_2 \in K(v)'$. Let e_3 be another edge at v, such that $\delta(e, v) = IN$, $\lambda(e) = RATED$, and $\Pi(e) = (rating \rightarrow 3, year \rightarrow 2013)$. Then $matches(e_3, \psi_1, v) \wedge matches(e_3, \psi_2, v)$. □

This introduces a potential problem: which one of the two cached counts k_1 and k_2 from the above proof should be incremented when creating relationship e_3? Choosing to increment both would clearly be wrong, because a query for all incoming relationships of type RATED would count e_3 twice; once in k_1 and once in k_2.

To find a solution to this problem, we need to realise that since they both match edge e_3, they are not mutually exclusive, i.e., $\neg mutex(\psi_1, \psi_2)$ (from Definition 4.28). We also know that $(\psi_1 \not\geq_g \psi_2) \wedge (\psi_1 \not\leq_g \psi_2)$, because if we could say something about their relative generality, one of them would have been removed during compaction on line 19 of Algorithm 4.1.

Consequently, the only queries that the cache can answer are with respect to a relationship description ψ_{query}, such that $((\psi_1 \leq_g \psi_{query}) \vee mutex(\psi_1, \psi_{query})) \wedge ((\psi_2 \leq_g \psi_{query}) \vee mutex(\psi_2, \psi_{query}))$, because otherwise, $((\psi_1 \not\leq_g \psi_{query}) \wedge \neg mutex(\psi_1, \psi_{query})) \vee ((\psi_2 \not\leq_g \psi_{query}) \wedge \neg mutex(\psi_2, \psi_{query}))$, which means the query cannot be served (by Definition 4.37).

Now, if the relationship description in an answerable query ψ_{query} is more general or equal to the relationship description in one of the cached counts, then the query cannot be mutually exclusive with the relationship description of the other cached count, because they share a common descendant in the lattice. Formally, since $\neg mutex(\psi_1, \psi_2)$, then $(\psi_1 \leq_g \psi_{query}) \rightarrow \neg mutex(\psi_2, \psi_{query})$. However, in order for the query to be answerable, it must be true that $\psi_2 \leq_g \psi_{query}$. In such a case, the only thing that matters is the final sum of the two cached counts, since $(\psi_1 \leq_g \psi_{query}) \wedge (\psi_2 \leq_g \psi_{query})$.

If the relationship description in an answerable query is not more general or equal to any of the two cached counts, then it must be mutually exclusive with both of them. However, the result of such a query would not be influenced by any of the relationships cached with ψ_1 and ψ_2, by definition of mutual exclusivity. Therefore, the solution to the problem is incrementing either one of the cached counts (k_1 or k_2) by one.

4.5.2 Compaction Threshold

We now define a compaction threshold τ, which is simply an arbitrary constant specifying the maximum number of cached counts for each vertex. Its purpose is to be used to determine, whether a compaction process needs to take place at a vertex, and after it has taken place, whether it should be run again, because the number of cached counts is still higher than the threshold.

4.6 Property Change Frequency Function

Before we conclude the chapter, there is one remaining topic to be discussed, the `PCF` function in Algorithm 4.1, i.e., the function that determines best-to-worst ordering of generalisations in the generalisation superset Ψ_{sg} for cache $K(v)$.

It is clear from the discussion above that in order to bring the space complexity of the cache to $O(1)$, some queries need to be sacrificed in the sense that the cache will not be able to answer them. Based on the movie graph example used throughout this report, we assume that the properties with most frequently changing values are less important and should, thus, be sacrificed first.

It is fairly reasonable to assume that one will ask how popular a movie is more frequently than how

many ratings were exactly created at a specific point in time. We do, however, acknowledge the fact that for some use cases, the opposite could be true. Users might want to count ratings based on the time when they were created. In this case, a different approach would have to be investigated. Below, we propose what we believe might be a sensible default strategy for many real-life applications.

It is important to note that in real-life graphs, it will not usually be the case that for all property keys $\alpha \in A$, a property value will be defined for every relationship. Instead, different properties are defined for different types of relationships. For instance, it makes sense to define a `timestamp` property on a `RATED` relationship, but not so much on an `IS_OF_GENRE` relationship. Therefore, the strategy should treat properties with respect to relationship type, rather than as a global concern.

4.6.1 Specification by Example

We now specify the requirements of the `PCF` function by examples (inspired by Adzic [1]). In all of them, we assume the compaction threshold $\tau = 4$ and ask the question, which cached degree shall be generalised and how.

Example 4.41. Consider the following cached degrees in cache $K(v)$:

$$k_1 = ((IN, RATED, \{rating \to (\beta = 2), timestamp \to (\beta = 1234)\}), 1)$$
$$k_2 = ((IN, RATED, \{rating \to (\beta = 2), timestamp \to (\beta = 1354)\}), 1)$$
$$k_3 = ((IN, RATED, \{rating \to (\beta = 2), timestamp \to (\beta = 2889)\}), 1)$$
$$k_4 = ((IN, RATED, \{rating \to (\beta = 3), timestamp \to (\beta = 4544)\}), 1)$$
$$k_5 = ((IN, RATED, \{rating \to (\beta = 4), timestamp \to (\beta = 0890)\}), 1)$$

In this case, generalising `rating` would not result in any compaction, since each `timestamp` has a different value. Also, there is only one relationship with `rating` equal to 3, and one with `rating` equal to 4. Therefore, we shall generalise the `timestamp` on relationships with `rating` equal to 2. This is consistent with the idea of sacrificing attributes that change most frequently. Thus, we would like the cache $K('v)'$ to look as follows, after compaction:

$$k'_1 = ((IN, RATED, \{rating \to (\beta = 2), timestamp \to ?\}), 3)$$
$$k'_2 = ((IN, RATED, \{rating \to (\beta = 3), timestamp \to (\beta = 4544)\}), 1)$$
$$k'_3 = ((IN, RATED, \{rating \to (\beta = 4), timestamp \to (\beta = 0890)\}), 1)$$

It would also be possible to generalise both `rating` and `timestamp`, but that would mean losing more

information than absolutely necessary.

Example 4.42. Consider the following cached degrees in cache $K(v)$:

$k_1 = ((IN, RATED, \{rating \to (\beta = 2), year \to ?\}), 3)$
$k_2 = ((IN, RATED, \{rating \to (\beta = 3), year \to (\beta = 2011)\}), 1)$
$k_3 = ((IN, RATED, \{rating \to (\beta = 3), year \to (\beta = 2011)\}), 1)$
$k_4 = ((IN, RATED, \{rating \to (\beta = 4), year \to (\beta = 2011)\}), 1)$
$k_5 = ((IN, RATED, \{rating \to (\beta = 1), year \to (\beta = 2012)\}), 1)$

In this case, generalising `rating` for some year, for instance 2011, could make sense, potentially leading to the following state of cache $K('v)'$:

$k'_1 = ((IN, RATED, \{rating \to (\beta = 2), year \to ?\}), 3)$
$k'_2 = ((IN, RATED, \{rating \to ?, year \to (\beta = 2011)\}), 3)$
$k'_3 = ((IN, RATED, \{rating \to (\beta = 1), year \to (\beta = 2012)\}), 1)$

However, this is not so desirable. $K(v)$ was already unable to answer queries for vertex degrees constrained to a specific `year`, because on k_1, `year` was already generalised. $K(v)'$, furthermore, has even more limited query answering ability, because queries for vertex degrees with a specific `rating` cannot be answered either. Instead, it would be preferable to generalise out another `year` property, since it has been treated as less important than `rating` by a previous compaction, resulting in the state of k'_1. In other words, we would like to pretend that the 3 incoming relationships in $K(v)$ of type `RATED` with `rating` equal to 2 all had a different `year` value. Moreover, we pretend that this value was different from any other concrete `year` value that other cached relationship descriptions have. We then end up with 5 different `year` values (3x unknown, 1x 2011, 1x 2012), but only 4 different `rating` values. Thus, a more suitable state of $K(v)'$ after a compaction of $K(v)$ would be:

$k'_1 = ((IN, RATED, \{rating \to (\beta = 2), year \to ?\}), 3)$
$k'_2 = ((IN, RATED, \{rating \to (\beta = 3), year \to ?\}), 2)$
$k'_3 = ((IN, RATED, \{rating \to (\beta = 4), year \to (\beta = 2011)\}), 1)$
$k'_4 = ((IN, RATED, \{rating \to (\beta = 1), year \to (\beta = 2012)\}), 1)$

Example 4.43. Consider the following cached degrees in cache $K(v)$:

$k_1 = ((IN, RATED, \{rating \rightarrow (\beta = 2), timestamp \rightarrow (\beta = 1234)\}), 1)$
$k_2 = ((IN, RATED, \{rating \rightarrow (\beta = 2), timestamp \rightarrow (\beta = 1354)\}), 1)$
$k_3 = ((OUT, RATED, \{rating \rightarrow (\beta = 3), timestamp \rightarrow (\beta = 2889)\}), 1)$
$k_4 = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \rightarrow (\beta = 1), timestamp \rightarrow (\beta = 4544)\}), 1)$
$k_5 = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \rightarrow (\beta = 1), timestamp \rightarrow (\beta = 0890)\}), 1)$

In this case, `timestamp` is different on each relationship description. Observe that there are 3 relationship descriptions with type `RATED` and 2 with type `WOULD_LIKE_TO_SEE`. Thus, we have more evidence for relationships of type `RATED` that timestamps change with every new relationship. We shall, therefore, prefer the following state of cache $K(v)'$ after compaction:

$k'_1 = ((IN, RATED, \{rating \rightarrow (\beta = 2), timestamp \rightarrow ?\}), 2)$
$k'_2 = ((OUT, RATED, \{rating \rightarrow (\beta = 3), timestamp \rightarrow (\beta = 2889)\}), 1)$
$k'_3 = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \rightarrow (\beta = 1), timestamp \rightarrow (\beta = 4544)\}), 1)$
$k'_4 = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \rightarrow (\beta = 1), timestamp \rightarrow (\beta = 0890)\}), 1)$

Based on these examples, we now come up with the notion of *informational value*, which is a measure of how useful a cached relationship description is, with the intention to use this measure for ordering candidate generalisation for achieving the smallest loss in informational value during compaction.

4.6.2 Informational Value

Definition 4.44. Let $\rho(K(v), \lambda, \alpha)$ be the total number of distinct values of property with key $\alpha \in A$ across all relationship descriptions $\psi(\delta_\psi, \lambda_\psi, \phi)$ with $\lambda_\psi = \lambda$, where $k(\psi, n) \in K(v)$. Each $\phi \in \psi$, such that $\phi(\alpha) \neq ?$ adds 1 to this number. If $\phi(\alpha) = ?$, then the number ρ is incremented by $n \in \psi$, in line with the decision to pretend that every relationship responsible for the cached degree with relationship description ψ had a different value assigned to property key α. This will ensure that properties already assigned a to a wildcard (?) constraint will always be generalised in preference to properties only constrained to concrete values.

Example 4.45. Let $K(v)$ contain the following cached degrees:

$k_1 = ((IN, RATED, \{rating \rightarrow (\beta = 2), timestamp \rightarrow ?\}), 2)$
$k_2 = ((IN, RATED, \{rating \rightarrow (\beta = 3), timestamp \rightarrow (\beta = 2889)\}), 1)$

$k_3 = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \to (\beta = 1), timestamp \to (\beta = 4544)\}), 1)$

Then $\rho(K(v), RATED, timestamp) = 3$, because k_1 contains (up to) 2 distinct values, and k_2 contains 1. k_3 is not taken into account, because it is of a different relationship type.

Definition 4.46. Let $\rho_{avg}(K(v), \lambda, \alpha)$ be the average number of distinct values of property with key $\alpha \in A$ across all relationship descriptions $\psi(\delta_\psi, \lambda_\psi, \phi)$ with $\lambda_\psi = \lambda$, where $k(\psi, n) \in K(v)$. In plain language, it is the number of distinct values of a property, per relationship of a given type. It is computed as $\rho_{avg}(K(v), \lambda, \alpha) = \frac{\rho(K(v), \lambda, \alpha)}{n_\lambda + 1}$. n_λ is the total number of relationships at vertex v with type λ, i.e., $n_\lambda = \sum_{\psi(\delta_\psi, \lambda_\psi, \phi) \in k, s.t. (k \in K(v) \wedge \lambda_\psi = \lambda)} n \in \psi$. Note that $\rho_{avg} < 1$ at all times, because the number of distinct values of a property for a specific relationship type cannot be greater than the number of relationships of that type. The larger the value of ρ_{avg}, the more frequently the property changes its value on relationships of given type.

Example 4.47. Let $K(v)$ contain the following cached degrees:

$k_1 = ((IN, RATED, \{rating \to (\beta = 2), timestamp \to ?), 2)$
$k_2 = ((IN, RATED, \{rating \to (\beta = 3), timestamp \to (\beta = 2889)\}), 1)$
$k_3 = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \to (\beta = 1), timestamp \to (\beta = 4544)\}), 1)$

Then $\rho_{avg}(K(v), RATED, timestamp) = \frac{3}{4}$, because $\rho(K(v), RATED, timestamp) = 3$ and the number of relationships with $\lambda = RATED$ is also 3, i.e., $n_{RATED} = 3$. Thus, $\rho_{avg}(K(v), RATED, timestamp) = \frac{\rho_{avg}(K(v), RATED, timestamp)}{n_{RATED} + 1} = \frac{3}{4}$.

On the other hand, there are only 2 distinct values for the `rating` property on `RATED` relationships, thus $\rho_{avg}(K(v), RATED, rating) = \frac{\rho_{avg}(K(v), RATED, rating)}{n_{RATED} + 1} = \frac{2}{4} = \frac{1}{2}$.

In other words, since $\frac{3}{4} > \frac{1}{2}$, the `timestamp` property, on average, changes its value more frequently than the `rating` property on `RATED` relationships.

Note the $+1$ in the denominator of ρ_{avg} calculation. This is to make sure that in the case of a potential tie, ρ_{avg} for more frequently occurring relationships is slightly larger than for less frequently occurring ones. The rationale is that we have more evidence for such an average number of distinct values and, thus, want to claim that the more frequently occurring relationships change the property value more frequently, as required by the specification.

For instance, consider $n_{\lambda 1} = 10$ relationships with $\rho_1 = 5$ distinct values of a property, and $n_{\lambda 2} = 20$ different relationships with $\rho_2 = 10$ distinct values of the same property. Then $\rho_{avg1} = \frac{5}{10+1}$ and

$\rho_{avg2} = \frac{10}{20+1}$. Although the property changes with the same frequency in both cases (i.e., a potential tie), $\rho_{avg1} < \rho_{avg2}$, thus we consider the property to be changing more frequently on relationships of type λ_2. As we'll see later, this results in relationships of type λ_2 being treated as better candidates for compacting out some properties, whist relationships of type λ_1 will get another chance to "prove" their property changes less frequently.

Definition 4.48. Let $\iota(K(v), \psi, \alpha)$ be the *informational value* of a property with key $\alpha \in A$ on a relationship description $\psi(\delta_\psi, \lambda_\psi, \phi)$ with respect to cache $K(v)$. It is defined as $\iota(K(v), \psi, \alpha) = 1$, if and only if $\psi(\alpha) \neq ?$, and $\iota(K(v), \psi, \alpha) = \rho_{avg}(K(v), \lambda_\psi, \alpha)$ otherwise.

Example 4.49. Let $K(v)$ contain the following cached degrees:

$k_1 = (\psi_1, n_1) = ((IN, RATED, \{rating \rightarrow (\beta = 2), timestamp \rightarrow ?), 2)$
$k_2 = (\psi_2, n_2) = ((IN, RATED, \{rating \rightarrow (\beta = 3), timestamp \rightarrow (\beta = 2889)\}), 1)$
$k_3 = (\psi_3, n_3) = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \rightarrow (\beta = 1), timestamp \rightarrow (\beta = 4544)\}), 1)$

Then,
$\iota(K(v), \psi_1, rating) = 1$, because `rating` is not mapped to ?. On the other hand,
$\iota(K(v), \psi_1, timestamp) = \rho_{avg}(K(v), RATED, timestamp) = \frac{3}{4}$ from previous example.

Intuitively, properties descriptions constraining properties to a single, concrete value (including *UNDEF*) carry more informational value than those placing no constraints on the property. The maximum informational value is 1 and it is only achieved for properties with concrete constraints, by definition. All others have informational value less than 1, by definition of ρ_{avg}. This means concrete values are always preferred to wildcards.

Definition 4.50. Let $\iota(K(v), \psi)$ be the *informational value* of a relationship description $\psi(\delta_\psi, \lambda_\psi, \phi)$ with respect to cache $K(v)$. It is defined as $\iota(K(v), \psi) = \frac{1}{|A(\lambda_\psi)|+1} \sum_{\alpha \in A(\lambda_\psi)} \iota(K(v), \psi, \alpha)$, where $A(\lambda_\psi)$ is the set of all attributes defined on relationship descriptions with type equal to λ_ψ in $K(v)$.

Example 4.51. Let $K(v)$ contain the following cached degrees:

$k_1 = (\psi_1, n_1) = ((IN, RATED, \{rating \rightarrow (\beta = 2), timestamp \rightarrow ?), 2)$
$k_2 = (\psi_2, n_2) = ((IN, RATED, \{rating \rightarrow (\beta = 3), timestamp \rightarrow (\beta = 2889)\}), 1)$
$k_3 = (\psi_3, n_3) = ((IN, WOULD\_LIKE\_TO\_SEE, \{level \rightarrow (\beta = 1), timestamp \rightarrow (\beta = 4544)\}), 1)$

Then,
$\iota(K(v), \psi_1) = \frac{1}{|A(RATED)|+1} \sum_{\alpha \in A(RATED)} \iota(K(v), \psi, \alpha) = \frac{1}{3}(\iota(K(v), \psi_1, rating) + \iota(K(v), \psi_1, timestamp)) =$
$\frac{1}{3}(1 + \frac{3}{4}) = \frac{7}{12}$

This time, the $+1$ in the denominator of the fraction is to prevent division by zero in the pathological case where $A(\lambda_\psi) = \emptyset$, thus $|A(\lambda_\psi)| = 0$.

Since every wildcard assigned to a property constraint decreases the value of ι more than a concrete constraint, more specific descriptions will have a higher informational value than more general ones. Thus, they will be considered "better" generalisations, as specified.

4.6.3 Sorting

During the cache compaction operation, generalisations can now be sorted according to their informational value, such that the ones with the highest informational value are the best ones and get used first, whilst the ones with the lowest informational value are the last, keeping the information loss as small as possible.

Definition 4.52. Property Change Frequency function `PCF` produces a sorted set $\Psi'_{sg} = PCF(\Psi_{sg}, K(v))$ from a generalisation superset Ψ_{sg} for cache $K(v)$, such that the elements of $\psi \in \Psi'_{sg}$ are sorted by decreasing value of $\iota(K(v), \psi)$.

Chapter 5

Implementation

In this chapter, we present the design and implementation of the software that puts the theoretical ideas, introduced thus far, into practice. We design a software module called *Relationship Count Module* that implements the theoretical vertex degree cache from previous chapter. Before that, however, we introduce a framework, called *GraphAware*, which supports this module as well as future modules with similar purposes.

Design vs. Implementation The practical outcome of this project will mainly be used by software developers. For this reason, it is occasionally helpful to show snippets of code, although that practice is kept to a minimum. It is also one of the reasons why design and implementation are presented together; another reason is the fact that the software has been developed in a number of design-test-implement-refine[1] iterations. It would thus be challenging to present some of the design considerations without going into some level of implementation detail.

Tools and Techniques Java 7 and Neo4j 1.9.2 are used in the implementation described below. Maven 3.0.3 is used as a dependency management and build tool. JUnit 4.10 and Mockito 1.9.0 are used for testing, and Log4j 1.2.17 for logging. The only requirement for compiling and running the code is Java 7 and Maven; all other dependencies get downloaded automatically. Detailed instructions are provided in Appendices B and C. Object-oriented design best-practices are used throughout, unless a deviation from them is beneficial for performance or maintainability of the code. Where these deviations occur, they are explained in the code itself as comments; when relevant, they are mentioned in this report as well.

[1] the practice of writing automated tests against designed interfaces before actually implementing the logic is sometimes called test-driven development (TDD); see, for instance, Beck [3]

Neo4j Core API Before we begin, it is useful to introduce the most relevant part of the Neo4j Core API, depicted in Figure 5.1. It is a low-level imperative API that provides graph mutation and traversal capabilities. The classes and methods have fairly self-explanatory names. The database itself is represented by the `GraphDatabaseService` interface and provides node creation and retrieval capabilities, along with a way to start a transaction. Nodes and relationships, both implementing the `PropertyContainer` interface, are represented as `Node` and `Relationship`, respectively. The `Node` class provides access to relationships at that node, whilst `Relationship` provides access to its type and both participating nodes. Both `Node` and `Relationship` allow property retrieval, mutation, and deletion, as well as deletion of themselves. In all UML diagrams presented in this chapter, Neo4j classes are green, GraphAware Framework classes are blue, and Relationship Count Module classes are pink.

5.1 GraphAware Framework

Caching on Node Properties The purpose of the Relationship Count Module is to implement a cache for vertex degrees, also known as relationship counts. The first design consideration is where to store the cached information. The theoretical cache developed in previous chapter imposes no limitations in this respect. However, storing the information outside of Neo4j and keeping it up to date would require the presence of another transactional resource and some sort of coordination between Neo4j's transaction manager and the other resource's transaction manager. Whilst well-known algorithms and protocols exist for such a coordination, it would introduce unnecessary overhead. Since Neo4j itself is a transactional database, it is possible to store cached relationship counts in the graph itself. Because the counts should be cached for each node and because they are essentially key-value pairs, node properties seem to be an ideal place for this caching.

Metadata in the Graph Generalising this idea, it is not difficult to envisage other useful, perhaps analytical, modules that would benefit from writing extra information into the graph. There is no reason such "metadata" should be limited to node properties; they could take the form of relationship properties, but also entire subgraphs. The framework, for the moment acting only as a library of useful code, provides a set of interfaces and abstract base classes to support such a development.

5.1.1 Representations

Graph Concept Representations The first presented group of classes and interfaces enables development of custom detached[2] representations of nodes, relationships, and properties with additional

[2] i.e., not referencing the database or any other Neo4j classes

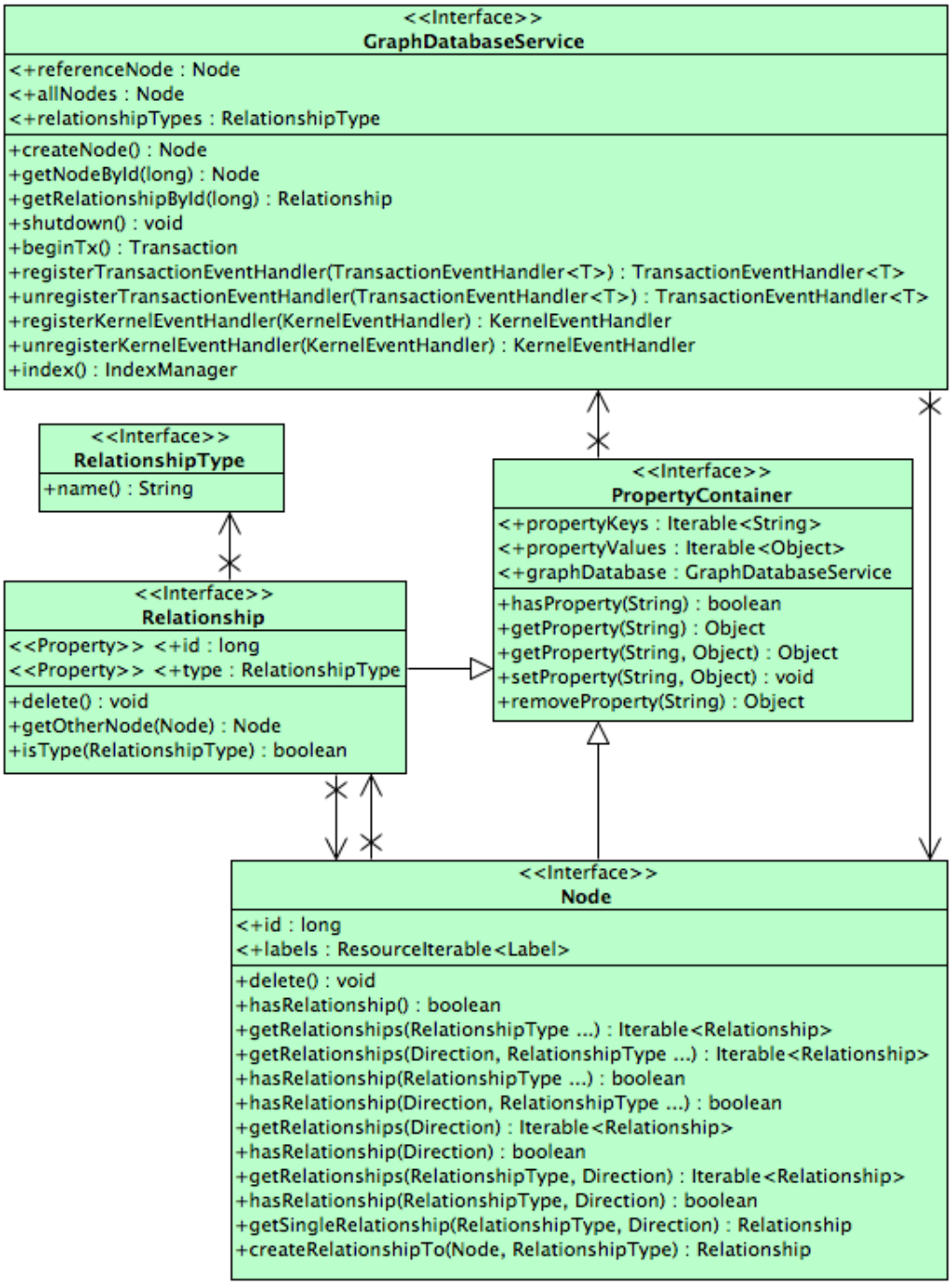

Figure 5.1: Neo4j Core API (with a few irrelevant methods omitted)

capabilities. For instance, a module might need a relationship, which can convert itself to a string representation. Another module, to give a different example, might want to use a relationship description aware of the general-to-specific ordering described in previous chapter. Both of these examples are, in fact, requirements for the Relationship Count Module, as we shall see later. One more example might be the need to serialise a node with all its properties and send it across the wire.

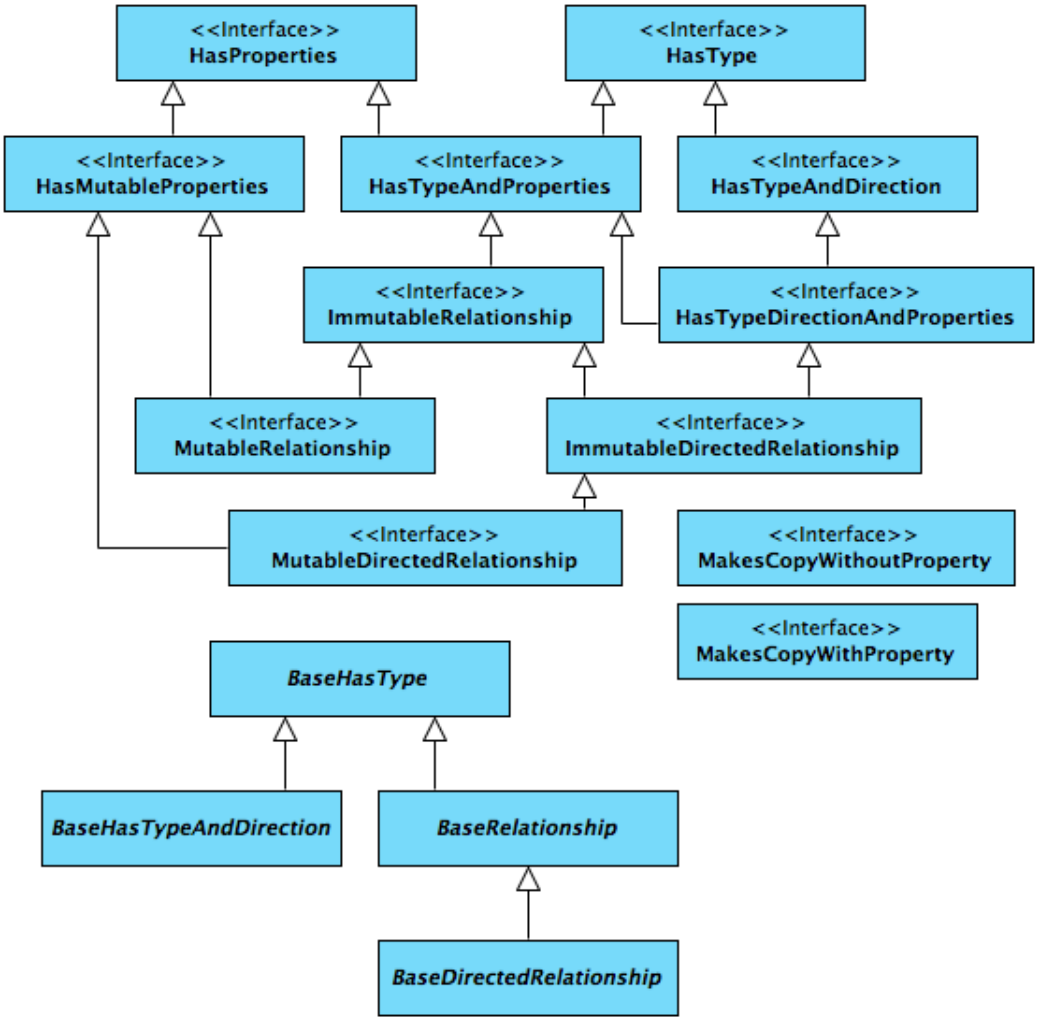

Figure 5.2: Interfaces and Abstract Classes for Relationship Representations

For these purposes, a hierarchy of extendible representations of nodes, relationships, and properties is provided. As an example, Figure 5.2 presents the hierarchy for relationship representations. Similar hierarchies exist for nodes and properties representations. Implementations and extensions of these classes simply maintain a copy of the information from the graph and optionally provide additional functionality. Four types of basic implementations of these interfaces, each intended for different use cases, are also included in the framework:

- Immutable representations that store properties as objects
- Mutable representations that store properties as objects

- Immutable representations that store properties as strings
- Mutable representations that store properties as strings

The use cases for some of these implementations are not defined here, because they will be determined by their future users, i.e., developers of additional GraphAware modules. As an indication, however, immutable representations are useful in multi-threaded environments where thread-safety needs to be guaranteed, while mutable ones are useful where the overhead of making immutable copies upon modification is not acceptable. Representations storing property values with their original Java types might be useful when these types matter, whilst representations with string property values are useful for string-convertible representations, as we shall see in the next paragraph.

String Convertible Representations The Relationship Count Module caches vertex degrees as key-value pairs on nodes, where key is the relationship description and value is the number of relationships with such a description at a node. Because property keys in Neo4j are strings, relationship and, indeed, properties descriptions need to be convertible, or serialisable, to a string. Since string-converted relationship descriptions represent data already present elsewhere in the graph, only one-way conversion is needed, i.e., from the real graph concept (node, relationship, property) to a string. Again, because other modules will potentially write additional metadata into the graph, string-convertible representations of nodes, relationships, and properties are provided by the framework.

String Convertible Relationships Chapter 4 shows that the relationship descriptions that are used in cached degrees and, therefore, need to be string-convertible, are the most specific descriptions of relationships, with all properties constrained to a concrete value, or generalisations thereof, where some properties might be unconstrained (i.e., mapped to the ? predicate). Unlike in a theoretical property graph, where the knowledge of all property keys $\alpha \in A$ was assumed at all times, a property container in a real graph only knows about properties actually defined on itself. For example, a `RATED` relationship with a single `rating` property knows nothing about properties of other relationships; thus, it does not know the set A of all property keys used in the graph. Based on these observations, string representations of relationship descriptions are designed to be of the form `_GA_MID_`$\lambda\#\delta\#\alpha_1\#\beta_1\#\alpha_2\#\beta_2\#\ldots$, where:

`_GA_` is a prefix indicating that this is GraphAware metadata
`MID` is a prefix indicating, which GraphAware module the metadata belongs to (module ID)
λ is a relationship type,
δ is a relationship direction,

α_x are property keys,

β_x are property constraints.

If a property α is constrained to a concrete value β, such that $\beta \neq UNDEF$, then β_x is this value β converted to a string. If, on the other hand, a property α is unconstrained, β_x takes on a special "wildcard" value `_GA_*`, which represents the ? predicate. Finally, when a property α is constrained to $\beta = UNDEF$, then it is not present in the string representation at all. In other words, $\beta = UNDEF$ is the default property constraint on string-convertible relationship descriptions, unless explicitly defined otherwise.

Example 5.1. Consider a relationship description $\psi = (IN, RATED, \{rating \rightarrow (\beta = 2), timestamp \rightarrow ?, role \rightarrow (\beta = UNDEF)\}$, written by module M1. The string representation of ψ would be `_GA_M1_RATED#INCOMING#rating#2#timestamp#_GA_*#`.

The `_GA_` prefix is used to distinguish metadata written by GraphAware from data written by the database user. Similarly, the `MID` (module identifier) prefix identifies the module this data belongs to, which prevents modules from interfering with each other's metadata.

Converting β values to strings is achieved using standard Java `toString()` methods for all primitive types[3], while arrays of primitives are converted to `[element1, element2, ...]`. Information about the original type of the value is thus lost during this conversion, which would be a problem, for example, if the value `2` (`int`) for a property on a relationship had a different meaning from value `"2"` (`String`) for the same property on a relationship of the same type. However, such a design would hardly be considered a good practice and could easily be avoided, rendering this limitation only theoretical.

In the string conversion presented above, the hash character (#) is used as information delimiter. This could cause a problem when some of the delimited information contains this character. For that reason, the framework can be configured to use a different separator of information from the default #.

Performance testing and profiling has shown that string conversions can take a relatively long time. As an optimisation, string representations of immutable node, relationship and property descriptions are only computed once for each instance, upon the first request for such a conversion (lazily), then cached on the object itself. Subsequent requests are then served from this cache.

5.1.2 Transaction Event API

Another clear requirement for the Relationship Count Module, as well as any other module that wishes to store metadata about the graph and keep them up to date, is the opportunity to react to every

[3] using their Object equivalents, e.g. `Integer.toString()` for `int` values

graph mutation before the transaction actually commits, so that it can alter this mutation, possibly by adding/updating metadata, or even preventing the transaction from committing altogether.

Neo4j Transaction Event API Neo4j provides a mechanism supporting such a requirement. Implementations of Neo4j's `TransactionEventHandler` interface can register with the database and receive a callback before data is about to be committed, after data has been committed, and after a transaction has been rolled back, for whatever reason. Information about mutations that are about to occur / have just occurred are encapsulated in an instance of `TransactionData`. This interface allows users to find out, which property containers have been created and deleted in the transaction, and which properties have been assigned to and removed from property containers. Figure 5.3 depicts this API.

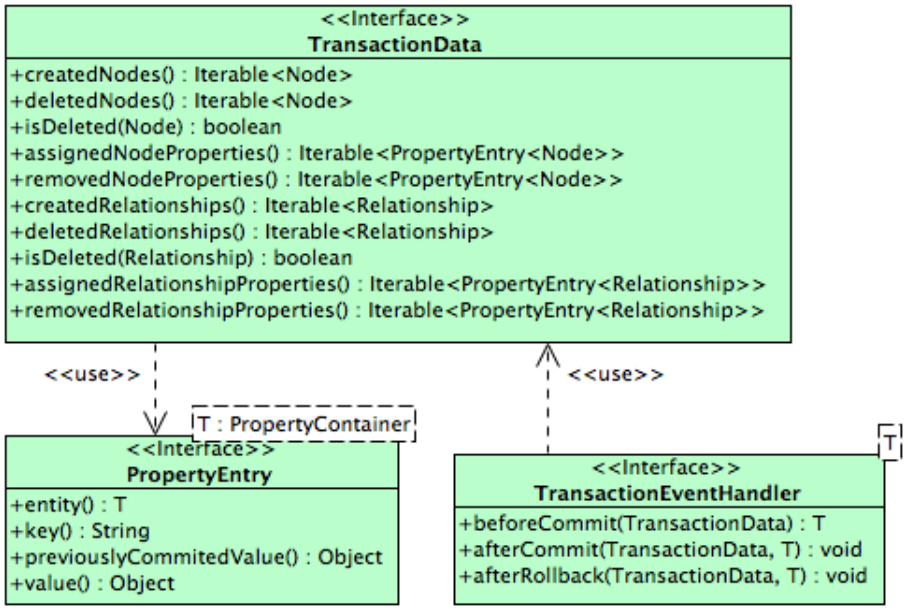

Figure 5.3: Neo4j Transaction Event Handler API

However, there is no straightforward way to find out, which property containers have been changed and how. One would have to iterate through assigned and removed properties for all property containers and organise them by property container ID to find that information. In fact, even inquiring about properties of a single deleted property container requires the developer to iterate through the removed properties of all property containers and pick out the right ones. Put in another way, getting information about a created relationship's properties can be easily done using methods provided for that purpose on the `Relationship` interface, whilst trying to do the same thing for a deleted relationship results in exceptions being thrown by Neo4j.

Consequently, before a transaction commits, it is impossible to traverse the graph using standard Neo4j APIs without taking the risk of encountering a property container deleted in the transaction and getting an exception, thus rolling back the entire transaction. It is, however, reasonable to expect that (analytical) modules will need to traverse the graph, in order to find out what metadata to create/update.

Finally, even if module developers took care and traversed the graph differently in these scenarios, checking `TransactionData` to make sure a deleted property container has not been encountered, each module would have to do this independently.

Besides increased chance of introducing bugs, this is wasteful, since all modules would essentially perform the same computation on the same instance of `TransactionData`, trying to find:

- created property containers and their properties
- deleted property containers and their properties
- changed property containers and which properties have been changed and how

Improved Transaction Event API GraphAware provides an improved API for transaction event handling, shown in Figure 5.4. This API allows direct access to all the created, deleted, and changed property containers along with the information about the exact extent of the changes. These are encapsulated in the `Change` object, which provides methods for accessing the old version of the changed object, as well as the new one. Unlike in the original Neo4j API, properties of deleted property containers can be directly accessed through the `Node` and `Relationship` objects, as one normally does when working with Neo4j. All this has been made possible thanks to two design decisions discussed in turn.

Lazy Transaction Event API The first is the introduction of a “lazy" implementation of `ImprovedTransactionData`, called `LazyTransactionData`, and related class hierarchy, depicted in Figure 5.5. `LazyTransactionData` uses `LazyNodeTransactionData` and `LazyRelationshipTransactionData` to provide data about nodes and relationships, respectively. These classes are decorators of Neo4j's `TransactionData` and contain a number of data structures, which support answering the desired questions efficiently, without the need to interrogate `TransactionData` over and over again. These data structures, shown in Listing 5.1[4], are built lazily, meaning that they are populated the first time information is requested. Consequently, `TransactionData` is not queried for the same information more than once. Moreover, if some data is never requested, `TransactionData` is not queried for that data at all, resulting in no additional overhead.

Property Container Wrappers The `ImprovedTransactionData` API and its lazy implementation allow easier and more efficient querying of graph mutations. However, it does not solve the problem of exceptions being thrown when properties of deleted property containers are accessed. In order to

[4]These are the same for nodes and relationships, thus reside in `LazyPropertyContainerTransactionData`. T is the type of `PropertyContainer`.

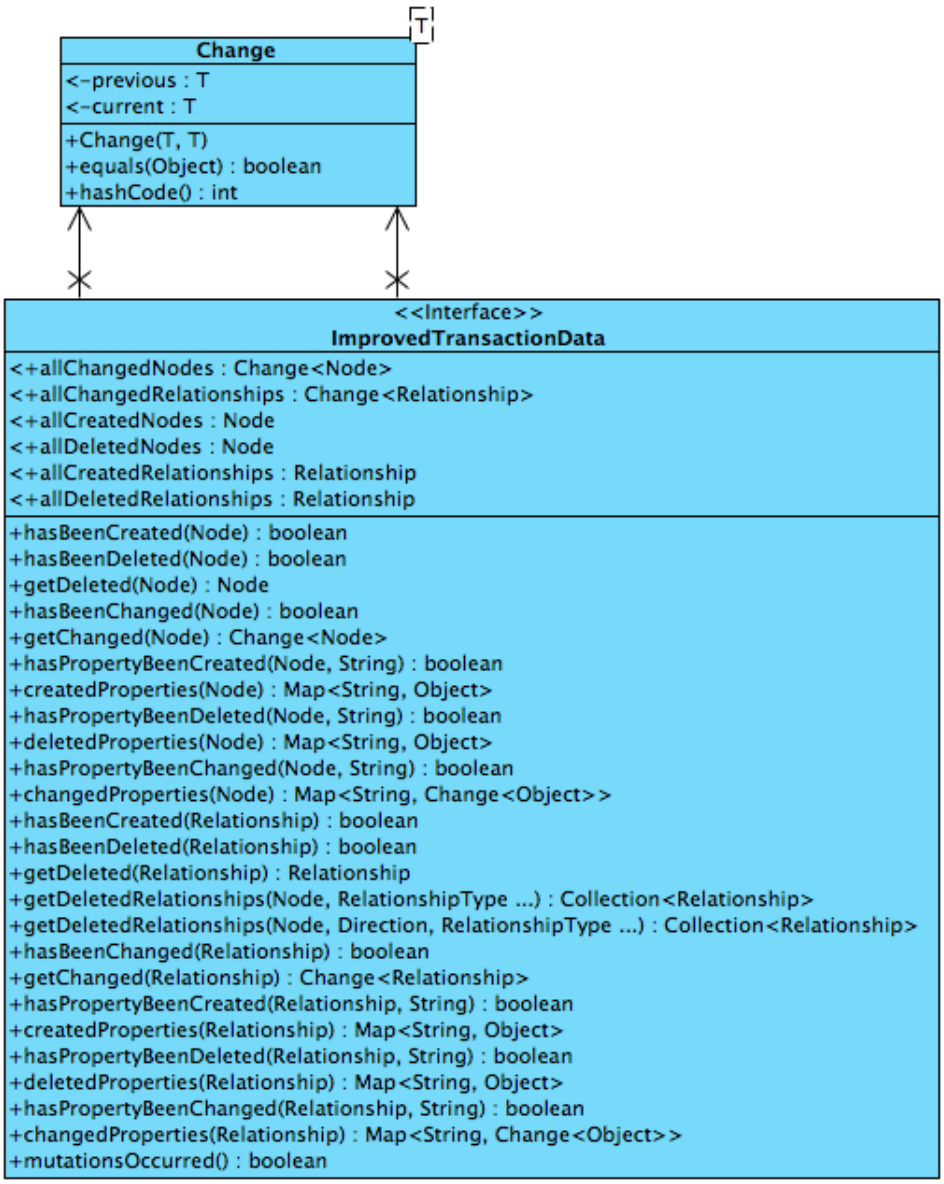

Figure 5.4: GraphAware Improved Transaction Event API

```
private Map<Long, T> created;
private Map<Long, T> deleted;
private Map<Long, Change<T>> changed;

/** <ID, <key, new value>> */
private Map<Long, Map<String, Object>> createdProperties;

/** <ID, <key, old value>> */
private Map<Long, Map<String, Object>> deletedProperties;

/** <ID, <key, old and new value>> */
private Map<Long, Map<String, Change<Object>>> changedProperties;

/** <ID, <key, old value>> of properties of deleted prop. containers */
private Map<Long, Map<String, Object>> deletedContainersProperties;
```

Listing 5.1: LazyPropertyContainerTransactionData Data Structures

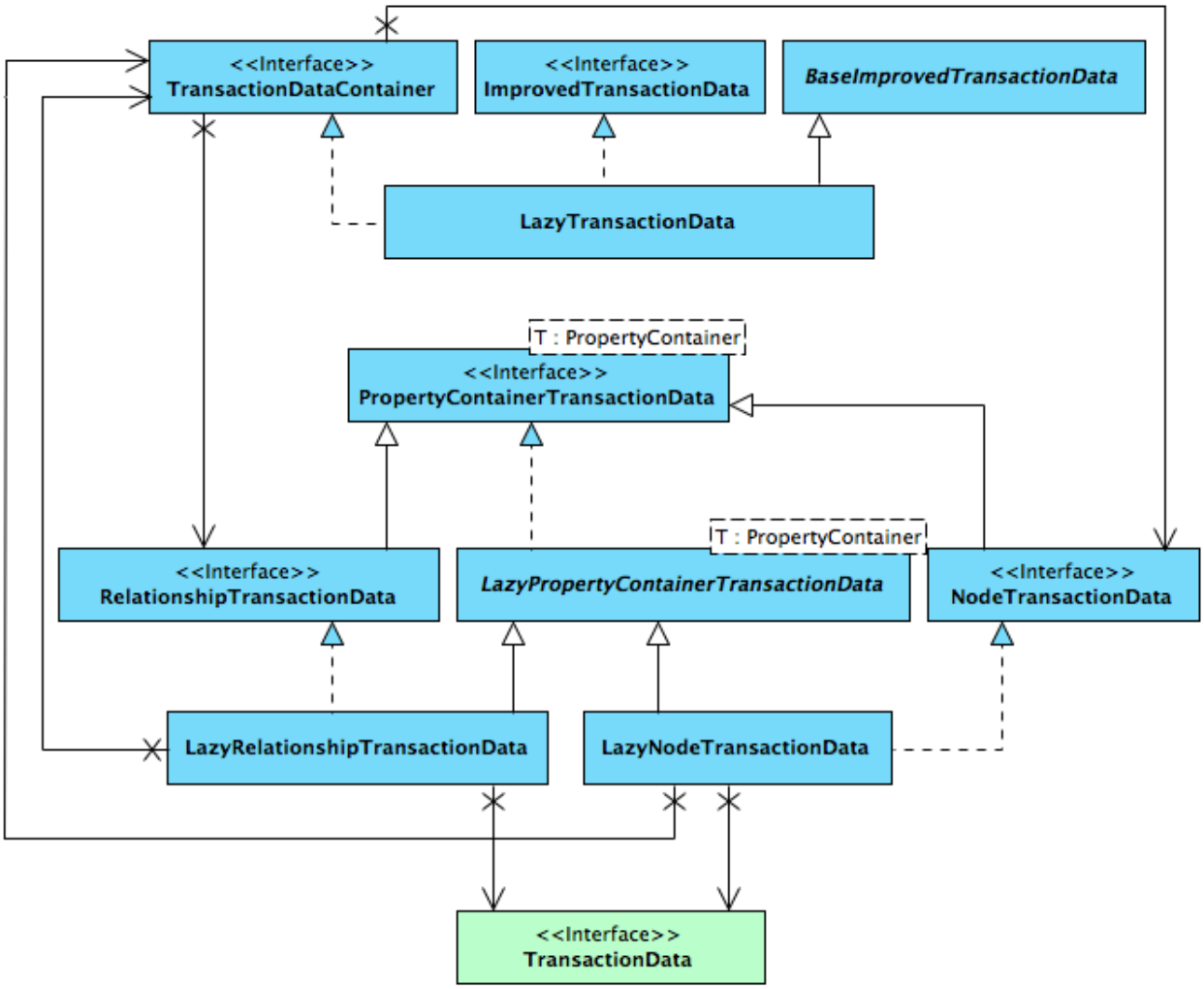

Figure 5.5: Lazy Implementation of Improved Transaction Event API

allow regular graph traversals in the context of transaction event handlers, decorators (also known as wrappers) [14] of Neo4j `Node` and `Relationship` objects have been designed and implemented, enabling these traversals. Before they are introduced, however, let us claim that `Node` and `Relationship` decorators have much wider applicability than this use case.

Consider, for example, a module (let us call it Temporal Graph Database for the moment), which prevents nodes, relationships, and properties from being deleted altogether and marks them as obsolete, perhaps with a timestamp, instead. Such a module would benefit from the use of property container decorators, re-implementing the methods that delete information from the graph, whilst delegating all other methods to the decorated object. Since the utility of property container decorators is not limited to the use case discussed in the previous paragraph, the framework provides generic support for building these decorators, in the form of the class hierarchy presented in Figure 5.6.

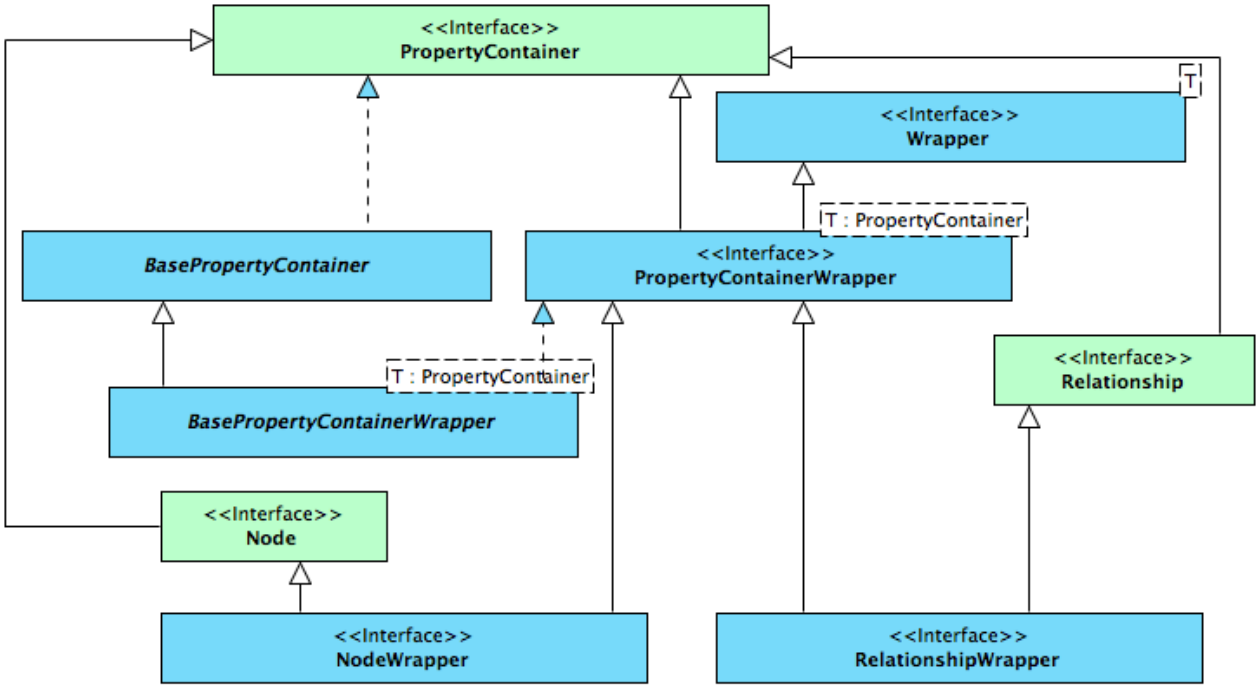

Figure 5.6: Base Classes and Interfaces for GraphAware PropertyContainer Decorators

Graph Snapshotting The first implementation that uses this support for decorators solves the issue presented above and introduces the ability to traverse the graph normally when handling transaction events, without the need to explicitly check for deleted property containers. It introduces a `Node` implementation, called `NodeSnapshot` and a `Relationship` implementation, called `RelationshipSnapshot`, both decorators representing the state of a node/relationship before the currently handled transaction started. These snapshots are aware of `NodeTransactionData` and `RelationshipTransactionData`, provided to them encapsulated in an instance of `TransactionDataContainer`, and consult the data during traversals, effectively filtering out changes that occurred in the transaction. This design is presented in Figure 5.7.

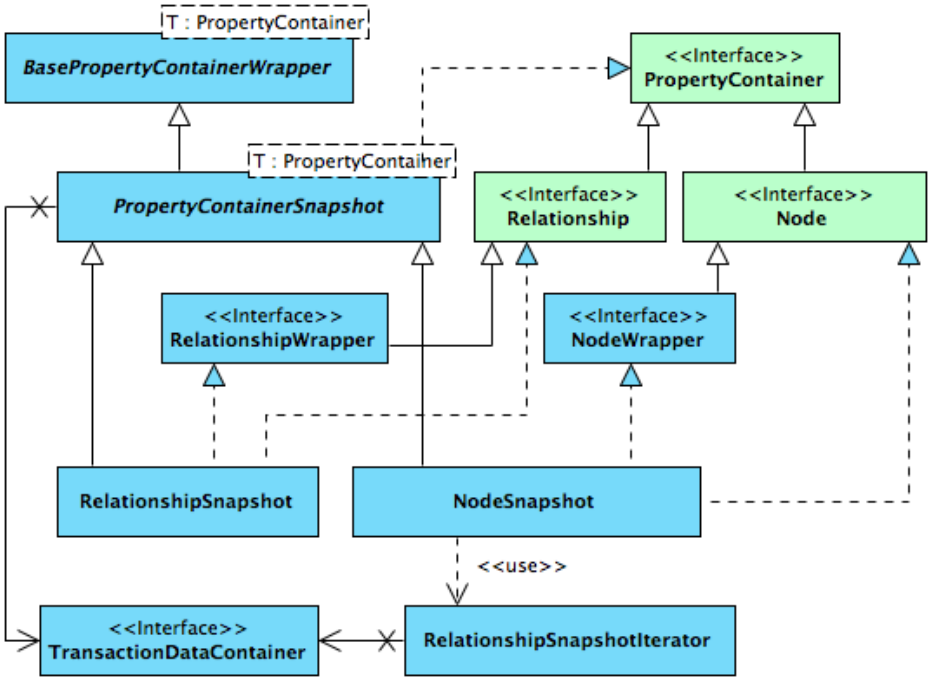

Figure 5.7: Node and Relationship Snapshots

Lazy Transaction Data Revisited This snapshotting functionality is actually used by `LazyTransactionData`. If, for instance, a relationship has been changed in a transaction, `LazyTransactionData` returns two versions of this changed relationship, wrapped in the `Change` object: the old version (as it was before the transaction started) as an instance of `RelationshipSnapshot` and a new instance (as it will be when the transaction commits) as Neo4j's internal relationship representation. Since both of these instances implement the `Relationship` interface, the old and the new version of the graph can be traversed in the exact same way, depending on which version of the relationship the traversal starts at. In case it starts with the old snapshot, every node and relationship produced by the traversal is a snapshot wrapper. Thus, if desired, the entire graph can be traversed this way, looking as if the transaction never happened. This happens completely transparently to the developer.

To reiterate, starting the traversal from an old version of a changed property container or a deleted property container traverses the old version of the graph, while starting from a new version of a changed property container or a newly created one, traverses the new version. This is very convenient for future module developers, because it happens transparently and the two versions of the graph can be seamlessly compared. Since the same instance of `LazyTransactionData` is used everywhere, it is only populated once (if ever), eliminating wasteful CPU cycles.

Filtered Transaction Data It is possible that users of `ImprovedTransactionData` might not be interested in all nodes, relationships, and properties. For example, the Relationship Count Module is not interested in any mutations to nodes at all. Or, for instance, some users could only wish to use the

module for counting relationships of a certain type. In such cases, it would be wasteful to keep track of changes that the user does not care about. For that reason, a set of classes, following the strategy design pattern [14], has been designed for expressing interest in certain kinds of information. These are shown in Figure 5.8. `InclusionStrategies` encapsulate all strategies for all the property graph concepts: nodes, relationships, node properties, and relationship properties. It is up to module developers and programmers using these modules to implement these strategy interfaces to suit particular business requirements of the application that uses Neo4j and the framework. The implementations can choose to base their decisions on relationship types, property presence, or any other characteristic of the object a decision is being made about. Trivial implementations are also provided by the framework, i.e., implementations that include all nodes, relationships, and properties, and implementations that include no nodes, relationships, or properties.

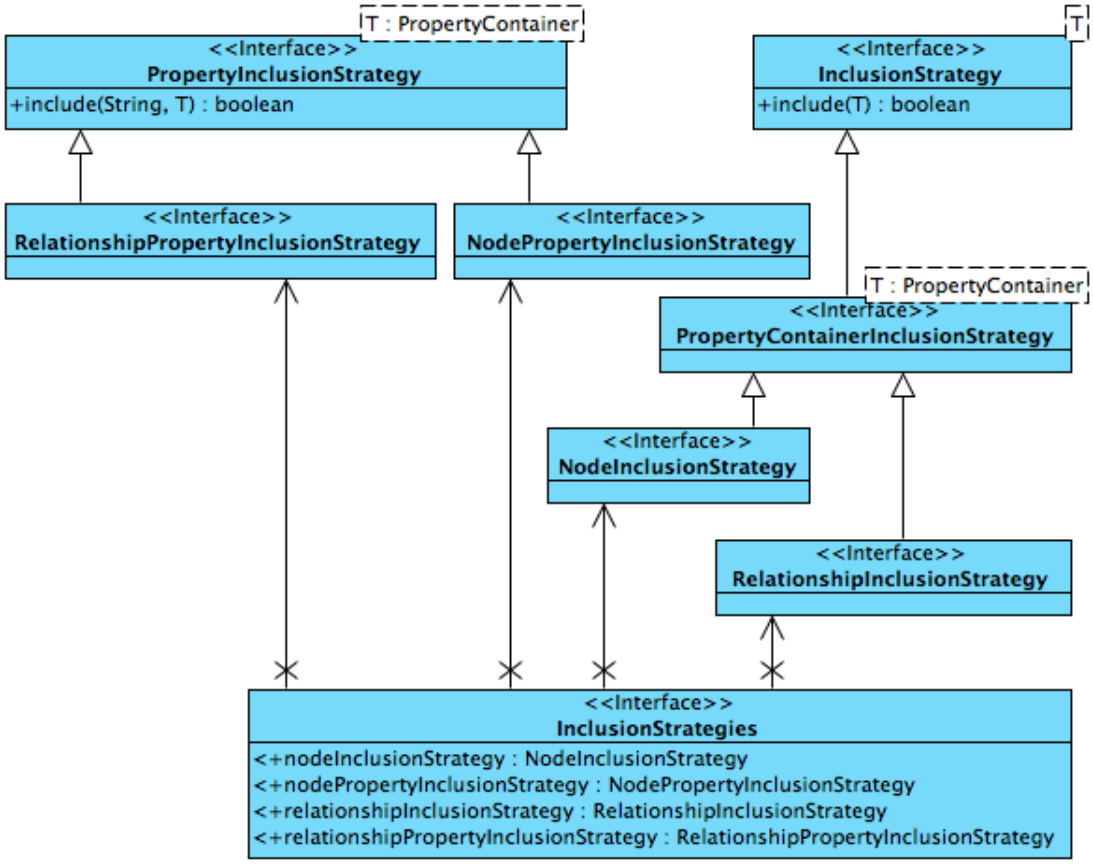

Figure 5.8: Strategies for Including Information

These strategies are used for filtering out unnecessary information when querying `ImprovedTransactionData` and when traversing the graph during transaction event handling. Two sets of decorator classes enable this filtering. Both of them consult these strategies and only return objects, for which the `include` method returns true. Figure 5.9 shows `FilteredTransactionData`, a decorator of `ImprovedTransactionData`, which is capable of filtering the data. Figure 5.10 shows `FilteredNode` and `FilteredRelationship`, decorators of any `Node` or `Relationship`, respectively, which effectively hide portions of the graph when used for traversals, according to the provided strategies.

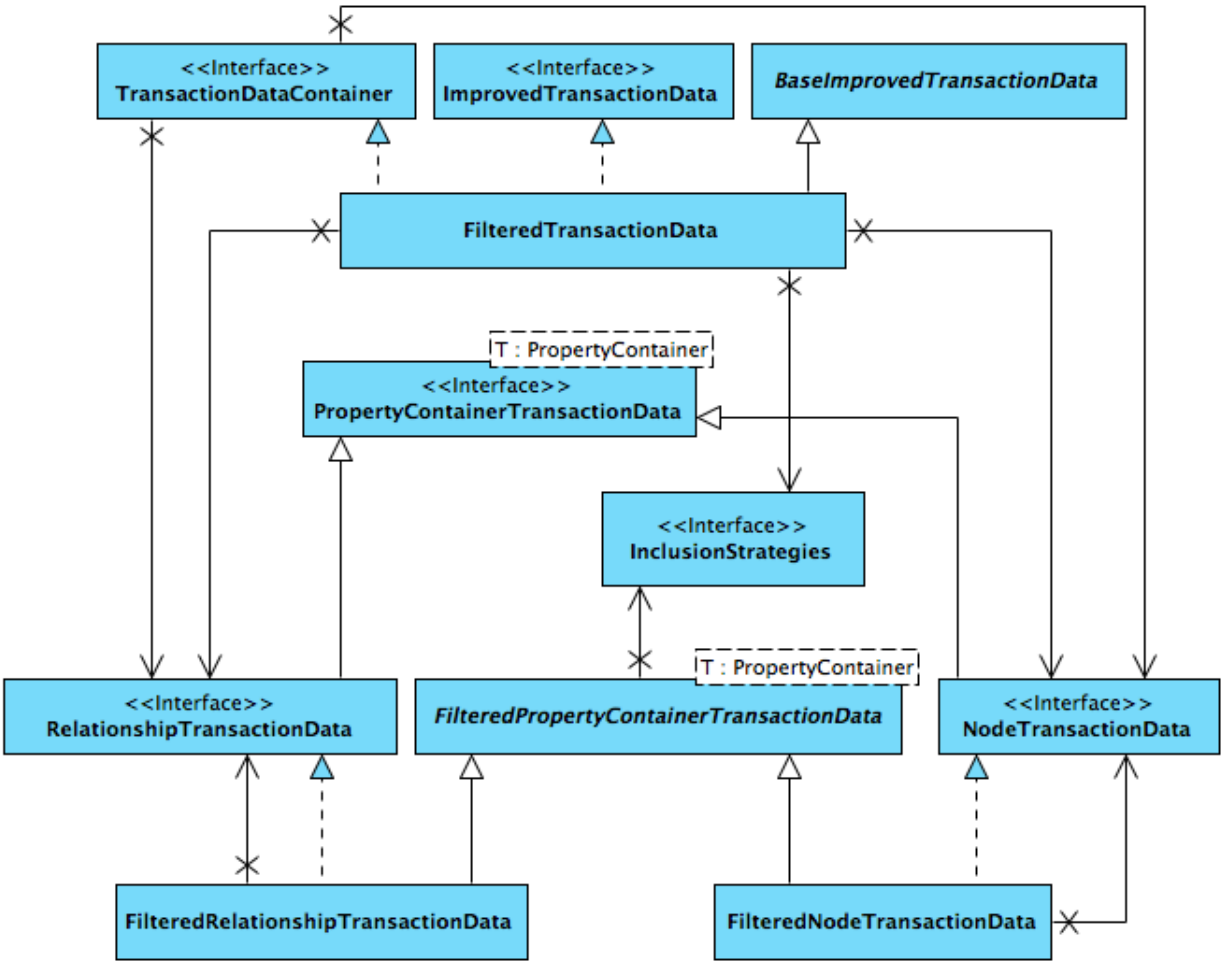

Figure 5.9: Filtered Transaction Data

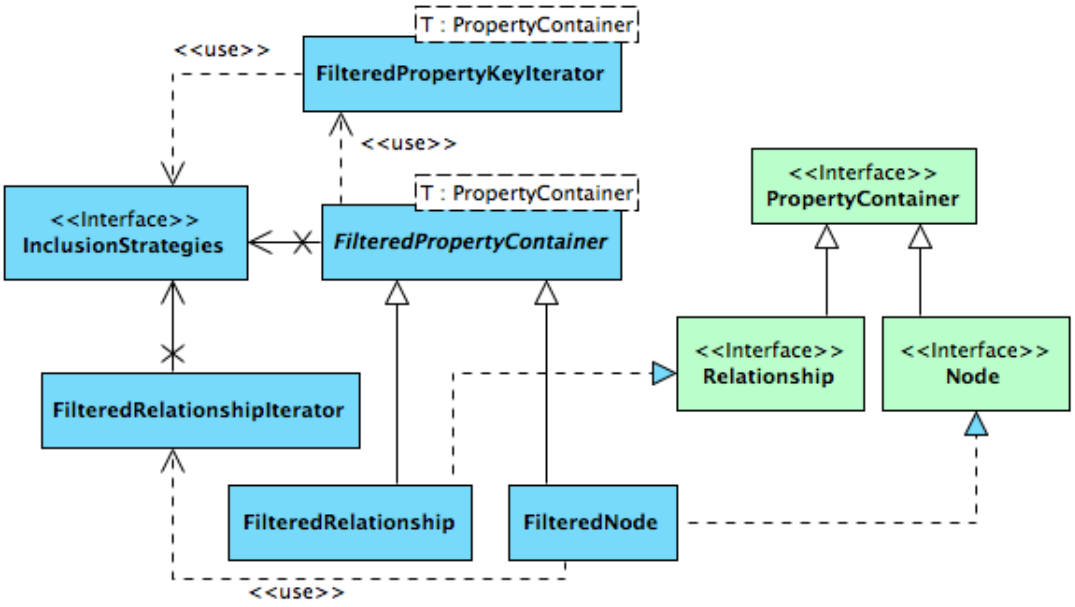

Figure 5.10: Filtered Nodes and Relationships

5.1.3 Batch Operations

Thanks to Neo4j's support for transactions and the API improvements discussed above, modules can write useful metadata to the graph and keep them up to date. Since there is some overhead inherent to transactional support in concurrent environments, Neo4j provides a mechanism for fast initial data import into the database with no transactional support, called the `BatchInserter`. Such an import must be run in a single-threaded fashion and has some functional limitations, such as the inability to delete nodes and relationships. The intentional lack of transactional support means that existing `TransactionEventHandlers` do not get notified of graph mutations. Consequently, modules that rely on metadata written to the graph cannot be used with this database population approach.

To solve this issue, a set of classes that simulate transactions when inserting data in batches has been introduced. `TransactionSimulatingBatchInserter` is a decorator of `BatchInserter`, which allows for the registration of `TransactionEventHandlers`. It records all graph mutations in an instance of `BatchTransactionData`, before delegating the actual work to the wrapped `BatchInserter`. After a configurable number of mutations occur (1,000 by default), the recorded mutations are submitted to the registered `TransactionEventHandlers`. Since `BatchTransactionData` implements Neo4j's `TransactionData`, this situation is indistinguishable from regular transactional graph database operations, from `TransactionEventHandlers`' point of view. Therefore, all code developed for transactional operations can be re-used for batch inserts without any modifications at all, including GraphAware's own `ImprovedTransactionData`. Figure 5.11 shows the design of this solution.

5.1.4 Module Lifecycle Management

With all the supporting functionality introduced, we now present the design of the main framework functionality. Let us first state the basic philosophy behind the framework. Modules, i.e., software components that wish to store metadata about the graph (most likely, but not necessarily in the graph itself) and keep them up to date, register with the framework. The framework, in turn, manages their lifecycle and notifies these modules at runtime about soon-to-be-committed changes to the database. The framework constitutes a few Java classes shown in Figure 5.12.

Modules Each module must implement the `GraphAwareModule` interface, which allows the framework to manage its lifecycle and delegate work to it. First of all, each module has to be able to provide its ID, which must be unique within the context of the framework. Multiple instances of the same module are allowed, as long as they have different IDd. The ID allows the framework to uniquely identify the module.

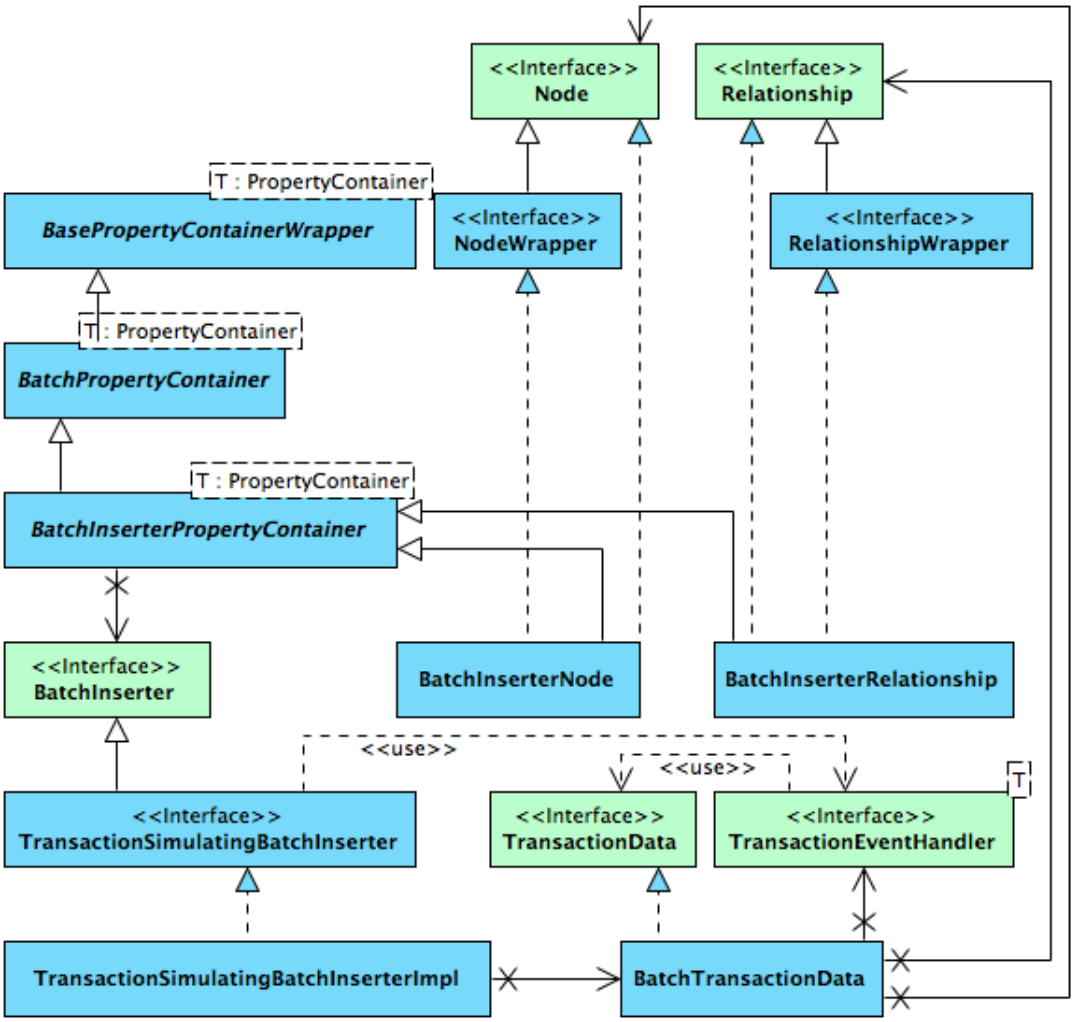

Figure 5.11: Batch Operations Support

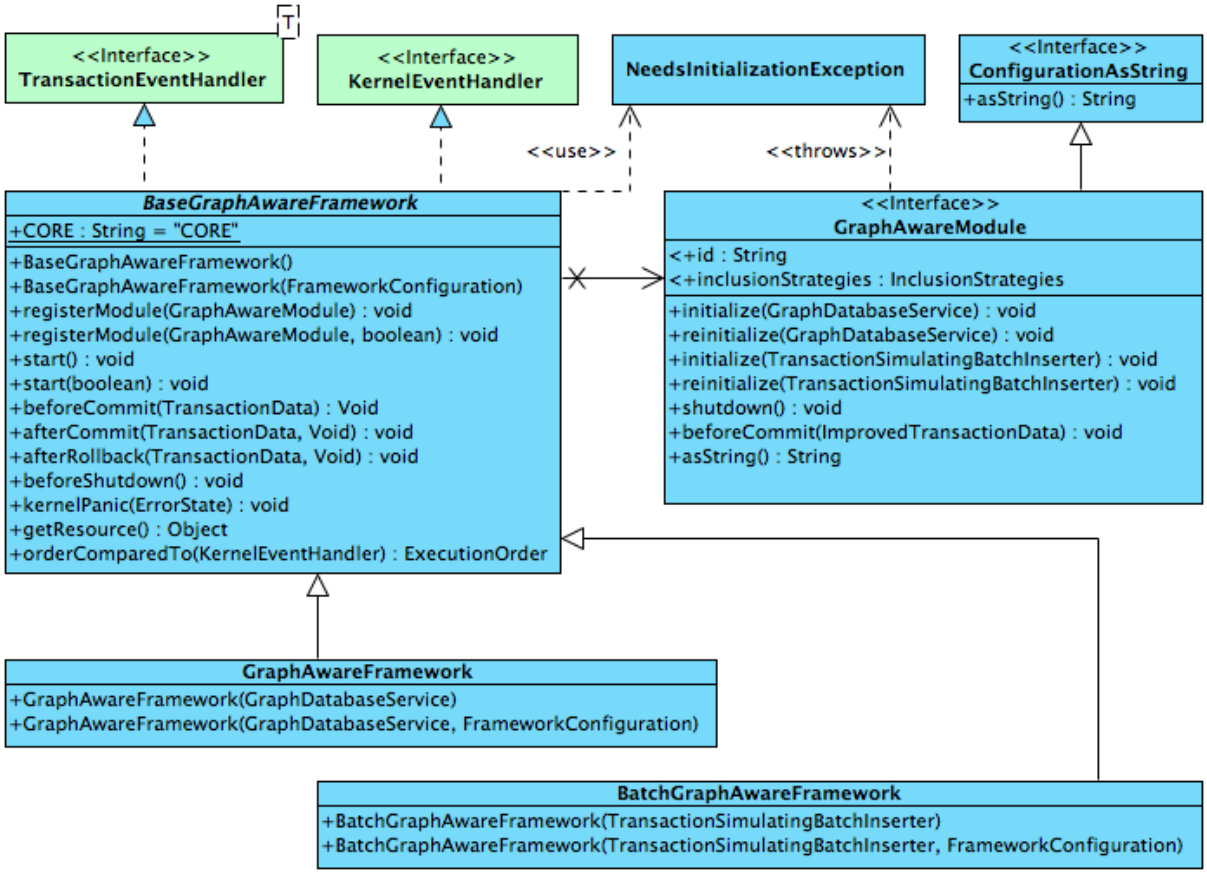

Figure 5.12: Core GraphAware Framework Classes

Secondly, each module (as well as any other class that implements `ConfigurationAsString`) must be able to represent its entire internal configuration as a single string. This does not have to be human-readable; the only requirement is that it changes when the configuration of the module changes. That way, the framework can detect changes in module configurations and act upon them, as described shortly.

Next, each module must provide an instance of `InclusionStrategies`, indicating, which parts of the graph it is interested in. As mentioned before, the Relationship Count Module, for instance, is not interested in nodes and their mutations at all. In fact, `InclusionStrategies` themselves are a good example of internal module configuration, described in the previous paragraph, as long as they can be changed by the module's user (i.e., they are not hard-coded).

The core module logic resides in the body of the `beforeCommit` method, which accepts an instance of `ImprovedTransactionData` as an argument. This is, in fact, an instance of `FilteredTransactionData`, filtered according to the provided `InclusionStrategies`. Based on this data, the module can create, update, and/or delete metadata in the graph. For example, the Relationship Count Module, discussed below, watches for relationship creations, changes, and deletions and updates the relationship count cache for each node accordingly.

In case the framework is used on an existing graph for the first time, metadata that each module would have written, had it been running since the graph was empty, has to be created. For that reason, each module implements the `initialize` method, which must achieve exactly that. This method is allowed to perform potentially expensive global graph operations. Similarly, in case metadata has become out of sync or corrupt, for instance due to the fact that the graph has been mutated without the framework running, or when the module's configuration has changed since the last run, the module must delete and rebuild all its metadata using the `reinitialize` method[5].

Finally, each module must implement the `shutdown` method, which is called by the framework before the database is shut down. Modules must release resources and commit any pending updates in this method.

Instantiation and Module Registration Two concrete classes are provided, representing the framework. `GraphAwareFramework` is intended for regular usage with `GraphDatabaseService`, which must be provided as an argument to its constructor. `BatchGraphAwareFramework`, on the other hand, accepts an instance of `TransactionSimulatingBatchInserter` and is intended for batch inserts, as the name suggests. Once instantiated, `GraphAwareModules` can be registered with the framework, using the `registerModule` method.

Framework Configuration There are some aspects of the framework that need to be globally configurable (in contrast to module internal configuration). An example of such a configuration is the delimiter used for separating information in string representations of nodes and relationships, discussed already. Modules can choose to be made aware of the framework's global configuration by implementing the `FrameworkConfigured` interface, shown in Figure 5.13. These modules will be informed about the configuration once they register with the framework, via invocation of their `configurationChanged` method. An instance of `FrameworkConfiguration`, encapsulating the framework configuration information, is passed to the method as an argument. `DefaultFrameworkConfiguration` with sensible defaults is provided; it is used automatically when no explicit configuration is passed to the framework's constructor. Note that `FrameworkConfiguration` also encapsulates some logic dependent on that configuration; for instance, the `createPrefix` method accepts a module ID as an argument and creates a prefix that the module should use when writing and reading graph metadata.

[5]Note that each one of these methods must be implemented twice; once for usage with the transactional `GraphDatabaseService` and once for batch insertions via `BatchInserter`. The reason for this is that the `initialize` and `reinitialize` operations are not run in the context of an about-to-be-committed transaction; instead, they are run against the database (or batch inserter) before it can be used by anyone else, and must manage starting and committing transactions (if applicable) themselves.

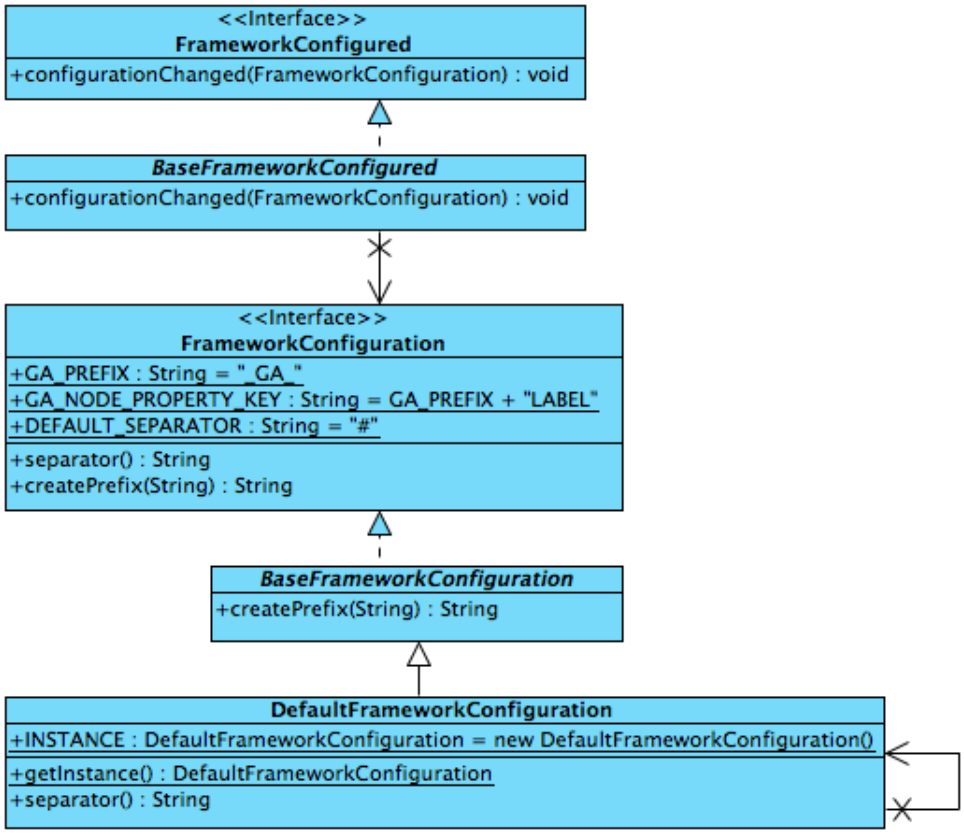

Figure 5.13: Class Hierarchy for Framework Configuration

Startup and Shutdown When all desired modules have been registered with the framework, the `start` method must be invoked. At that point, the framework checks that all registered modules have been run before with the same configuration they are reporting this time, using their `asString` method. Modules that have not been run before get the `initialize` method invoked on them, whilst `reinitialize` is called on modules that changed their internal configuration since the last time the database was started. Finally, any modules that threw a `NeedsInitializationException` during the last run of the database/framework must also `reinitialize`. Modules can throw this exception when they detect their own metadata is out of sync. Before the database is shut down, it invokes the `beforeShutdown` method on all registered `KernelEventHandlers`, which is an interface that `GraphAwareFramework` also implements for this reason. When this method gets invoked, the `shutdown` method is invoked on each registered `GraphAwareModule`. This paragraph is summarised by the sequence diagram in Figure 5.14.

Reference Node Information about which modules were registered during the last run of the framework, about individual module configurations, and about modules that need to be re-initialised is stored in properties of the reference node (also called the root node), which is the node with ID=0. The framework checks for the node's presence upon startup and fails fast, if this node is not present in the graph. At runtime, the framework prevents any transaction that tries to delete this node from

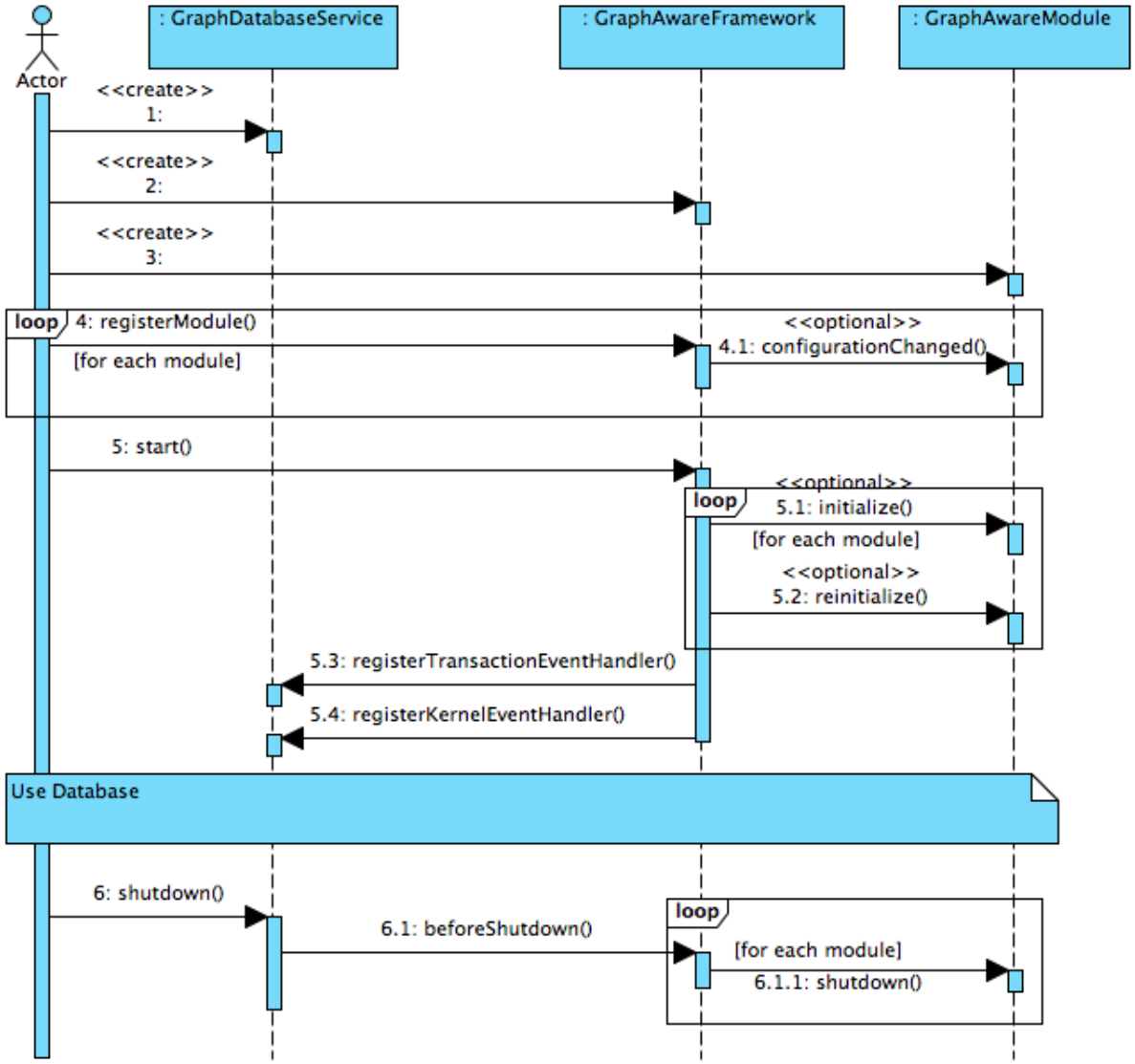

Figure 5.14: Sequence Diagram of Framework Startup and Shutdown

committing. Like other metadata that the framework and its modules write to the graph, these are prefixed with a framework-specific string to prevent interference with user-written properties. Therefore, the framework places no limitations on usage of the reference node; it can (but does not have to) be used as a regular graph node. For each registered module, there is a property with key equal to `_GA_CORE_MID`, where `MID` is the module ID, and value equal to `CONFIG:configAsString`, where `configAsString` is the string returned by the module's `asString` method. In case the module threw a `NeedsInitializationException` during its lifetime, the value of the property gets changed to `FORCE_INIT:timestamp`, where `timestamp` is the timestamp of the first occurrence of the exception.

Runtime The framework keeps a list of modules that wish to be notified about transaction events. In order to let these modules know about upcoming graph mutations, the framework itself implements Neo4j's `TransactionEventHandler`. Before a transaction commits, the framework first translates the `TransactionData`, provided by Neo4j, into `ImprovedTransactionData`, introduced earlier. It then wraps this data in an instance of `FilteredTransactionData` for each module, according to the module's `InclusionStrategies`, and passes the data to the module's `beforeCommit` method. This happens in the same order in which the modules registered with the framework. This runtime behaviour is summarised by the simplified sequence diagram in Figure 5.15. As mentioned before (not shown in the diagram), if a module detects that its own metadata does not seem to be up to date, it can choose to throw a `NeedsInitializationException`. This is caught by the framework, logged as a warning, and the module is marked for re-initialisation upon the next framework startup.

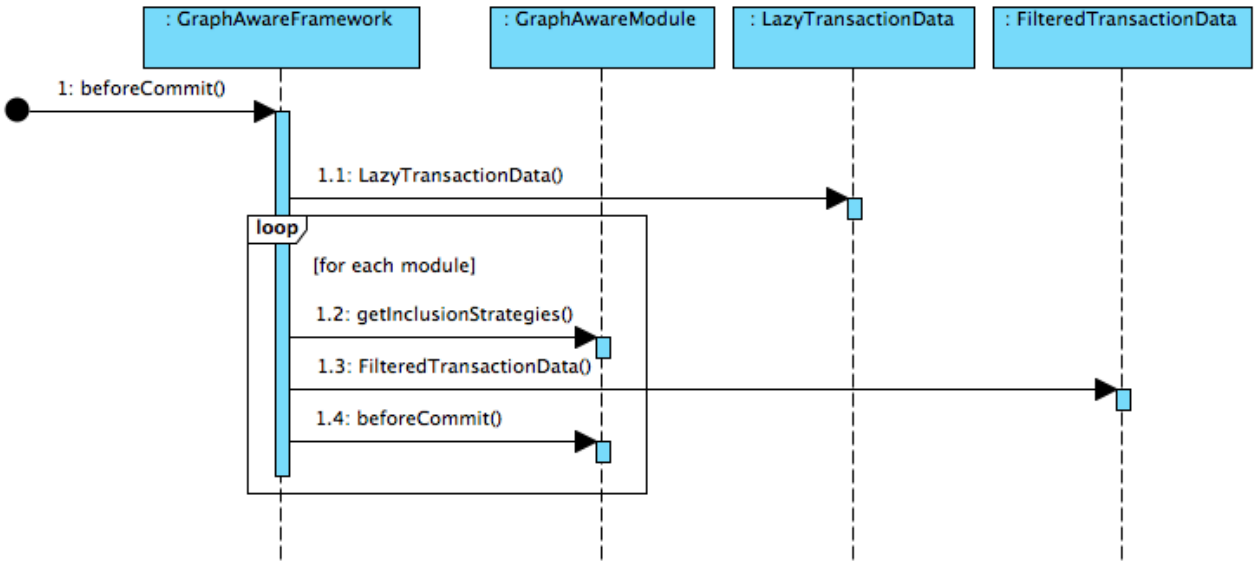

Figure 5.15: Sequence Diagram of Framework Runtime Usage

5.2 Relationship Count Module

The Relationship Count Module is a GraphAware Framework module that implements the vertex degree cache from Chapter 4. It caches relationship counts as properties on nodes. In fact, two versions of the module have been designed and implemented. `FullRelationshipCountModule` and corresponding classes implement the theory from Chapter 4 with some minor deviations driven by performance, API clarity, and other considerations. These are clearly stated as encountered. On the other hand, `SimpleRelationshipCountModule`, which is not discussed any further in this chapter, is a significantly simplified version that only takes relationship types and directions into account, but ignores properties. Thus, none of the concepts discussed in Chapter 4, such as general-to-specific ordering or cache compaction apply to `SimpleRelationshipCountModule`. Its purpose is mainly performance evaluation, which is discussed later in Chapter 6.

5.2.1 Property and Relationship Representations

Cacheable Representations Functionality based on some of the concepts introduced in Chapter 4, namely general-to-specific ordering for properties (Definition 4.16) and relationships (Definition 4.23), their mutual exclusivity (Definitions 4.22 and 4.28), the limited generalisation operation for relationship descriptions (Definition 4.33), and the set of all generalisations (Definition 4.34), is implemented by `CacheablePropertiesDescription` and `CacheableRelationshipDescription` class hierarchies. Figure 5.16 shows the `CacheablePropertiesDescription` hierarchy (pink) in the context of supporting framework classes (blue, method signatures omitted).

The method signatures on `CacheablePropertiesDescriptionImpl` should clearly explain their purpose. When comparing a cacheable properties description to another properties representation using `isMoreGeneralThan`, `isMoreSpecificThan`, or `isMutuallyExclusive`, it is important to note that the objects might not (and typically will not) be aware of all the property keys available in the graph. This is not a problem, however, since a non-existent property key is treated as an undefined property. The comparisons are done using all the property keys of the two compared objects combined. Any other property keys are, therefore, undefined for both of the objects, thus equal.

`generateAllMoreGeneral` is an operation that produces the set of all more general properties descriptions, as defined in Definition 4.34. It is a recursive function; on each level of the recursion, a property constraint is replaced by a wildcard. The base case, and thus terminating condition, occurs when all properties have been mapped to a wildcard, i.e., no further generalisation is possible. This method is used when performing the cache compacting operation. Whilst the cache knows about all its cached relationship counts, a single relationship description does not have this visibility. Therefore, it might

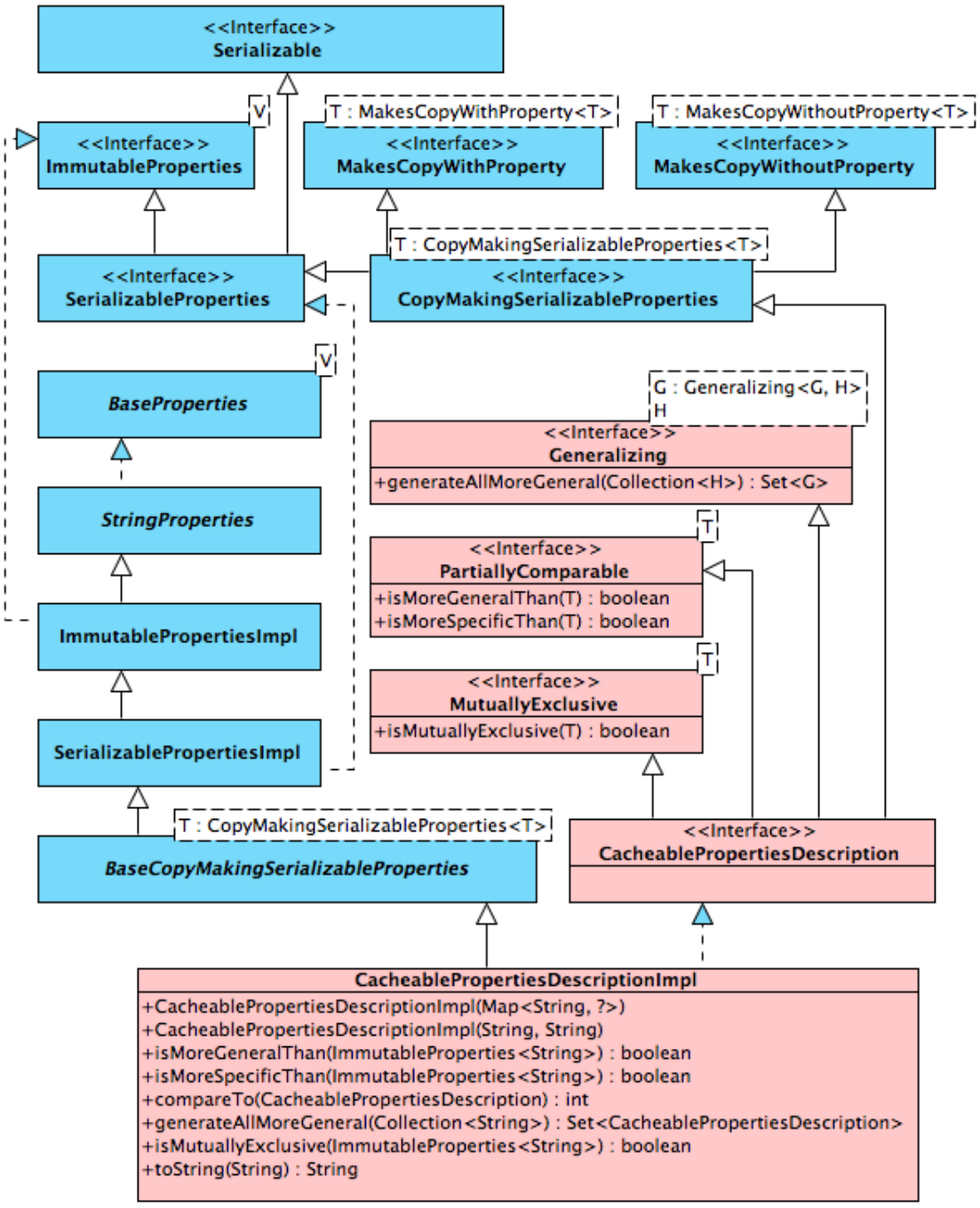

Figure 5.16: Cacheable Properties Description, Related Classes, and Supporting Framework Classes

be unaware of some properties that are present on other relationships of the same type in the cache. Consequently, in order to be able to generalise these properties, the `generateAllMoreGeneral` takes an `additionalKeys` argument, which the caller uses to make the object aware of other known property keys that it should use for generalisation. Listing 5.2 shows the Java implementation of this generalisation.

Very similar class hierarchy, presented in Figure 5.17 (this time without framework support) is used for cacheable relationship representations.

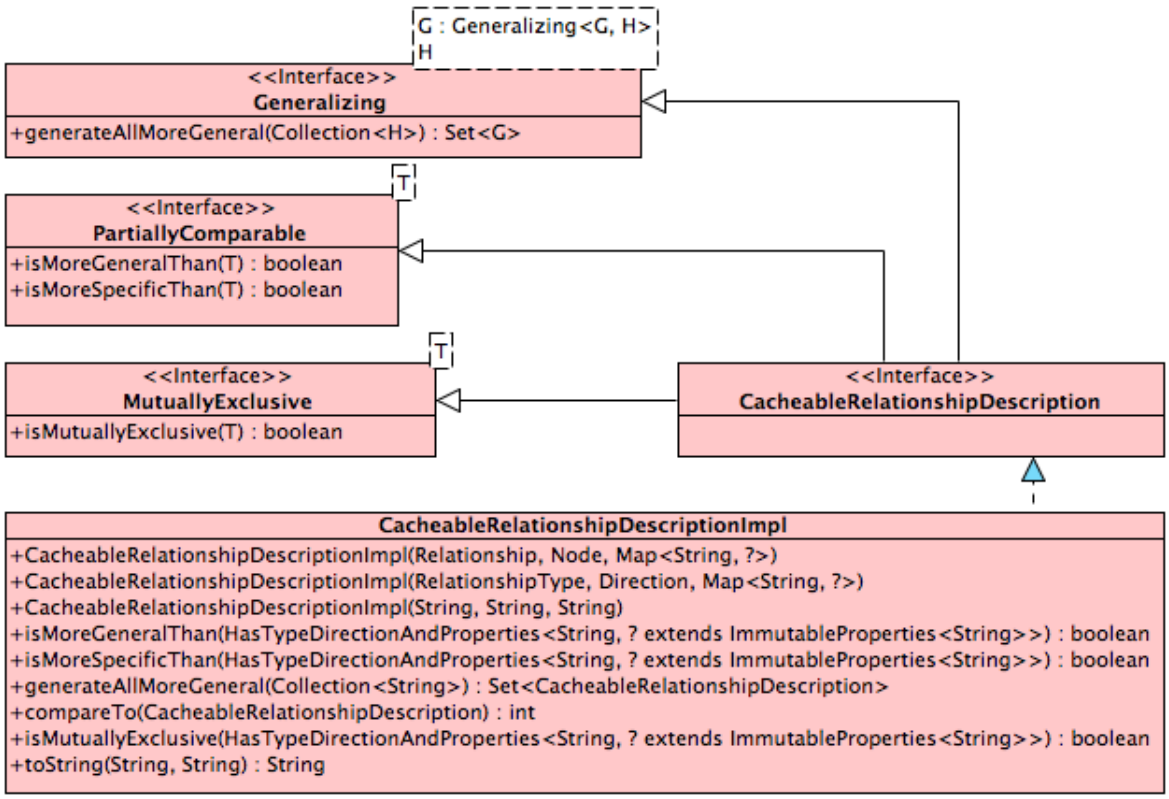

Figure 5.17: Cacheable Relationship Description and Related Classes

Query Representations Relationship and properties descriptions, used for describing which relationships to count, also called query representations, have different requirements from those used as cached representations. First of all, they do not need to be self-generalising and as long as the cached representations can be compared to them (for general-to-specific ordering and mutual exclusion), these query representations do not need to be mutually comparable in these respects. Secondly, whilst cached relationship representations are limited to constraining properties to a specific value or wildcard, the query representations can, theoretically, specify any property constraint predicate.

In the interest of API simplicity, however, an artificial constraint has been imposed on these predicates in the sense that they can also only take on a concrete value, or a wildcard. However, no aspect of the framework's or this module's architecture prevents this constraint from being lifted in future versions.

```
public Set<CacheablePropertiesDescription> generateAllMoreGeneral(
        Collection<String> additionalKeys) {

    Set<CacheablePropertiesDescription> result
        = generateAllMoreGeneral(this, additionalKeys);
    result.remove(this);
    return result;
}

private Set<CacheablePropertiesDescription> generateAllMoreGeneral(
        CacheablePropertiesDescription properties,
        Collection<String> additionalKeys) {

    Set<String> nonWildcardKeys
            = nonWildcardKeys(properties, additionalKeys);

    Set<CacheablePropertiesDescription> result = new HashSet<>();
    result.add(properties);

    //base case
    if (nonWildcardKeys.isEmpty()) {
        return result;
    }

    //recursion
    for (String key : nonWildcardKeys) {
        result.addAll(generateAllMoreGeneral(
                properties.with(key, ANY_VALUE),
                additionalKeys));
    }

    return result;
}

private Set<String> nonWildcardKeys(
        CacheablePropertiesDescription properties,
        Collection<String> additionalKeys) {

    Set<String> result = new HashSet<>();

    for (String key : properties.keySet()) {
        if (!ANY_VALUE.equals(properties.get(key))) {
            result.add(key);
        }
    }

    for (String key : additionalKeys) {
        if (!properties.containsKey(key)) {
            result.add(key);
        }
    }

    return result;
}
```

Listing 5.2: Recursive Property Description Generalisation

Another API simplification has been introduced as a part of this design. Consider the following question: how many `INCOMING` relationships of type `RATED` with property `rating` equal to 2 does a node have? There are two ways to interpret this question and, thus, two programmatic representations thereof. The question could be asking for relationships with `rating` equal to 2 and not care about other properties these relationships might have. It could also be asking for relationships with just the `rating` equal to 2 and all other properties undefined. To support this dichotomy and to prevent the need for explicit specification of all other properties' values for one or the other kind of query, two types of relationship (and thus also property) query descriptions have been designed; literal query descriptions and wildcard query descriptions, all shown in Figure 5.18. Literal query descriptions behave as if every property, which has not been explicitly constrained, is constrained to undefined. Wildcard descriptions, on the other hand, treat these unspecified properties as wildcards.

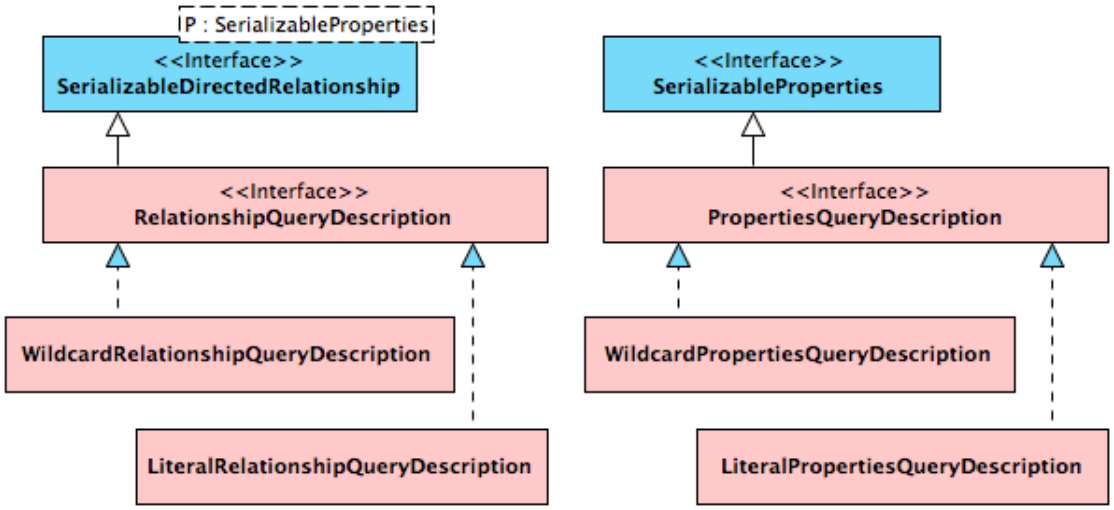

Figure 5.18: Relationship and Property Query Descriptions

5.2.2 Additional Strategies

Two additional strategies have been designed and used throughout the Relationship Count Module, in order to give its users more flexibility. Both are shown in Figure 5.19.

Properties Extraction Strategy Imagine a scenario where each node, which represents a user in a hypothetical movie rating application, has a `gender` property that takes on the values `"Male"` and `"Female"`. Assume further that users can rate movies, which is represented by a `RATED` relationship from a user node to a movie node. In such a system, it would not be an unreasonable requirement to count incoming `RATED` relationships for a movie node based on the gender of the originating node in order to find out, how much the movie appeals to men and women. For this reason, an interface called `RelationshipPropertiesExtractionStrategy` has been created. It allows for custom logic

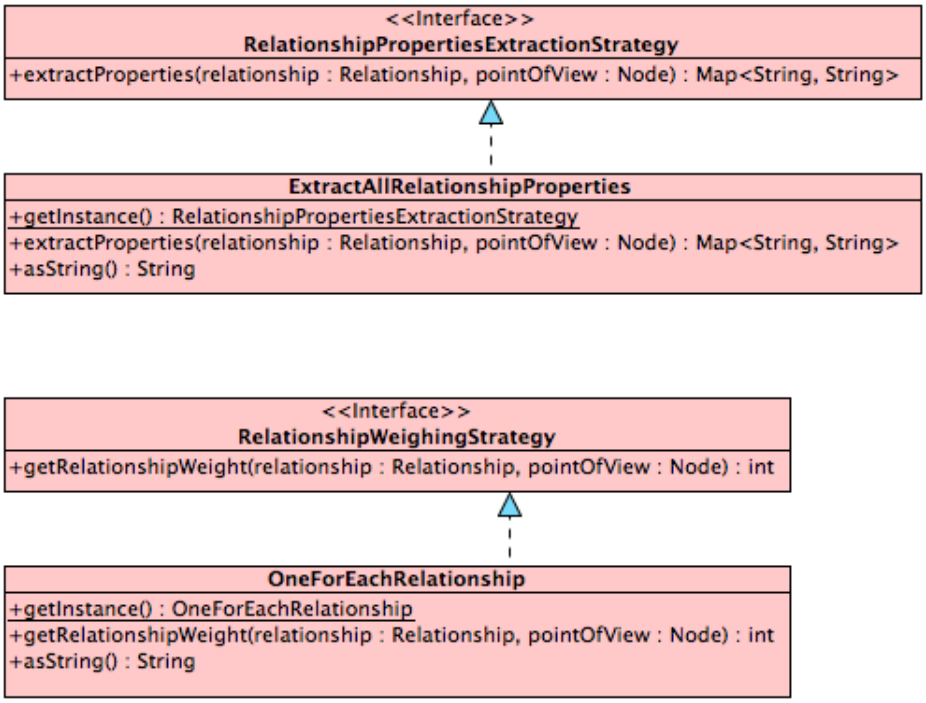

Figure 5.19: Additional Strategies for Relationship Count Module

that determines properties on relationships. Not only can it be used to exclude certain properties, based on the relationship type, for instance, but it can also be used to derive properties that are not actually present on the relationship. The movie example in this paragraph could thus be solved by deriving a `gender` property for `RATED` relationships by looking at the `gender` of the originating node. Such a derived property would then never appear on the relationship itself, but for the purposes of caching the relationships counts, it would be treated as an existing property[6]. The provided default implementation, `ExtractAllRelationshipProperties`, simply returns each relationship's properties as they are.

Relationship Weighing Strategy We mentioned before that there are different ways of modelling the same domain as a property graph. Consider, for example, modelling multiple interactions of the same kind between two nodes. One way would be creating a separate relationship for each interaction; the other way would be maintaining a property on a single relationship, indicating the number of interactions. For cases modelled using the latter option, `RelationshipWeighingStrategy` is provided. Whilst the default implementation, `OneForEachRelationship`, gives each relationship a weight of 1 (thus it is counted as one relationship), other implementations could determine the weight by looking at a specific relationship property.

Consider another use case, where friendship is modelled as a `FRIEND_OF` relationship with a `strength` property taking on values 1 to 10. To find the average friendship strength for a node, one could

[6] such a property would have to be fixed for the lifetime of the node though, because changes to node properties are not handled by the module; this is a limitation

be caching these relationships with two instances of the Relationship Count Module; one counting each relationship as one, and the other taking the `strength` property as the number of relationships. Computing the average would then involve reading two properties of a node and dividing their values.

5.2.3 Degree Caching Node

We now develop the cache, defined in Chapter 4 in Definition 4.29, and the compaction operation outlined in Algorithm 4.1, encapsulated in a representation of a node that is capable of caching its relationship counts. It is called `RelationshipCountCachingNode` and it is presented, along with its full (as opposed to simple property-ignoring) implementation called `FullRelationshipCountCachingNode`, in Figure 5.20.

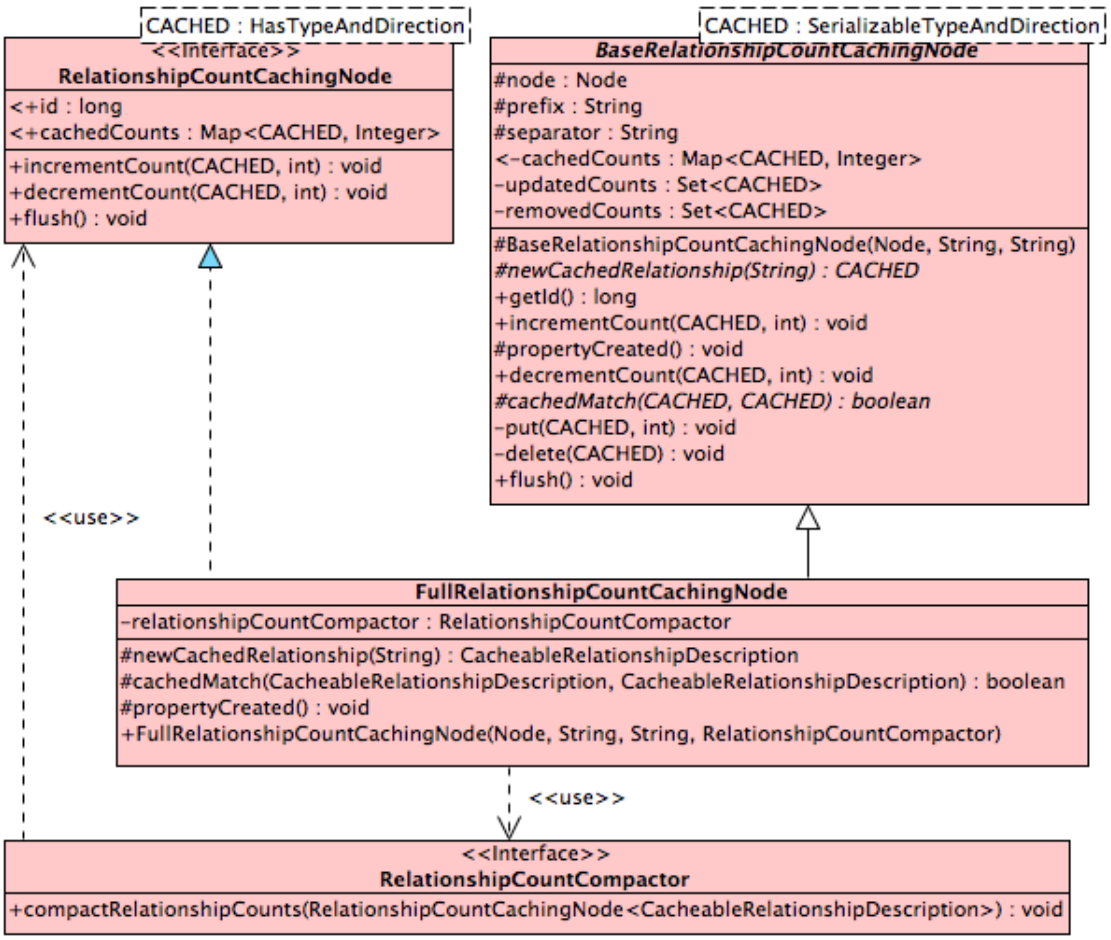

Figure 5.20: Relationship Count Caching Node

Full Relationship Count Caching Node The class holds a reference to the Neo4j node for which it caches relationship counts. On top of that, it provides three basic operations for manipulating these cached counts. First, using the `getCachedCounts` method, it is able to return all relationship counts cached as properties on the node, directly corresponding to cached degrees from Definition 4.29, as a

map of relationship descriptions to integer counts. It is also capable of incrementing and decrementing the cached count of a relationship with a given description by a given integer, using the `incrementCount` and `decrementCount` methods, respectively.

When decrementing a count for a relationship description, typed as `CacheableRelationshipDescription`, a cached count for that description is found, such that the cache relationship description is more specific or equal to the provided relationship description. The value of the cached count for the description is then decremented. If the value happens to be zero afterwards, the entire cached count is removed, i.e., the node property is deleted. If the value falls under zero, or no cached count to decrement can be found, a `NeedsInitializationException` is thrown, because it means the module's metadata is out of sync.

Similarly, when incrementing a count for a relationship description, a more specific or equal cached count for that description is found and its value incremented. If no suitable cached count exists, a node property is created, where its key is the string representation of the provided relationship description and its value is the number by which the count should have been incremented (the integer `delta` argument). When a new property has been created, there is a chance that there are more cached relationship counts than dictated by the compaction threshold. Therefore, `RelationshipCountCompactor` is asked to `compactRelationshipCounts`. The compaction threshold is encapsulated in the `RelationshipCountStrategies` class and is 20 by default.

Before discussing compaction in detail, we briefly explain a performance optimisation performed on `BaseRelationshipCountCachingNode`. It has been found during performance testing and profiling that updating properties on Neo4j nodes takes a relatively long time. In fact, during initial testing with no optimisations, over 80% of the CPU time was spent getting nodes' properties from Neo4j, when inserting data with GraphAware Framework running and Relationship Count Module registered. This inefficiency has been solved by a number of optimisations, including the following.
`BaseRelationshipCountCachingNode` loads the properties from the underlying Neo4j node only once and keeps the relevant ones as `cachedCounts`. It also keeps the changes to these as `updatedCounts` and `removedCounts`. The `flush` method copies the created and updated properties back to the underlying node and removes the ones that should be removed. Unchanged properties are left untouched. This way, the object can be referenced for the duration of a transaction event handling and only flushed at the end, minimising database access.

Compaction The compaction algorithm, as shown in Algorithm 4.1 and the Property Change Frequency function from Definition 4.52 have been implemented using the set of classes shown in Figure 5.21. `ThresholdBasedRelationshipCountCompactor` is instantiated and provided with the config-

urable `compactionThreshold`; every time it is asked to perform compaction, it first checks whether it is at all needed. If that is the case, it uses a `GeneralizationStrategy` to produce all generalisations of all cached counts, ordered from best to worst. The only provided implementation, `PropertyChangeFrequencyBasedGeneralizationStrategy` does so by implementing the Property Change Frequency function from Definition 4.52, using the informational value from Definition 4.48 as a score for its `ScoredCompactibleRelationship` inner class, a wrapper of `CacheableRelationshipDescription`. The compactor takes the first `CacheableRelationshipDescription` (i.e., the one with the highest score) that results in actual cache compaction and uses it for that purpose. After the compaction, it checks again, whether the total number of cached counts is now below or at the desired threshold. If not, the process is repeated. In case no compaction can be performed and the threshold still has not been reached, a meaningful warning is logged. This situation means that the threshold is not greater than twice the number of relationship types in the system, as proven in Theorem 4.35.

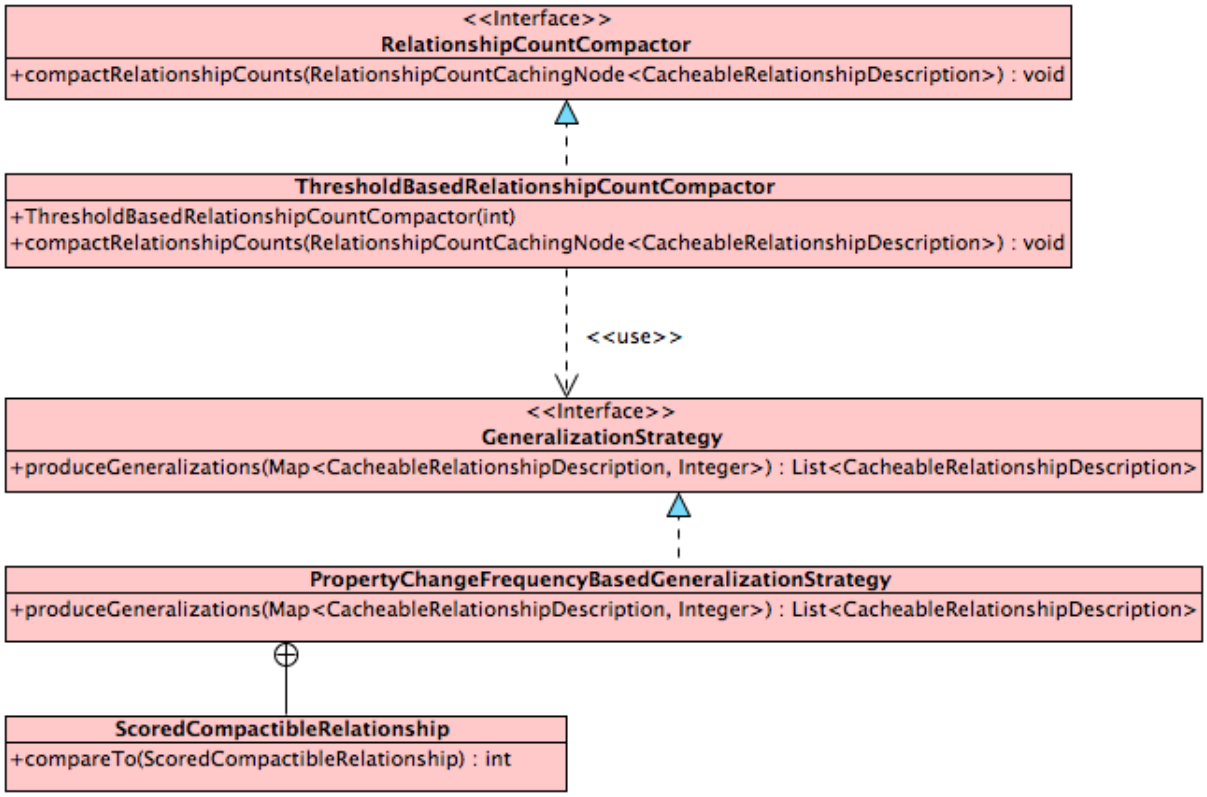

Figure 5.21: Relationship Count Compactor

5.2.4 Full Relationship Count Module

Degree Cache A global representation of the degree cache is depicted in Figure 5.22. Its responsibility is to handle created and deleted Neo4j relationships for a node during transaction event handling. It creates a `CacheableRelationshipDescription` based on the `RelationshipCountStrategies` it has been configured with. These include framework `InclusionStrategies` as well as module-

specific strategies, discussed earlier, and the compaction threshold. Next, it obtains an instance of `RelationshipCountCachingNode`, either from its own cache, or by creating it, and increments or decrements the relationship counts on it.

`BaseRelationshipCountCache` has its own caching mechanism for `RelationshipCountCachingNodes`, which is initialised when `startCaching` is invoked on it, and later flushed by flushing each individual `RelationshipCountCachingNode`, when `endCaching` is called. This way, if there are multiple relationships created/deleted for a single node in the same transaction, which is not uncommon, the node and its properties only get read and written back once from/to the database. Since a single transaction is committed (and its transaction events handled) by a single thread, this caching is performed against `ThreadLocal`.

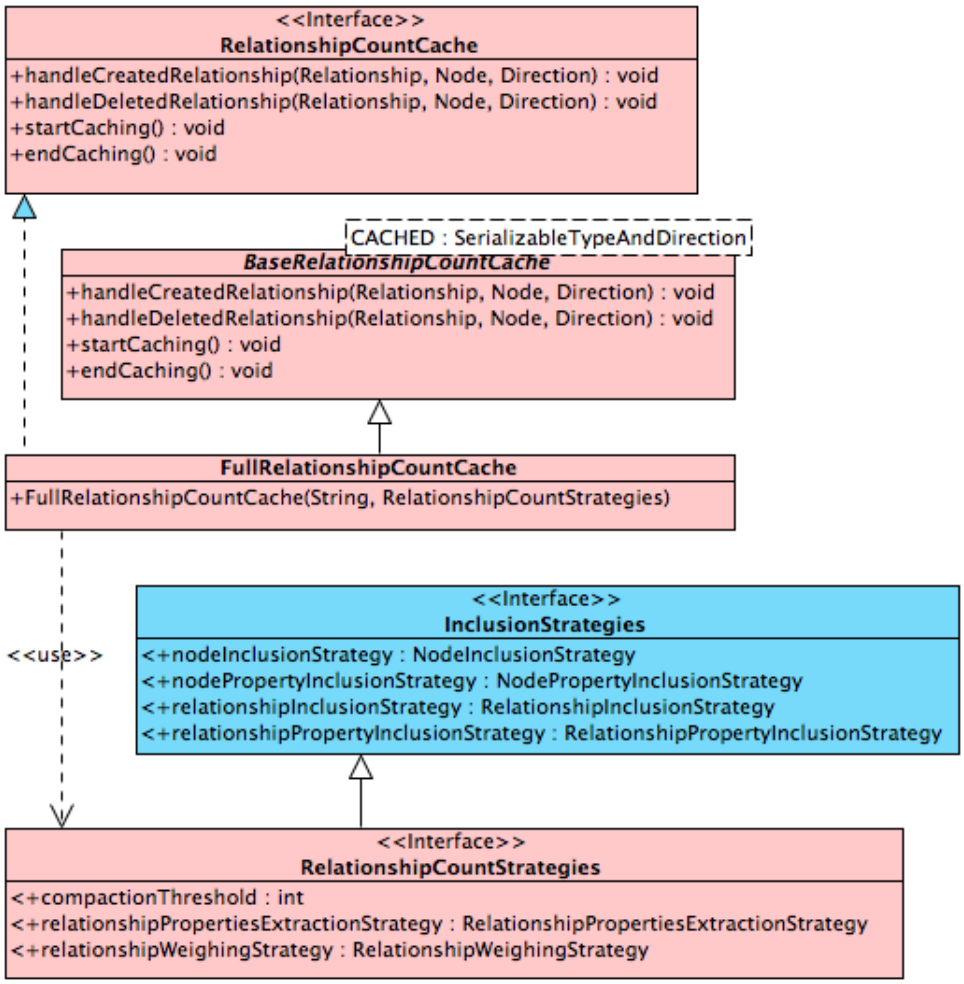

Figure 5.22: Relationship Count Cache

Module Finally, the `FullRelationshipCountModule`, shown in its surrounding context in Figure 5.23, receives information about transaction events from the framework as `ImprovedTransactionData` and uses the `FullRelationshipCountCache` to handle that information. For created relationships, it calls the `handleCreatedRelationship` method on the cache twice; once for each node participating in the

relationship. The same is true for deleted relationships and the `handleDeletedRelationship` method. Changed relationships are treated as if the old version of the relationship was deleted, and the new one created, thus resulting in a total of four cache calls.

Note that the `handleCreatedRelationship` and `handleDeletedRelationship` methods on the `RelationshipCountCache` take three arguments: the `relationship` that has been created/deleted, the node from whose `pointOfView` the relationship is being handled, and a `defaultDirection`, which is only used in case the relationship is a loop. When handling a relationship, the `FullRelationshipCountModule` calls the appropriate method twice, as mentioned above, once with `INCOMING` and once with `OUTGOING` `defaultDirection`. This way, if the relationship is a loop, it gets accounted for as incoming and outgoing, as established in Theorem 4.25.

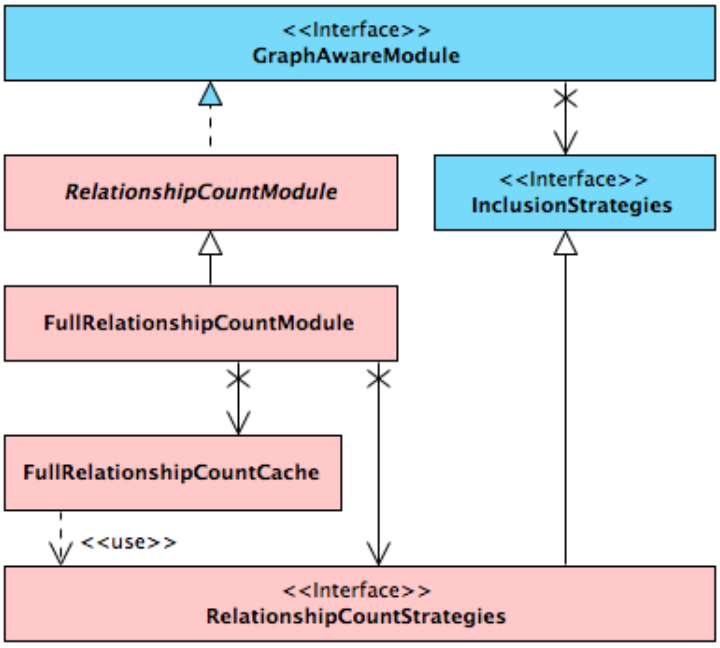

Figure 5.23: Full Relationship Count Module

5.2.5 Degree Counting Nodes

Even without any caching at all, we now have a convenient API for describing a relationship count query. Therefore, we introduce the notion of `RelationshipCountingNode`, a representation of a node capable of determining its own degree, based on a provided relationship query description. The hierarchy stemming from this representation is presented in Figure 5.24.

Naive Counting Given a `RelationshipQueryDescription`, the `FullNaiveRelationshipCountNode` can count relationships without any caching at all, by traversing and inspecting every single one of the node's relationships. For this reason, it needs to be aware of the `RelationshipWeighingStrategy` and

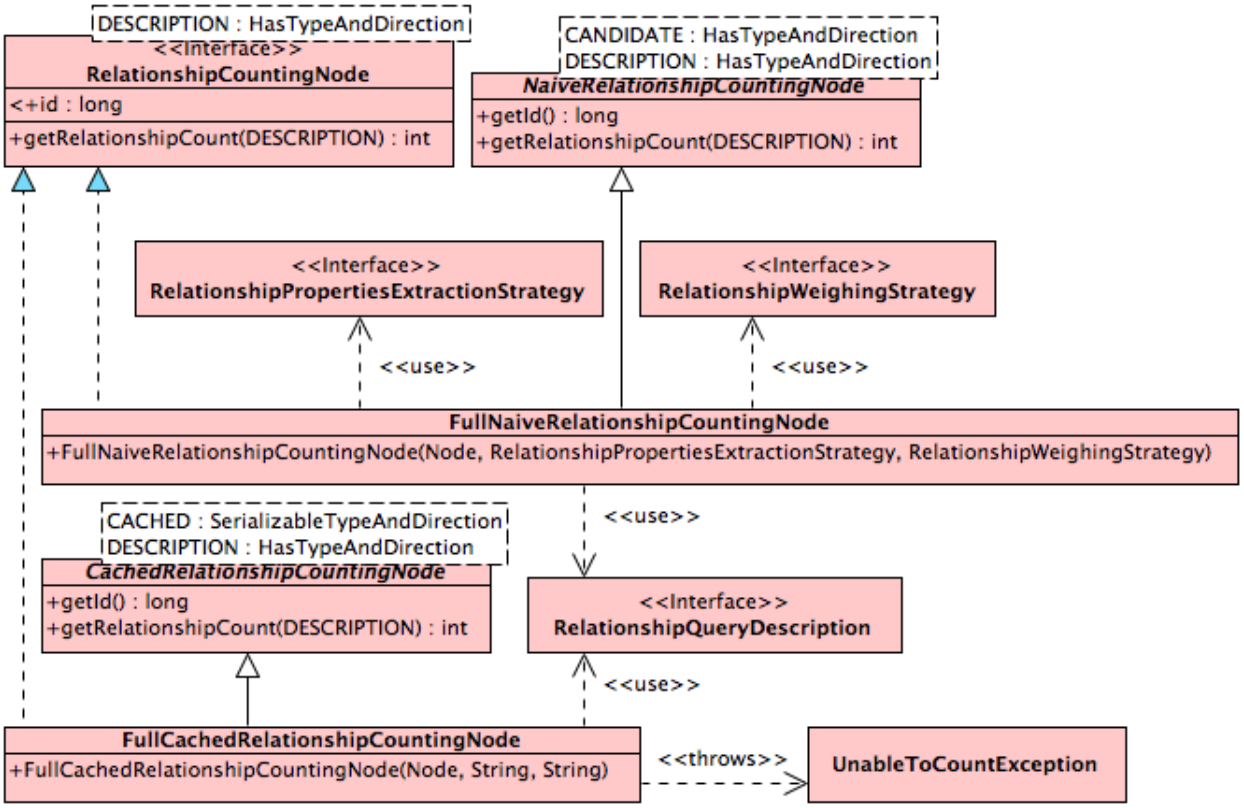

Figure 5.24: Relationship Counting Nodes

`RelationshipPropertiesExtractionStrategy`, which can be passed into its constructor.

Cache-Based Counting `FullCachedRelationshipCountingNode`, on the other hand, reads the relationship counts from the cached properties on the node, in the form of `CacheableRelationshipDescription` and associated integer count. According to Definition 4.37, there are situations where the count cannot be determined this way, because the compaction process has taken away the granularity needed for answering such a query. In this case, an `UnableToCountException` is thrown.

5.2.6 Public API

The purpose of the Relationship Count Module is not only to serve as an implementation of the theory from Chapter 4, but also to become a software component available for developers that work with Neo4j. For that reason, the API should be as simple and intuitive as possible. Hence, a public API layer has been designed, the implementation of which orchestrates the caching and counting processes described above. This API consists of the `FullRelationshipCountModule` itself, and a set of `RelationshipCounters`. A more detailed description of the API is provided in the User Guide in Appendix C, while a full description is provided by the JavaDoc, attached to the source code. The following paragraphs only describe the basics.

```
//standard way to instantiate the database
GraphDatabaseService database
    = new GraphDatabaseFactory().newEmbeddedDatabase("path/to/db");

//create the framework
GraphAwareFramework framework = new GraphAwareFramework(database);

//create and register module
FullRelationshipCountModule module = new FullRelationshipCountModule();
framework.registerModule(module);

//start the framework
framework.start();

//use database as usual
```

Listing 5.3: Using Full Relationship Count Module with GraphAware Framework

Relationship Count Module In order to use the Relationship Count Module with the framework without modifying the default configuration, the framework has to be instantiated, the module registered, and the framework started. Listing 5.3 shows this process in Java code.

Counting Relationships Once the module has been registered and the framework is running, three different concrete implementations of `RelationshipCounter` can be used to count relationships. They are shown in Figure 5.25. All of them provide a constructor, which is used to specify the type and direction of the relationships to be counted, and a fluent API to add property constraints for the relationships to count using the `with` method, passing in the property key and value. Then, the `count` method can be invoked, with a node as the only argument, returning the count of all relationships at that node, which fulfil the specified criteria. Any properties not explicitly constrained are treated as “do not care", i.e., as wildcards. The `countLiterally` method, on the other hand, counts relationships by treating unspecified property constraints as undefined. This means they must not be present on a relationship in order for it to be counted.

Instead of instantiating a `RelationshipCounter` directly, a convenient option of obtaining an instance from the `RelationshipCountModule` is provided. The advantage of that approach is that the counter is already pre-configured with all the strategies used by the module.

`FullNaiveRelationshipCounter` counts the relationships for a node in a naive fashion, i.e., does not use any caching; it simply inspects every relationship and checks, whether it matches the specified criteria. `FullCachingRelationshipCounter`, on the other hand, uses the cached counts on the node. In case the compaction process has taken away the needed granularity, this counter throws an `UnableToCountException`. Finally, the `FullFallingBackRelationshipCounter` first attempts to count relationship using the cached counts on a node, but falls back to the naive approach if

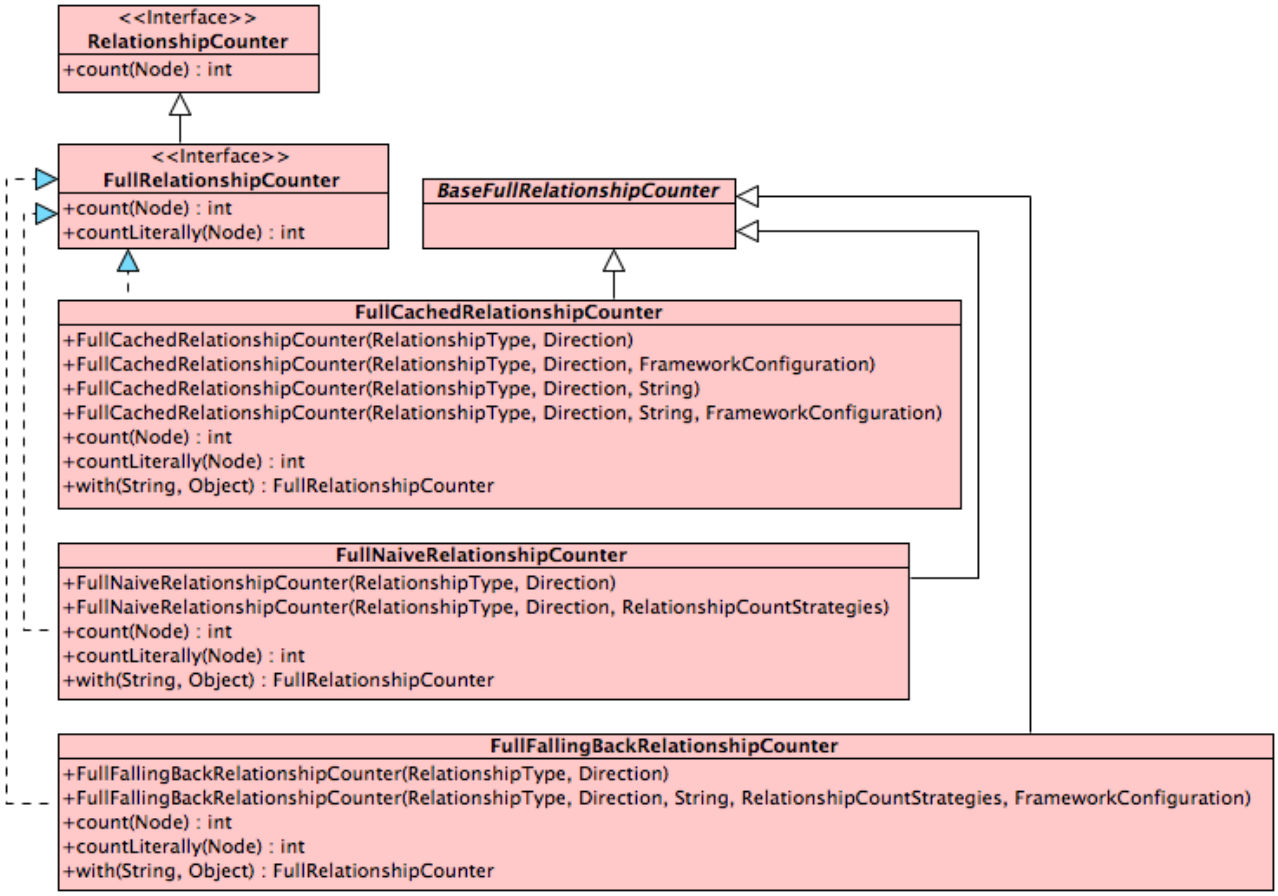

Figure 5.25: Relationship Counters

```
//assume database and module from previous listing available
Node pulpFiction = database.getNodeById(123);
FullRelationshipCounter ratingOfTwoCounter
    = module.cachedCounter(RATED, INCOMING).with("rating", 2);
ratingOfTwoCounter.count(pulpFiction); //returns the count
```

Listing 5.4: Using Full Cached Relationship Counter

`UnableToCountException` was thrown during the first attempt. Thus, it itself never throws the exception but always returns a count instead.

Listing 5.4 shows the usage of `FullCachedRelationshipCounter` for counting all incoming relationships of type `RATED` with a property `rating` set to 2 for a node representing Pulp Fiction.

Listing 5.5 shows the usage of `FullFallingBackRelationshipCounter` for counting all incoming relationships of type `RATED` with a property `rating` set to 2 for a node representing Pulp Fiction, with no other properties present on the relationship.

We conclude this chapter with a short summary. A framework for Neo4j, called GraphAware, has been designed, implemented, and fully tested. It supports development of analytical modules that wish to

```
Node pulpFiction = database.getNodeById(123);
FullRelationshipCounter ratingOfTwoCounter
    = module.fallingBackCounter(RATED, INCOMING).with("rating", 2);
ratingOfTwoCounter.countLiterally(pulpFiction); //returns the count
```

Listing 5.5: Using Full Falling Back Relationship Counter

write metadata into the graph and keep them up to date. One such module, called Relationship Count Module, has also been developed, in accordance with the theoretical analysis of the vertex degree problem in Chapter 4. With a clean and simple API, both components can be readily used by software developers.

Chapter 6

Evaluation

In this chapter, we evaluate the work presented thus far from a functional and performance perspective. The following section describes the use of automated unit and integration tests for evaluating functional correctness of the software. The subsequent section introduces performance tests, presents their setup, testing methodology, and measurement results.

6.1 Functional Testing

It has been mentioned previously that the software has been developed in a test-driven fashion [3]. When developing new functionality, a specification is written in the form of executable code that asserts the specification has been met, once the functionality has been implemented. Depending on its scope and purpose, this programmatic specification is called a unit test or an integration test. These tests are only as good as the developer that writes them; that is to say, they are not a formal proof of software correctness. When an error is discovered later during further testing efforts (e.g. user acceptance testing), a failing test that exposes the error is first written, followed by a fix of the error. All tests written for the software are run automatically as a part of the build process and the build fails if any of the assertions fail.

Automated unit and integration tests have been written for all the functionality of the GraphAware Framework and the Relationship Count Module, covering as many scenarios as practically possible[1]. Listing 6.1 shows an example of such a test: one of the tests verifying that a cacheable relationship description can be correctly constructed from a string. Unit tests have descriptive method signatures, so that is immediately clear what the test is meant to assert. To illustrate the extent to which the software has been tested, Listing 6.2 presents all unit test method signatures for the `GraphAwareFramework` class.

[1] There are theoretically infinitely many test scenarios that can be written for all but the most trivial functionalities. It is a part of the engineering process to find a pragmatic balance between exhaustive testing and the overall testing effort.

```
@Test
public void relationshipShouldBeCorrectlyConstructed() {
    //null in the following indicates no prefix, "#" is the delimiter
    CacheableRelationshipDescription desc
            = new CacheableRelationshipDescriptionImpl(
                "test#INCOMING#key1#value1#key2#value2", null, "#");

    assertEquals(INCOMING, desc.getDirection());
    assertEquals("test", desc.getType().name());
    assertTrue(desc.getProperties().containsKey("key1"));
    assertTrue(desc.getProperties().containsKey("key2"));
    assertEquals("value1", desc.getProperties().get("key1"));
    assertEquals("value2", desc.getProperties().get("key2"));
    assertEquals(2, desc.getProperties().size());
}
```

Listing 6.1: Example Unit Test for `CacheableRelationshipDescriptionImpl`

```
shouldNotWorkOnDatabaseWithNoRootNode()
moduleRegisteredForTheFirstTimeShouldBeInitialized()
moduleAlreadyRegisteredShouldNotBeInitialized()
changedModuleShouldBeReInitialized()
forcedModuleShouldBeReInitialized()
moduleAlreadyRegisteredShouldBeInitializedWhenForced()
changedModuleShouldNotBeReInitializedWhenInitializationSkipped()
shouldNotBeAbleToRegisterModuleWithTheSameIdTwice()
unusedModulesShouldBeRemoved()
usedCorruptModulesShouldThrowException()
unusedCorruptModulesShouldBeRemoved()
allRegisteredInterestedModulesShouldBeDelegatedTo()
noRegisteredInterestedModulesShouldBeDelegatedToBeforeFrameworkStarts()
moduleThrowingInitExceptionShouldBeMarkedForReinitialization()
moduleThrowingInitExceptionShouldBeMarkedForReinitializationOnlyTheFirstTime()
modulesCannotBeRegisteredAfterStart()
frameworkCanOnlyBeStartedOnce()
frameworkConfiguredModulesShouldBeConfigured()
unConfiguredModuleShouldThrowException()
shutdownShouldBeCalledBeforeShutdown()
shouldNotBeAllowedToDeleteRootNode()
```

Listing 6.2: Unit Test Method Signatures for `GraphAwareFramework`

Of the 14,200 lines of Java code comprising the GraphAware Framework, 9,100 (64%) are automated tests, exercising 92% of the main (i.e. non-test) codebase. The Relationship Count Module contains a total of 7,400 lines of code, of which 5,100 (69%) are tests, covering 97% of the code. Most of the untested lines are overloaded constructors and private constructors, used for preventing utility classes with static methods from ever being instantiated, i.e., unreachable code.

6.2 Performance Testing

The purpose of the Relationship Count Module is to store metadata about the graph in the graph itself and keep it up to date, so it can be retrieved faster at a later point in time. Caching relationship counts will have an inevitable negative impact on the write throughput of the database, since extra information is written to the database with each created, deleted, and changed relationship. On the other hand, since counting relationships by inspecting them individually takes time linearly proportional to the number of relationships, while reading the counts from cache requires constant time, we expect that reading relationship counts from cache outperforms naive counting for vertices with relationship count higher than some constant. Without any modules, we expect the framework itself to have no significant impact on the write throughput of the database, since `LazyTransactionData` evaluates information lazily, i.e., on demand, only when some module asks for the data. The aim of this section is to quantify these performance implications.

6.2.1 Use of Caches

This subsection provides information about Neo4j's caching mechanisms, an understanding of which is essential for designing the performance testing experiments. Recall from Section 2.4 that there are two levels of caching in the Neo4j architecture. The low level cache (or file system cache) caches the Neo4j data stored on the durable media in the same format as it is represented on the media. The size of this cache can be controlled by Neo4j configuration parameters and it is recommended that this cache is large enough to fit all the Neo4j record files, if at all possible [21]. Recall further that it takes 9 B for a node to be stored in this format. Therefore, one million nodes will only require approximately 8.6 MB of space in this cache, while a hundred million relationships would consume roughly 3 GB (33 B each). Consequently, modern servers can be often configured to comfortably accommodate this recommendation for many real-life graphs[2].

The high level cache (or object cache), on the other hand, caches data as Java objects, optimised for

[2]This also heavily depends on the number of properties stored in the graph and their sizes. The information is presented here as an illustration to be contrasted with the next level of caching.

fast traversals [21]. The total memory required to store a single node in this cache, including its object overhead, is 344 B. One million nodes would then require 328 MB, excluding any relationships and properties. One hundred million relationships would require 19.4 GB (208 B each). Since this memory requirement is much larger and cannot be accommodated for many graphs, Neo4j allows for setting a cache eviction strategy for this high level cache. Once the heap allocated to the JVM fills up with objects, some of them are evicted and subsequently garbage-collected by the JVM. The garbage collection process is controlled exclusively by the JVM and apart from the actual garbage collection algorithm, developers cannot influence the timing of the garbage collection process. This has to be taken into account when creating a performance testing strategy, since pauses caused by garbage collection, which could take significant amounts of time at irregular intervals, introduce an undesirable element of non-determinism into the tests.

6.2.2 Experiment Design

One way to perform the measurements would be choosing a graph from a real-life application, finding its mutation and query patterns, and writing a benchmarking suite that simulates these patterns. However, such benchmarks would be specific to the application, thus hard to generalise for other use cases. Instead, we design our experiments to evaluate a single measure at a time, whilst varying a parameter or a set of parameters to understand their impact on the measure. Using the knowledge of Neo4j, the GraphAware Framework and the Relationship Count Module built up throughout this report, we discern meaningful measures and parameters.

Write Throughput From the design of the Relationship Count Module, it is clear that creating, deleting, and changing relationships will be slower when the module is running, compared to using a "plain" database, i.e., Neo4j without the framework. Since the Relationship Count Module ignores any vertex mutations, they are also ignored for the purposes of this evaluation. Furthermore, we do not place any emphasis on the total number of vertices in the graph, a parameter that would be quite important for many other benchmarks. The rationale behind that decision is the fact that relationship counting is local to a vertex, no matter which way it is performed.

Since every mutating operation on the graph is written to disk, we disable both low level and high level caching for write throughput tests, in order to eliminate as much non-determinism as possible. However, some variance in measurements, caused by the operating system and non-uniform disk access times, will be inevitable.

We are interested in measuring how much longer it takes to create, delete, or change[3] a relationship when relationship counts are being cached. Depending on the use case, different numbers of relationships will be created, deleted, or changed within a single transaction. For instance, creating a friendship in a social network would create a single relationship in a single transaction. On the other hand, creating a new airport (node) and all the routes from and to that airport (relationships) in a travel planning application would create hundreds or thousands of relationships in a single transaction. Therefore, we vary the number of relationships created/deleted/changed in a transaction as a parameter when evaluating write performance.

The presence of relationship properties, their number, and their values have write performance implications for relationship mutations in both scenarios: with and without the Relationship Count Module. When creating a relationship with properties, the property store has to be accessed, which is not the case when creating property-less relationships. Additionally, with the Relationship Count Module up and running, the compaction process could be triggered by a created relationship, degrading performance even further. Hence, we also vary the number of relationship properties and their value change frequency as a parameter when measuring write performance.

Read Throughput Since the theoretical time complexity of naive (plain database) relationship counting is linearly proportional to the actual number of relationships, one variable parameter for vertex degree query performance evaluation is the average number of relationships per vertex.

Another important parameter for read throughput measurements is whether the data being accessed is on disk or in one of the caches. Therefore, we perform three kinds of read throughput measurements. First, we run the measurements with both caches disabled. Second, we enable the file system cache, size it correctly so that the entire graph fits into it, and "warm it up" by running the experiment 100 - 10,000 times[4] before actually starting the measured runs. Similarly, we turn on the high level cache for the third kind of measurement, configure it to use "strong" caching policy that never evicts any cached objects, and set the maximum heap size to fit the entire graph in Java-object form. Again, we first warm up the cache before starting any measurements.

Finally, as in the case of write throughput measurements, we vary the number of relationship properties and their change frequency.

Space Requirement Additional space required by the database on disk and in memory due to the relationship count caching depends on a number of factors, particularly the compaction threshold, i.e.,

[3] we perform no explicit benchmarking on changed relationships, because the penalties would be equivalent to the sum of create and delete penalties

[4] depending on the kind of experiment

how many extra properties are allowed to be written to each node, the distribution of vertex degrees across the nodes, and the number and change frequency of relationship properties. Without knowing these graph characteristics, it is difficult to estimate the space requirement. If known, however, the additional space requirement can be determined analytically. Hence, we perform no space-related measurements.

Hardware All experiments have been performed on a Mid-2012 MacBook Air running Mac OS X 10.8.4 with 2GHz Quad-Core Intel Core i7 processor, 8 GB RAM, and an SSD hard drive.

Methodology We have established that the total number of vertices in the graph is not relevant for any of these measurements. Therefore, the number of vertices has been selected so that the total runtime of the tests is long enough to provide a good resolution, but short enough to finish each test in a few minutes. Hence, write and read throughput measurements were performed on graphs with 10 and 100 vertices, respectively.

After caches had been warmed up (if applicable), every write throughput measurement was performed 20 times and every read throughput measurement 100 times[5]. Runtimes were measured in microseconds and are reported as mean values in milliseconds, plus or minus one standard deviation. When comparing two scenarios and reporting a ratio (sometimes as a percentage) of their runtimes, the first run of the first scenario is compared to the first run of the second scenario, second run of the first scenario to the second run of the second scenario, and so on, until the last runs of both scenarios. The result is then the mean of the ratios (percentages), plus or minus one standard deviation. All raw measurement data have been submitted together with the code[6].

6.2.3 Simple Relationship Counting

In the first experiment, the time taken to create 10,000 relationships between random pairs of nodes was measured. None of the relationships had any properties. Figure 6.1 shows a plot of the measured times with log-scaled axes for four different scenarios. The first scenario (purple) shows times taken when using Neo4j without the GraphAware Framework. For the second one (red), the framework was running without any modules. The third one (green) shows the framework running with the `SimpleRelationshipCountModule` and the last one (blue) with `FullRelationshipCountModule`.

A number of observations can be made about this plot. First of all, according to expectations, the framework running without any modules adds very little overhead. Secondly, since relationships have

[5] since reads typically have shorter runtimes

[6] `/neo4j-relcount/src/test/resources/perf`

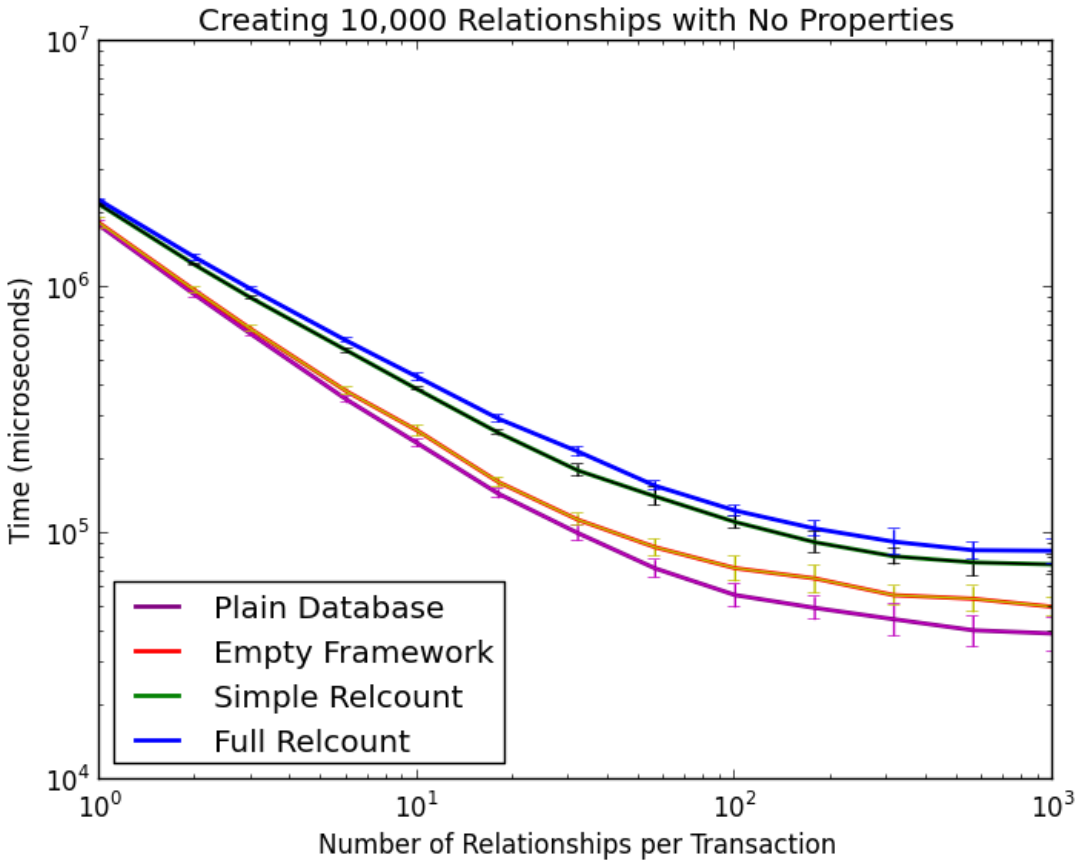

Figure 6.1: Relationship Write Performance (No Properties on Relationships)

no properties, there is a small difference between the two modules, i.e.„ the simple one that ignores properties, and the full one that takes them into account. Finally, GraphAware Modules are adding some amount of overhead when creating relationships, as expected, because they create properties on nodes. Table 6.1 shows the mean runtimes in milliseconds and Table 6.2 quantifies the overhead by showing the throughput (relationships created per unit of time) as a mean percentage of the full throughput of a plain database. In the worst case, the throughput roughly halves.

	Scenario:			
Rels/Tx	No Framework	Empty Framework	Simple Relcount	Full Relcount
1	1782 ± 64	1822 ± 83	2168 ± 16	2255 ± 30
2	931 ± 24	969 ± 31	1232 ± 10	1322 ± 38
3	645 ± 10	676 ± 21	905 ± 13	982 ± 15
6	349 ± 11	377 ± 12	554 ± 11	606 ± 12
10	233 ± 8	262 ± 14	385 ± 6	433 ± 16
18	145 ± 6	162 ± 7	257 ± 7	293 ± 10
32	100 ± 7	114 ± 6	180 ± 10	215 ± 10
56	72 ± 6	88 ± 7	142 ± 12	156 ± 6
100	56 ± 6	72 ± 9	111 ± 7	124 ± 6
178	50 ± 5	66 ± 9	92 ± 9	105 ± 8
316	45 ± 7	56 ± 5	81 ± 6	93 ± 11
562	40 ± 6	54 ± 7	76 ± 9	86 ± 7
1000	39 ± 6	50 ± 5	75 ± 7	85 ± 10

Table 6.1: Times (ms) Taken to Create 10,000 Relationships with No Properties

	Scenario:		
Rels/Tx	Empty Framework	Simple Relcount	Full Relcount
1	$98\% \pm 8\%$	$82\% \pm 3\%$	$79\% \pm 3\%$
2	$96\% \pm 5\%$	$76\% \pm 2\%$	$70\% \pm 3\%$
3	$96\% \pm 4\%$	$71\% \pm 1\%$	$66\% \pm 2\%$
6	$93\% \pm 5\%$	$63\% \pm 2\%$	$58\% \pm 2\%$
10	$89\% \pm 7\%$	$60\% \pm 2\%$	$54\% \pm 2\%$
18	$90\% \pm 6\%$	$57\% \pm 3\%$	$50\% \pm 2\%$
32	$89\% \pm 8\%$	$56\% \pm 4\%$	$47\% \pm 3\%$
56	$83\% \pm 12\%$	$51\% \pm 6\%$	$46\% \pm 4\%$
100	$79\% \pm 12\%$	$51\% \pm 7\%$	$46\% \pm 6\%$
178	$77\% \pm 13\%$	$55\% \pm 8\%$	$48\% \pm 5\%$
316	$81\% \pm 18\%$	$56\% \pm 10\%$	$49\% \pm 8\%$
562	$76\% \pm 19\%$	$54\% \pm 9\%$	$48\% \pm 8\%$
1000	$79\% \pm 17\%$	$53\% \pm 9\%$	$47\% \pm 9\%$

Table 6.2: Throughput when Creating Relationships with No Properties, as a Percentage of Full Throughput

Tables 6.3 and 6.4 show the same experiment for deleting relationships. A plot of this data would have very similar characteristics to the previous one. When deleting relationships, the overhead of the

modules is slightly larger than the overhead when creating relationships.

	Scenario:		
Rels/Tx	No Framework	Simple Relcount	Full Relcount
1	1824 ± 176	2437 ± 17	2516 ± 17
2	951 ± 59	1455 ± 44	1504 ± 10
3	671 ± 42	1081 ± 11	1155 ± 27
6	360 ± 18	681 ± 14	734 ± 21
10	232 ± 14	477 ± 20	521 ± 12
18	141 ± 5	330 ± 10	373 ± 15
32	94 ± 4	232 ± 15	254 ± 7
56	72 ± 9	162 ± 3	183 ± 8
100	55 ± 3	121 ± 2	140 ± 7
178	49 ± 3	99 ± 4	115 ± 8
316	45 ± 6	84 ± 2	97 ± 8
562	43 ± 9	76 ± 2	89 ± 5
1000	40 ± 8	72 ± 5	84 ± 7

Table 6.3: Times (ms) Taken to Delete 10,000 Relationships with No Properties

	Scenario:	
Rels/Tx	Simple Relcount	Full Relcount
1	$75\% \pm 7\%$	$73\% \pm 7\%$
2	$65\% \pm 5\%$	$63\% \pm 4\%$
3	$62\% \pm 4\%$	$58\% \pm 4\%$
6	$53\% \pm 3\%$	$49\% \pm 3\%$
10	$49\% \pm 3\%$	$45\% \pm 3\%$
18	$43\% \pm 2\%$	$38\% \pm 2\%$
32	$41\% \pm 2\%$	$37\% \pm 2\%$
56	$45\% \pm 5\%$	$40\% \pm 5\%$
100	$45\% \pm 3\%$	$39\% \pm 3\%$
178	$49\% \pm 3\%$	$43\% \pm 4\%$
316	$53\% \pm 8\%$	$46\% \pm 8\%$
562	$57\% \pm 11\%$	$49\% \pm 10\%$
1000	$56\% \pm 12\%$	$48\% \pm 10\%$

Table 6.4: Throughput when Deleting Relationships with No Properties, as a Percentage of Full Throughput

In the next experiment, graphs with 10 to 10,000 relationships per node (two different types, no properties) have been created. The times taken to find vertex degrees of 10 randomly selected vertices for a single relationship type have been measured with no caching, low level cache enabled, and high level cache enabled. They are plotted in Figure 6.2. Since there are no properties on relationships `SimpleRelationshipCountModule` has been used. As expected, performance of the module is close to constant, whilst the performance of naive counting is nearly linear. Table 6.5 quantifies how much faster (in vertex degree calculations per unit of time) the database is with the module running. For

instance, degrees of nodes with 1,000 relationships can be computed approximately 126 times faster when data is on disk, but only 6 times faster when in the high level cache. Note that when data is not in the high level cache, vertex degree analysis using the Relationship Count Module is always faster than the naive approach. On the other hand, counting relationships for vertices with fewer than 100 relationships, when data is in high level cache, is more efficient using the naive approach.

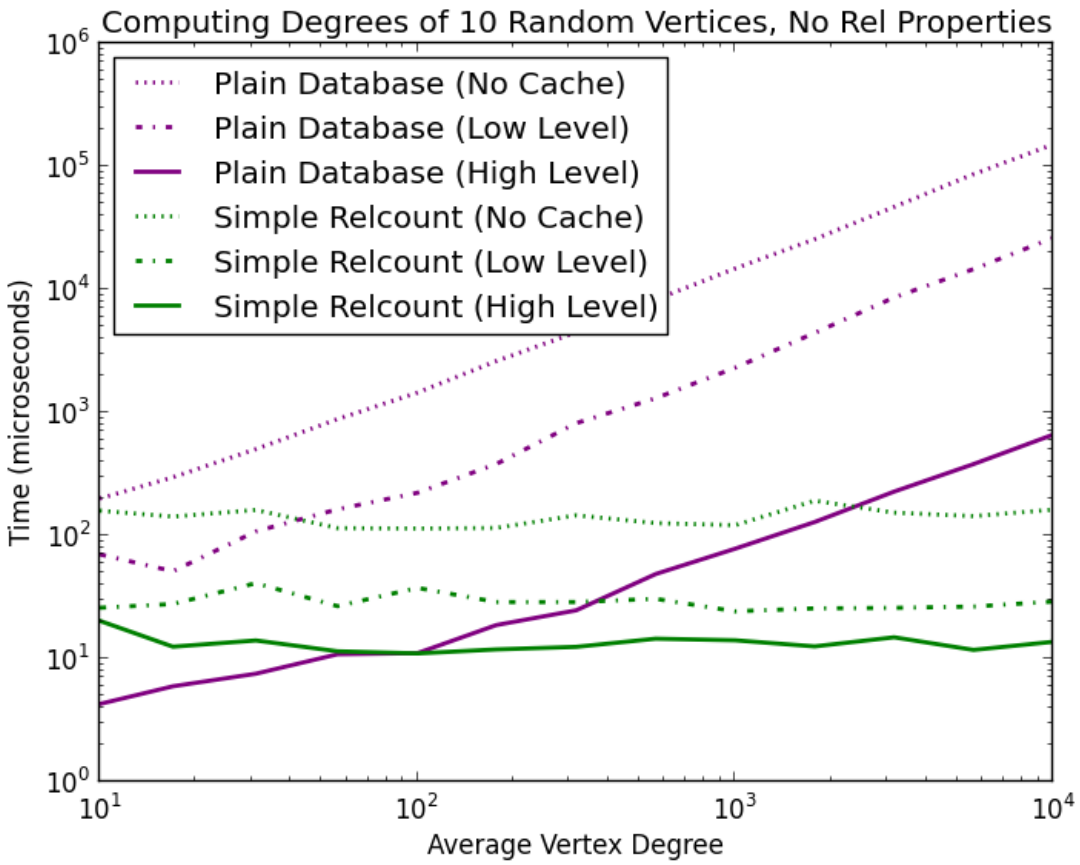

Figure 6.2: Vertex Degree Read Performance (No Properties on Relationships)

	Caching:		
Avg Degree:	No Cache	Low Level	High Level
10	1.29 ± 0.33	2.89 ± 0.77	0.25 ± 0.17
17	2.13 ± 0.49	1.98 ± 0.69	0.53 ± 1.48
31	3.24 ± 0.83	2.60 ± 5.78	0.57 ± 0.22
56	7.75 ± 1.80	6.53 ± 1.88	0.96 ± 0.23
100	12.98 ± 2.83	6.31 ± 2.30	1.03 ± 0.37
177	23.37 ± 5.44	14.44 ± 4.82	1.62 ± 1.49
316	31.15 ± 5.54	30.90 ± 9.77	2.07 ± 0.44
562	68.73 ± 16.67	45.86 ± 12.91	3.79 ± 1.10
1000	126.44 ± 24.69	96.75 ± 17.35	6.45 ± 2.56
1778	141.05 ± 33.46	176.06 ± 28.43	10.85 ± 5.08
3162	311.88 ± 44.67	346.88 ± 77.35	16.79 ± 5.96
5623	642.91 ± 178.30	573.44 ± 97.58	34.36 ± 8.30
10000	985.82 ± 251.90	955.44 ± 204.24	51.18 ± 11.22

Table 6.5: Read Performance as a Speedup Factor, No Properties on Relationships

6.2.4 Full Relationship Counting

Let us now turn our attention to relationships with properties. To measure the write throughput penalty, we repeated the experiments from previous section three more times. On the first repetition, every relationship has two properties, both of which only take on two different values. Since there are two relationship types and two directions, the total number of unique relationship descriptions is $2x2x2x2 = 16$, whilst the compaction threshold is 20. Therefore, no compaction takes place. Second, every relationship has two properties, one of which takes on two different values, whilst the other one changes for every relationship. Therefore, compaction will occur. Finally, every relationship has four different properties, one of which takes on four different values, whilst the other three change for every relationship. Where GraphAware Framework is used, it is with `FullRelationshipCountModule`.

No Compaction Figure 6.3 and Tables 6.6 and 6.7 show the write throughput when relationship properties are involved, but no compaction occurs. In the worst case, the write throughput drops to about a fifth of its original value.

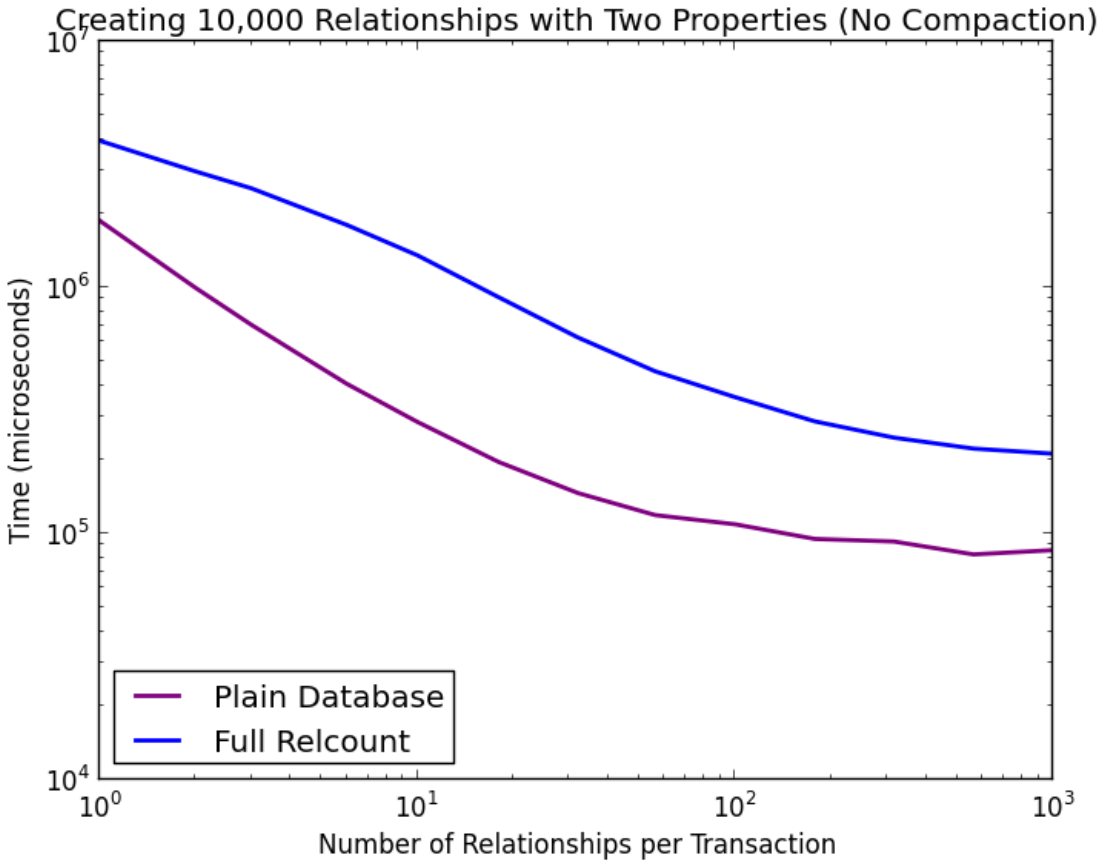

Figure 6.3: Relationship Write Performance (Two Properties on Relationships, No Compaction)

	Scenario:	
Rels/Tx	No Framework	Full Relcount
1	1866 ± 55	3944 ± 104
2	997 ± 23	2962 ± 40
3	705 ± 16	2528 ± 14
6	405 ± 18	1793 ± 12
10	284 ± 10	1350 ± 15
18	195 ± 10	913 ± 13
32	146 ± 12	626 ± 9
56	119 ± 9	456 ± 10
100	109 ± 13	358 ± 40
178	95 ± 7	286 ± 9
316	93 ± 14	245 ± 16
562	82 ± 7	221 ± 13
1000	86 ± 6	211 ± 14

Table 6.6: Times (ms) Taken to Create 10,000 Relationships, Two Properties Each, No Compaction

	Scenario:
Rels/Tx	Full Relcount
1	$47\% \pm 2\%$
2	$34\% \pm 1\%$
3	$28\% \pm 1\%$
6	$23\% \pm 1\%$
10	$21\% \pm 1\%$
18	$21\% \pm 1\%$
32	$23\% \pm 2\%$
56	$26\% \pm 2\%$
100	$31\% \pm 4\%$
178	$33\% \pm 2\%$
316	$38\% \pm 6\%$
562	$37\% \pm 4\%$
1000	$41\% \pm 4\%$

Table 6.7: Throughput when Creating Relationships with Two Properties (No Compaction), as a Percentage of Full Throughput

Compaction When compaction takes place, the throughput suffers even more. With increasing number of relationship properties, write throughput further decreases, due to compaction. Figure 6.4 illustrates the performance when creating 1,000 relationships. In scenario A, the framework is not running and each relationship has two properties, whilst in scenario B, each relationship has four properties. Scenarios C and D measure the creation of the first 1,000 and the next 1,000 relationships, respectively, where each relationship has two properties and the Full Relationship Count Module is running. Finally, each relationship has four properties in scenarios E and F, again measuring the creation of the first thousand and the second thousand relationships.

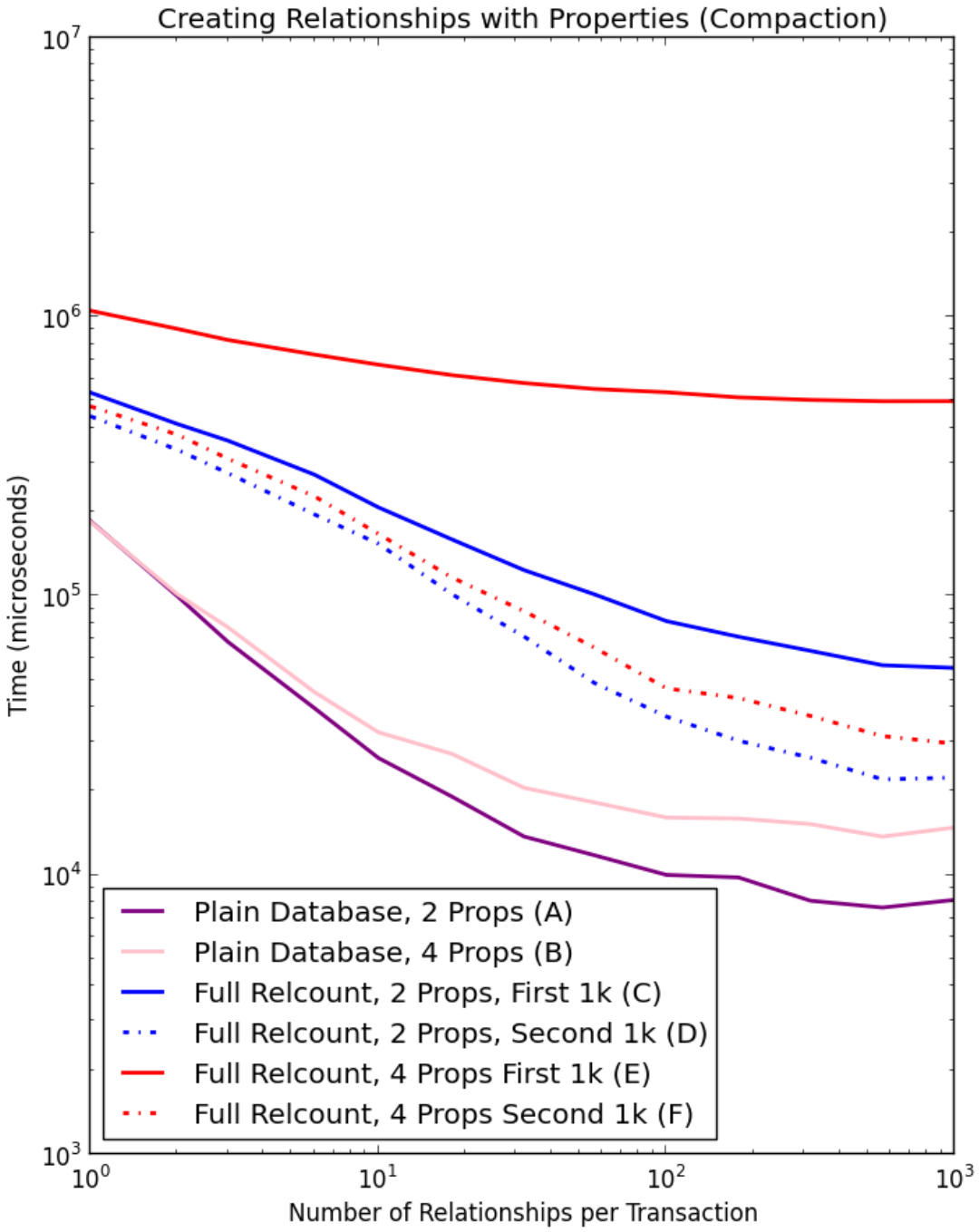

Figure 6.4: Relationship Write Performance (Properties on Relationships, with Compaction)

It is apparent from Figure 6.4 that compaction causes significant performance degradation. However, once the system is in a “steady state", i.e., all the frequently changing properties have been compacted out from cached relationship counts, the write throughput increases again to the levels experienced in no-compaction scenarios. Tables 6.8 and 6.9 show the measurements for these experiments. In the absolute worst case, creating first 1,000 relationships with four properties each, three of which are changing all the time, can be about 30 times slower when running the Relationship Count Module, compared to a plain database. However, once in a steady state, the throughput is again up to a fifth of its original value.

	Scenario:					
Rels/Tx	A	B	C	D	E	F
1	188.8 ± 17.1	184.9 ± 6.9	544.1 ± 64.9	443.9 ± 62.5	1059.3 ± 81.0	482.1 ± 45.3
2	101.8 ± 8.3	101.6 ± 5.2	417.9 ± 32.9	336.2 ± 27.4	910.4 ± 46.3	381.7 ± 44.4
3	69.5 ± 6.0	77.0 ± 8.3	363.3 ± 30.8	276.6 ± 12.6	830.5 ± 27.9	311.5 ± 24.2
6	40.0 ± 2.4	45.1 ± 6.2	276.5 ± 26.2	196.1 ± 11.6	736.2 ± 27.4	228.4 ± 22.7
10	26.7 ± 2.1	32.4 ± 3.7	208.8 ± 9.9	154.2 ± 10.4	675.9 ± 20.9	166.4 ± 13.9
18	19.2 ± 5.2	27.0 ± 7.2	159.4 ± 11.5	101.6 ± 9.4	618.8 ± 23.1	118.0 ± 11.5
32	13.9 ± 1.3	20.6 ± 3.6	124.8 ± 7.5	71.7 ± 8.1	579.9 ± 16.2	87.9 ± 17.5
56	11.8 ± 3.2	18.1 ± 4.9	103.0 ± 10.4	49.7 ± 5.5	552.2 ± 25.7	64.9 ± 9.3
100	10.1 ± 2.5	16.0 ± 4.5	81.7 ± 6.5	37.1 ± 4.9	535.5 ± 22.7	46.7 ± 6.9
178	9.8 ± 5.5	15.8 ± 4.2	72.1 ± 8.9	30.3 ± 3.8	514.1 ± 30.9	43.2 ± 8.9
316	8.2 ± 0.8	15.1 ± 4.7	63.4 ± 6.4	26.2 ± 2.8	505.0 ± 15.1	36.8 ± 7.5
562	7.7 ± 1.2	13.8 ± 4.0	56.5 ± 5.8	22.5 ± 3.3	497.6 ± 25.0	31.5 ± 4.0
1000	8.1 ± 1.9	14.7 ± 3.2	55.0 ± 8.1	22.4 ± 5.8	497.9 ± 43.7	30.3 ± 6.6

Table 6.8: Times (ms) Taken to Create 1,000 Relationships, with Compaction

	Scenario:			
Rels/Tx	C	D	E	F
1	34.9% ± 3.0%	43.1% ± 5.0%	17.5% ± 1.1%	38.6% ± 3.1%
2	24.4% ± 1.5%	30.4% ± 2.5%	11.2% ± 0.7%	26.9% ± 2.9%
3	19.2% ± 1.9%	25.2% ± 2.0%	9.3% ± 1.1%	24.9% ± 3.6%
6	14.5% ± 1.0%	20.4% ± 1.5%	6.1% ± 0.9%	20.0% ± 3.6%
10	12.8% ± 0.9%	17.4% ± 1.6%	4.8% ± 0.6%	19.6% ± 2.8%
18	12.0% ± 2.8%	19.0% ± 5.4%	4.4% ± 1.2%	23.3% ± 7.4%
32	11.2% ± 0.9%	19.7% ± 2.5%	3.5% ± 0.6%	24.0% ± 4.9%
56	11.6% ± 3.2%	24.0% ± 6.9%	3.3% ± 1.0%	28.5% ± 9.3%
100	12.4% ± 3.1%	27.5% ± 6.9%	3.0% ± 0.9%	35.0% ± 11.5%
178	14.0% ± 9.2%	32.9% ± 20.2%	3.1% ± 0.8%	37.9% ± 11.7%
316	13.0% ± 2.0%	31.4% ± 3.9%	3.0% ± 1.0%	42.6% ± 17.0%
562	13.9% ± 2.7%	34.7% ± 4.4%	2.8% ± 0.9%	44.2% ± 12.0%
1000	14.9% ± 3.4%	37.0% ± 7.9%	3.0% ± 0.7%	50.2% ± 13.6%

Table 6.9: Throughput when Creating Relationships with Compaction, as a Percentage of Full Throughput

Software performance optimisation is a long process; once a bottleneck is removed, another one appears. After we have optimised the database access, compaction clearly became the bottleneck, according to the profiler and the experiment presented above. The main reason for this is the fact that for a relationship with x different properties, there are 2^x more general or equal representations than its most specific representation, all of which are generated first, before being scored. This is one area of the software that could potentially be improved from performance point of view by employing some kind of score-based heuristic and only generating a few generalisations. Another way to avoid performance problems is using a custom `RelationshipPropertyInclusionStrategy` and excluding properties that are known to be frequently changing from the caching process altogether, treating compaction only as a safety mechanism for keeping the number of node properties low.

Finally, let us take a look at the performance benefits of the Full Relationship Count Module when analysing vertex degrees, a plot of which is presented in Figure 6.5. Table 6.10 shows the speedup factors.

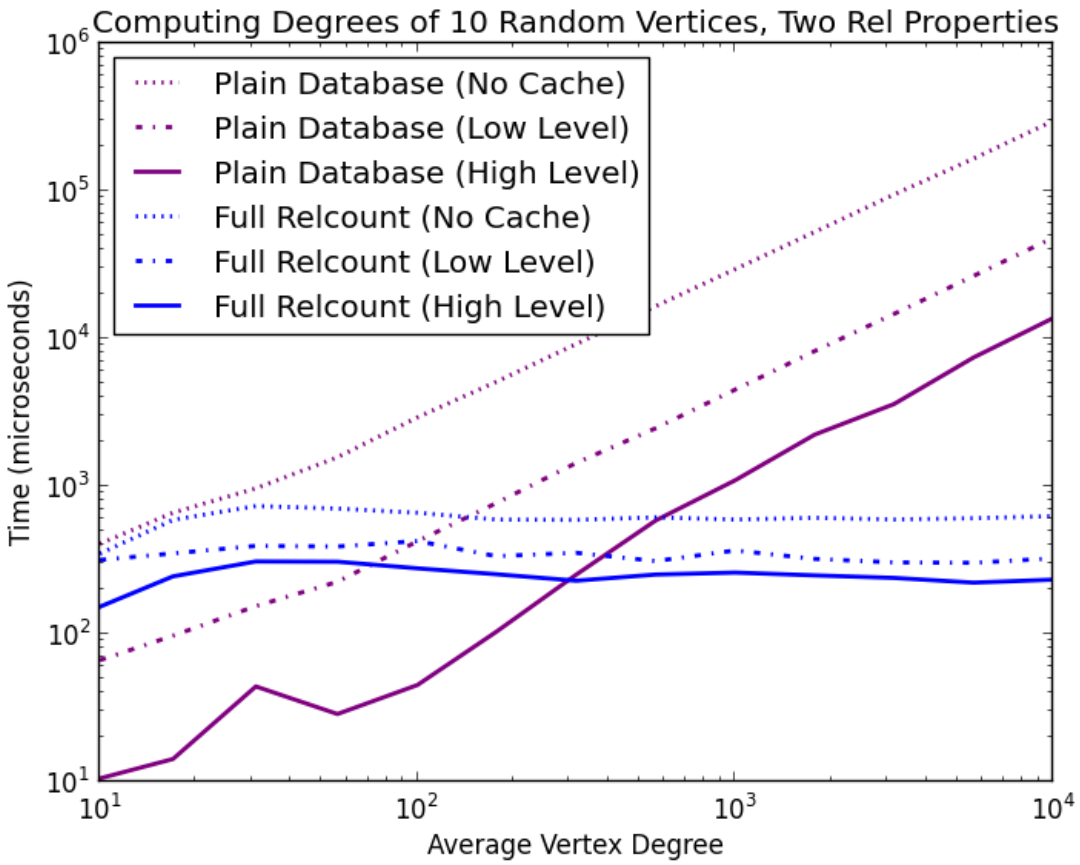

Figure 6.5: Vertex Degree Read Performance (Two Properties on Relationships)

	Caching:		
Avg Degree	No Cache	Low Level	High Level
10	1.25 ± 0.37	0.23 ± 0.12	0.07 ± 0.03
17	1.16 ± 0.36	0.30 ± 0.11	0.06 ± 0.03
31	1.39 ± 0.36	0.40 ± 0.16	0.15 ± 0.15
56	2.27 ± 0.39	0.60 ± 0.14	0.10 ± 0.02
100	4.50 ± 0.87	1.07 ± 0.33	0.17 ± 0.04
177	8.67 ± 1.28	2.39 ± 0.76	0.42 ± 0.09
316	15.70 ± 1.99	4.39 ± 1.55	1.11 ± 0.19
562	27.50 ± 3.98	8.16 ± 1.65	2.41 ± 0.56
1000	50.09 ± 6.17	13.10 ± 3.44	4.50 ± 1.09
1778	88.97 ± 13.81	26.49 ± 4.92	9.34 ± 1.87
3162	160.99 ± 19.13	49.25 ± 6.77	15.58 ± 2.65
5623	277.84 ± 34.43	88.58 ± 9.81	33.73 ± 3.41
10000	487.95 ± 73.04	153.82 ± 25.17	60.53 ± 8.25

Table 6.10: Read Performance as a Speedup Factor, Two Properties on Relationships

When the entire graph is in high level cache, the Relationship Count Module only outperforms naive counting for vertices with more than 300 relationships. For vertices with 10,000 relationships, the count can be 60 times faster. The real benefit of relationship count caching is shown in scenarios where the graph does not fit into the object cache. Reading the cached counts starts to be faster than the naive approach around the 100 mark and for a vertex with 1,000 relationships, it is 13 times faster.

Furthermore, in a real life application with high level cache enabled but not large enough to fit the entire graph, counting relationships by traversing all of them will generate a lot of garbage collection overhead, since each traversed relationship becomes a cached Java object. When using cached counts, however, this will not be the case, since no relationships are actually traversed. This could result in significantly higher performance improvements than the ones presented here, measured in artificially set up testing conditions that intentionally avoid any garbage collection.

Finally, the number of read operations in many Neo4j applications will be very large, compared to the number of write operations, rendering read performance improvements much more important than write throughput penalties.

Chapter 7

Conclusion and Future Work

7.1 Achievements

This project introduces the problem of online analytical processing in OLTP graph databases that use the property graph data model and reviews existing related work, which is relatively scarce. A seemingly simple problem, analysing vertex degrees in a property graph, has been selected and its relevance to other graph analytics has been described. The problem has been used to explore the challenge of storing metadata about a changing property graph and keeping them up to date, a technique that might be useful for other kinds of graph analytical problems.

Along the way, a number of definitions, theorems, and proofs have been created, exploring the problem of vertex degree analysis in depth. Specifically, a formal definition of vertex degree in a property graph has been introduced, followed by partial general-to-specific ordering of edge representations, a caching operation, and a cache compaction operation, enabled by a heuristic based on the change frequency of edge property values.

The architecture and implementation of Neo4j, an open-source OLTP property-graph database, have been studied and described. Based on the aforementioned theoretical foundation, a software module for Neo4j, called the Relationship Count Module, has then been developed from scratch in two flavours, “simple” and “full".

Throughout the design and development process, generic functionality that could be useful to other modules with similar purpose has been identified and implemented as the GraphAware Framework. Consequently, the framework provides support for the Relationship Count Module as well as a convenient starting point for future module developers.

Performance testing has shown that, depending on the specific use case and database usage patterns,

as little as 20% of the write throughput can be traded off for several orders of magnitude performance increase when analysing vertex degrees. Furthermore, the work provides a starting point for theoretical analysis and practical development of solutions to other graph database challenges. Finally, by demonstrating the complexity of a seemingly simple problem and quantifying some of the achievements, it serves as a small step on the long journey towards online analytical processing in graph databases.

Naturally, there are also a few limitations of the current implementation. First of all, as mentioned before, performance could always be improved. Whilst theoretically sound, self-generalising relationship representations would benefit from an informed search through the space of generalisations during compaction. Additionally, some design decisions have diminished the flexibility of the solution. For example, relationships can only be cached based on a property being equal to certain value, not greater, smaller, in a specific range, etc. Also, the solution does not provide a way to only count loops. Finally, for some use cases, the solution might be inappropriate altogether, especially due to the choice of "sacrificing" most frequently changing properties first.

7.2 Applications

This project has resulted in a documented, fully tested and working, production-ready framework and an easily customisable module with a clean and simple API, which can be applied to use cases that require speedy vertex degree analysis, trading off some write throughput.

The framework also enables the development of other analytical modules as well as modules with different purposes. For instance, it could be used by development teams to create domain-specific or company-specific Neo4j extensions.

Finally, the framework can serve as a library of useful, tested code that can benefit any developer working with Neo4j, since it provides functionality and APIs that often need to be written from scratch when building software based on the graph database. Many of these routines get no attention in the main body of the report, since they are straightforward implementations. They can, however, be used to reduce development effort and the risk of errors. Hence, they are mentioned in Appendix B.

7.3 Future Work

Apart from already proposed potential improvements to the Relationship Count Module, the amount of future work that can be based on this project is extensive.

In the analytics space, metadata-based operations on changing property graphs, such as roll-ups, drill-downs, slicing and dicing could be analysed and prototyped. One path that has not been explored and could well be undertaken is performing random walks on property graphs to approximate results of certain analytical queries. The framework would have to be extended to support this functionality. Also, to prevent write throughput penalties, one could investigate asynchronous metadata maintenance outside of the main transactional processing.

Analytics aside, modules with other purposes could be written for the framework. An example could be a "Temporal Graph Database", which never actually deletes any data, but marks them obsolete with a timestamp. It could then provide temporal queries and/or historical graph analysis. Another example of a useful module would be type and/or property-based relationship indexing module, which would allow for fast random relationship selection and performant paging (offsets and limits) during relationship traversals. Yet another module could provide the ability to define a graph schema and enforce it by not allowing commits that result in schema violations.

All code written for this project is open-source and licensed under the GNU General Public Licence (GPL). It is our aim to maintain, support, and improve the framework, so it continues to be useful for graph database adopters. We invite interested individuals and teams to use it free of charge under GPL and we welcome all contributions.

Bibliography

[1] G. Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Pubs Co Series. Manning Publications Company, 2011. ISBN 9781617290084.

[2] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *Proceedings of the 5th international conference on Algorithms and models for the web-graph*, WAW'07, pages 124–137, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-77003-8, 978-3-540-77003-9.

[3] K. Beck. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, 2003. ISBN 9780321146533.

[4] Ulrik Brandes. A faster algorithm for betweenness centrality*. *The Journal of Mathematical Sociology*, 25(2):163–177, June 2001. ISSN 0022-250X. doi: 10.1080/0022250X.2001.9990249.

[5] Ulrik Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, May 2008. ISSN 03788733. doi: 10.1016/j.socnet.2007.11.001.

[6] Ulrik Brandes and Daniel Fleischer. Centrality measures based on current flow. In *Proceedings of the 22nd annual conference on Theoretical Aspects of Computer Science*, STACS'05, pages 533–544, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24998-2, 978-3-540-24998-6. doi: 10.1007/978-3-540-31856-9_44.

[7] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. Graph OLAP: Towards Online Analytical Processing on Graphs. In *2008 Eighth IEEE International Conference on Data Mining*, pages 103–112, Pisa, December 2008. IEEE. ISBN 978-0-7695-3502-9. doi: 10.1109/ICDM.2008.30.

[8] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970. ISSN 00010782. doi: 10.1145/362384.362685.

[9] TM Connolly and CE Begg. *Database systems: a practical approach to design, implementation, and management*. Addison Wesley, 5 edition, 2005. ISBN 978-0321523068.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2nd edition, 2001. ISBN 0-262-03293-7, 9780262032933.

[11] Reinhard Diestel. *Graph Theory {Graduate Texts in Mathematics; 173}*. Springer, 3rd ed. edition, 2000. ISBN 978-3540261834.

[12] Martin Everett and Stephen P. Borgatti. Ego network betweenness. *Social Networks*, 27(1):31–38, January 2005. ISSN 03788733. doi: 10.1016/j.socnet.2004.11.007.

[13] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3): 215–239, January 1978. ISSN 03788733. doi: 10.1016/0378-8733(78)90021-7.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[15] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In J. Ian Munro and Dorothea Wagner, editors, *ALENEX*, pages 90–100. SIAM, 2008.

[16] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[17] M. Newman and M Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, February 2004. ISSN 1539-3755. doi: 10.1103/PhysRevE.69.026113.

[18] Carlos Ordonez and Javier García-García. Evaluating join performance on relational database systems. *JCSE*, 4(4):276–290, 2010.

[19] Qiang Qu, Feida Zhu, Xifeng Yan, Jiawei Han, SY Philip, and Hongyan Li. Efficient topological OLAP on information networks. *Database Systems for Advanced Applications*, Volume Par:389–403, 2011. doi: 10.1007/978-3-642-20149-3_29.

[20] Matthew J. Rattigan, Marc Maier, and David Jensen. Using structure indices for efficient approximation of network properties. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06*, page 357, New York, New York, USA, 2006. ACM Press. ISBN 1595933395. doi: 10.1145/1150402.1150443.

[21] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013. ISBN 9781449356262.

[22] P.J. Sadalage and M.J. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Always learning. Addison Wesley Professional, 2012. ISBN 9780321826626.

[23] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 567–580, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376675.

[24] JD Ullman, H Garcia-Molina, and J Widom. *Database systems: the complete book*. Prentice Hall, 2 edition, 2001. ISBN 978-0131873254.

[25] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, KDD '03, page 266, New York, New York, USA, 2003. ACM Press. ISBN 1581137370. doi: 10.1145/956750.956782.

[26] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*, page 853, New York, New York, USA, 2011. ACM Press. ISBN 9781450306614. doi: 10.1145/1989323.1989413.

Appendices

Appendix A

Symbols Reference

This appendix is an organised summary of symbols used throughout the report and informal explanation thereof. It is here for readers' convenience, especially as a reference for reading Chapter 4. Should there be any ambiguity, the definitions in the main text shall be used.

α	an attribute key, such that $\alpha \in A$
A	the set of attribute keys
β	an attribute value, such that $\beta \in B$
B	the set of attribute values
$attr$	a total map $C \times A \rightarrow B$, assigning every property container $c \in C$ and an attribute key $\alpha \in A$ a value $\beta \in (B \cup \{UNDEF\})$
$attrp$	a partial map $C \times A \nrightarrow B$, assigning a property container $c \in C$ and an attribute key $\alpha \in A$ a value $\beta \in B$, if defined for c and α.
c	a property container (vertex or edge), such that $c \in C$
C	the set of property containers, i.e., $C = (V \cup E)$
$COMPACT(K(v))$	a cache compaction operation of cache $K(v)$
γ	a predicate on an attribute value β, such that $\gamma \in \Gamma$ and $\gamma(\beta) \rightarrow \{true, false\}$
Γ	the set of predicates on property values of a property container c in a properties description ϕ
$d(v)$	the degree of vertex v, equivalent to $\lvert E(v) \rvert$
$d_{in}(v)$	the indegree of vertex v, equivalent to $\lvert E_{in}(v) \rvert$
$d_{out}(v)$	the outdegree of vertex v, equivalent to $\lvert E_{out}(v) \rvert$
δ	a partial map $(E \times V) \nrightarrow (\Delta \cup \{LOOP\})$, mapping each vertex and edge at that vertex to a direction
$\delta(e, v)$	the direction of e from the point of view of v; it is one of $(\Delta \cup \{LOOP\})$

δ_ψ	an edge direction (one of $\{IN, OUT\}$) in a relationship description ψ
$\Delta(e, v)$	the set of edge directions $\{IN, OUT\}$
e	an edge, such that $e \in E$
E	the set of edges of a graph G
$E(v)$	the set of edges at a vertex v
$E_{in}(v)$	the set of incoming edges at a vertex v
$E_{out}(v)$	the set of outgoing edges at a vertex v
G	a graph, i.e., a pair $G = (V, E)$ (undirected) or $G = (V, E, init, ter)$ (directed)
$init$	a map $E \to V$, assigning every edge $e \in E$ an initial (or start) vertex (or tail) v
$init(e)$	the initial vertex of edge e
$\iota(K(v), \psi, \alpha)$	informational value of a property with key α on a relationship description ψ with respect to cache $K(v)$
$\iota(K(v), \psi)$	the informational value of a relationship description ψ with respect to cache $K(v)$
k	cached vertex degree, such that $k \in K(v) = (\psi \in \Psi, n \in \mathbb{N})$
$K(v)$	vertex degree cache at vertex v
$label$	a map $E \to \Lambda$, assigning every edge $e \in E$ a label $\lambda \in \Lambda$
$label(e)$	the label of edge e
λ	an edge label, such that $\lambda \in \Lambda$
λ_ψ	an edge label $\lambda_\psi \in \Lambda$ in a relationship description ψ
Λ	the set of edge labels
$LOOP$	the direction of a relationship, which is a loop
$matches(c, \phi)$	a predicate meaning that the properties of property container c match the properties description ϕ
$matches(\delta(e, v), \delta_\psi)$	a predicate meaning that the direction of edge e at vertex $v$matches the direction δ_ψ of relationship description ψ
$matches(e, \phi)$	a predicate meaning that the properties of edge e match the properties description ϕ
n	a natural number $n \in \mathbb{N}$ (including 0), indicating the degree in a cached degree k
P	a path through a graph
PCF	Property Frequency Change function, used to sort Ψ_{sg}
$\Pi(c)$	the set of key-value pairs assigned to a property container c; value can be $UNDEF$
ϕ	a properties description; a map $A \to \Gamma$, assigning every attribute key $\alpha \in A$ a predicate $\gamma \in \Gamma$
ϕ_{maxg}	the most general properties description
$\phi_{maxs}(c)$	the most specific properties description for property container $c \in C$
Φ	a set of properties descriptions

ψ	a relationship description in the form $\psi = (\delta_\psi, \lambda_\psi, \phi)$
Ψ	the set of all relationship descriptions
Ψ_g	the generalisation set of Ψ
Ψ_{sg}	the generalisation superset, i.e., a set of all Ψ_g for all Ψ in all $k \in K(v)$
$\psi_{maxg}(e, v)$	the most general relationship description for relationship e at node v
$\psi_{maxs}(e, v)$	the most specific relationship description for relationship e at node v
$\rho(K(v), \lambda, \alpha)$	number of distinct values of property α on relationship descriptions in cache $K(v)$ with type λ
$\rho_{avg}(K(v), \lambda, \alpha)$	average ρ across relationship descriptions in cache $K(v)$ with type λ
τ	compaction threshold for cache compaction
ter	a map $E \rightarrow V$, assigning every edge $e \in E$ a terminal (or end) vertex (or head) v
$ter(e)$	the terminal vertex of edge e
UNDEF	a value representing undefined attribute on a property container
v	a vertex, such that $v \in V$
V	the set of vertices of a graph G
?	a predicate, such that $\forall \beta \in (B \cup \{UNDEF\})$, $?(\beta) \rightarrow true$

Appendix B

GraphAware Framework User Guide

This user guide has been created for the GraphAware Framework. It is intended as a public manual for GraphAware users and does not assume familiarity with any other part of this report. It does, on the other hand, require readers to have a basic understanding of Neo4j, Java, and Maven. It covers most of the framework functionality from a developer's point of view; some of this functionality might be irrelevant for the purposes of this report and is, therefore, not mentioned in the main text at all. This user guide as well as all the code can be downloaded from https://github.com/graphaware.

B.1 GraphAware Framework

The aim of the GraphAware Framework is to speed-up development with Neo4j and provide useful generic and domain-specific modules, analytical capabilities, graph algorithm libraries, etc.

The framework can be used in two ways: as a library of useful tested code that simplifies tasks commonly needed when developing with Neo4j, or as a real framework with modules, which perform some (behind-the-scenes) mutations on the graph as transactions occur. In the latter case, you get the benefit of the "library" as well.

B.1.1 Download

Releases Releases are synced to Maven Central repository. To use the latest release, download it and put it on your classpath. When using Maven, include the following snippet in your pom.xml:

```
<dependencies>
    ...
```

```
    <dependency>
        <groupId>com.graphaware</groupId>
        <artifactId>neo4j-framework</artifactId>
        <version>1.9-1.7</version>
    </dependency>
    ...
</dependencies>
```

Snapshots To use the latest development version, just clone the Github repository, run `mvn clean install` and put the produced .jar file (found in target) into your classpath. If using Maven for your own development, include the following snippet in your pom.xml instead of copying the .jar:

```
<dependencies>
    ...
    <dependency>
        <groupId>com.graphaware</groupId>
        <artifactId>neo4j-framework</artifactId>
        <version>1.9-1.8-SNAPSHOT</version>
    </dependency>
    ...
</dependencies>
```

Note on Versioning Scheme The version number has two parts, separated by a dash. The first part indicates compatibility with a Neo4j version. The second part is the version of the framework. For example, version 1.9-1.2 is a 1.2 version of the framework compatible with Neo4j 1.9.x

B.1.2 Framework Usage

Using the framework is very easy. Instantiate it, register desired modules, and start it. For example:

```
GraphDatabaseService database
  = new TestGraphDatabaseFactory().newImpermanentDatabase(); //replace with
     your own permanent database

GraphAwareFramework framework = new GraphAwareFramework(database);

framework.registerModule(new SomeModule());
framework.registerModule(new SomeOtherModule());

framework.start();

//use database as usual
```

So far, the only production-ready GraphAware module is the relationship count module. To write your own module, read the Javadoc on `GraphAwareModule` and `GraphAwareFramework`. It would also be useful to read the rest of this guide to understand the ideas and design decisions behind the framework better.

B.1.3 Framework Usage (Batch)

For populating a database quickly, people sometimes use Neo4j `BatchInserters`. The framework can be used with those as well, in the following fashion:

```
TransactionSimulatingBatchInserter batchInserter
  = new TransactionSimulatingBatchInserterImpl("/path/to/your/db");

GraphAwareFramework framework = new GraphAwareFramework(batchInserter);

framework.registerModule(new SomeModule());
framework.registerModule(new SomeOtherModule());

framework.start();

//use batchInserter as usual
```

B.1.4 Configuration

In the above examples, the framework is used with sensible default configuration. At the moment, the only thing that is configurable on the framework level is the character/string that the framework is using to delimit information in its internal metadata secretly written into the graph. By default, this separator is the hash character (#). In the unlikely event of interference with your application logic (e.g. # is used in property keys or values in your application), this can be changed.

If, for instance, you would like to use the dollar sign ($) as a delimiter instead, instantiate the framework in the following fashion:

```
GraphDatabaseService database
  = new TestGraphDatabaseFactory().newImpermanentDatabase();

FrameworkConfiguration customFrameworkConfig
  = new BaseFrameworkConfiguration() {
    @Override
    public String separator() {
      return "$";
    }
};

GraphAwareFramework framework
  = new GraphAwareFramework(database, customFrameworkConfig);

framework.registerModule(new SomeModule());
framework.registerModule(new SomeOtherModule());
```

```
framework.start();
```

B.1.5 Features

Whether or not you use the framework as described above, you can always add it as a dependency and take advantage of its useful features.

Simplified Transactional Operations

Every mutating operation in Neo4j must run within the context of a transaction. The code dealing with that typically involves try-catch blocks and looks something like this:

```
try {
  //do something useful, can throw a business exception
  tx.success();
} catch (RuntimeException e) {
  //deal with a business exception
  tx.failure();
} finally {
  tx.finish(); //can throw a database exception
}
```

GraphAware provides an alternative, callback-based API called `TransactionExecutor` in `com.graphaware.tx.executor`. `SimpleTransactionExecutor` is a simple implementation thereof and can be used on an instance-per-database basis. Since you will typically run a single in-process database instance, you will also only need a single `SimpleTransactionExecutor`.

To create an empty node in a database, you would write something like this.

```
GraphDatabaseService database
  = new TestGraphDatabaseFactory().newImpermanentDatabase();

TransactionExecutor executor = new SimpleTransactionExecutor(database);

executor.executeInTransaction(new VoidReturningCallback() {
  @Override
  public void doInTx(GraphDatabaseService database) {
    database.createNode();
  }
});
```

You have the option of selecting an `ExceptionHandlingStrategy`. By default, if an exception occurs, the transaction will be rolled back and the exception re-thrown. This is true for both application/business exceptions (i.e., the exception your code throws in the `doInTx` method above), and Neo4j exceptions (e.g. constraint violations). This default strategy is called `RethrowException`.

The other available implementation of `ExceptionHandlingStrategy` is `KeepCalmAndCarryOn`. It still rolls back the transaction in case an exception occurs, but it does not re-throw it (only logs it). To use a different `ExceptionHandlingStrategy`, perhaps one that you implement yourself, just pass it in to the `executeInTransaction` method:

```
executor.executeInTransaction(transactionCallback,
  KeepCalmAndCarryOn.getInstance());
```

Batch Transactional Operations

It is a common requirement to execute operations in batches. For instance, you might want to populate the database with data from a CSV file, or just some generated dummy data for testing. If there are many such operations (let's say 10,000 or more), doing it all in one transaction is not the most memory-efficient approach. On the other hand, a new transaction for each operation results in too much overhead. For some use cases, `BatchInserters` provided by Neo4j suffice. However, operations performed using these do not run in transactions and have some other limitations (such as no node/relationship delete capabilities). GraphAware can help here with `BatchTransactionExecutors`. There are a few of them:

Input-Based Batch Operations If you have some input, such as lines from a CSV file or a result of a Neo4j traversal, and you want to perform an operation for each item of such an input, use `IterableInputBatchTransactionExecutor`. As the name suggests, the input needs to be in the form of an `Iterable`. Additionally, you need to define a `UnitOfWork`, which will be executed against the database for each input item. After a specified number of batch operations have been executed, the current transaction is committed and a new one started, until we run out of input items to process.

For example, if you were to create a number of nodes from a list of node names, you would do something like this:

```
GraphDatabaseService database
  = new TestGraphDatabaseFactory().newImpermanentDatabase();

List<String> nodeNames = Arrays.asList("Name1", "Name2", "Name3");  //there
    will be many more

int batchSize = 10;
BatchTransactionExecutor executor
  = new IterableInputBatchTransactionExecutor<>(database, batchSize, nodeNames
      , new UnitOfWork<String>() {
    @Override
    public void execute(GraphDatabaseService database, String nodeName, int
        batchNumber, int stepNumber) {
      Node node = database.createNode();
      node.setProperty("name", nodeName);
    }
});
```

```
executor.execute();
```

Batch Operations with Generated Input or No Input In case you wish to do something input-independent, for example just generate a number of nodes with random names, you can use the `NoInputBatchTransactionExecutor`.

First, you would create an implementation of `UnitOfWork<NullItem>`, which is a unit of work expecting no input:

```
/**
* Unit of work that creates an empty node with random name. Singleton.
*/
public class CreateRandomNode implements UnitOfWork<NullItem> {
  private static final CreateRandomNode INSTANCE = new CreateRandomNode();

  public static CreateRandomNode getInstance() {
    return INSTANCE;
  }

  private CreateRandomNode() {}

  @Override
  public void execute(GraphDatabaseService database, NullItem input, int
      batchNumber, int stepNumber) {
    Node node = database.createNode();
    node.setProperty("name", UUID.randomUUID());
  }
}
```

Then, you would use it in `NoInputBatchTransactionExecutor`:

```
//create 100,000 nodes in batches of 1,000:
int batchSize = 1000;
int noNodes = 100000;

BatchTransactionExecutor batchExecutor = new NoInputBatchTransactionExecutor(
    database, batchSize, noNodes, CreateRandomNode.getInstance());

batchExecutor.execute();
```

Multi-Threaded Batch Operations If you wish to execute any batch operation using more than one thread, you can use the `MultiThreadedBatchTransactionExecutor` as a decorator of any `BatchTransactionExecutor`. For example, to execute the above example using 4 threads:

```
int batchSize = 1000;
int noNodes = 100000;

BatchTransactionExecutor batchExecutor = new NoInputBatchTransactionExecutor(
    database, batchSize, noNodes, CreateRandomNode.getInstance());

BatchTransactionExecutor multiThreadedExecutor = new
    MultiThreadedBatchTransactionExecutor(batchExecutor, 4);

multiThreadedExecutor.execute();
```

Improved Transaction Event API

In `com.graphaware.tx.event`, you will find a decorator of the Neo4j Transaction Event API (called `TransactionData`). Before a transaction commits, the improved API allows users to traverse the new version of the graph (as it will be after the transaction commits), as well as a "snapshot" of the old graph (as it was before the transaction started). It provides a clean API to access information about changes performed by the transaction as well as the option to perform additional changes.

The least you can gain from using this functionality is avoiding `java.lang.IllegalStateException: Node/Relationship has been deleted in this tx` when trying to access properties of nodes/relationships deleted in a transaction. You can also easily access relationships/nodes that were changed and/or deleted in a transaction, again completely exception-free.

Usage To use the API, simply instantiate one of the `ImprovedTransactionData` implementations. `LazyTransactionData` is recommended as it is the easiest one to use.

```
GraphDatabaseService database
  = new TestGraphDatabaseFactory().newImpermanentDatabase();

database.registerTransactionEventHandler(new TransactionEventHandler<Object>()
    {
  @Override
  public Object beforeCommit(TransactionData data) throws Exception {
    ImprovedTransactionData improvedTransactionData = new LazyTransactionData(
       data);

    //have fun here with improvedTransactionData!

    return null;
  }

  @Override
   public void afterCommit(TransactionData data, Object state) {}

  @Override
   public void afterRollback(TransactionData data, Object state) {}
});
```

`FilteredTransactionData` can be used instead. They effectively hide portions of the graph, including any changes performed on nodes and relationships that are not interesting. `InclusionStrategies` are used to convey the information about what is interesting and what is not. For example, of only nodes with name equal to "Two" and no relationships at all are of interest, the example above could be modified as follows:

```
GraphDatabaseService database
  = new TestGraphDatabaseFactory().newImpermanentDatabase();

database.registerTransactionEventHandler(new TransactionEventHandler<Object>()
    {
  @Override
  public Object beforeCommit(TransactionData data) throws Exception {
```

```
    InclusionStrategies inclusionStrategies = InclusionStrategiesImpl
      .all()
      .with(new IncludeAllBusinessNodes() {
        @Override
        protected boolean doInclude(Node node) {
          return node.getProperty("name", "default").equals("Two");
        }

        @Override
        public String asString() {
          return "includeOnlyNodeWithNameEqualToTwo";
        }
      })
      .with(IncludeNoRelationships.getInstance());

    ImprovedTransactionData improvedTransactionData
      = new FilteredTransactionData(new LazyTransactionData(data),
          inclusionStrategies);

    //have fun here with improvedTransactionData!

    return null;
  }

  @Override
  public void afterCommit(TransactionData data, Object state) {}

  @Override
  public void afterRollback(TransactionData data, Object state) {}

});
```

Example Scenario Let's illustrate why this might be useful on a very simple example. Let's say we have a `FRIEND_OF` relationship in the system and it has a `strength` property indicating the strength of the friendship from 1 to 3. Let's further assume that we are interested in the total strength of all `FRIEND_OF` relationships in the entire system.

We'll achieve this by creating a custom transaction event handler that keeps track of the total strength. While not an ideal choice from a system throughput perspective, let's say for the sake of simplicity that we are going to store the total strength on the root node (with ID=0) as a `totalFriendshipStrength` property.

```
public class TotalFriendshipStrengthCounter implements TransactionEventHandler
    <Void> {
  public static final RelationshipType FRIEND_OF = DynamicRelationshipType.
      withName("FRIEND_OF");
  public static final String STRENGTH = "strength";
  public static final String TOTAL_FRIENDSHIP_STRENGTH = "
      totalFriendshipStrength";

  private final GraphDatabaseService database;

  public TotalFriendshipStrengthCounter(GraphDatabaseService database) {
    this.database = database;
  }

  @Override
  public Void beforeCommit(TransactionData data) throws Exception {
```

```
    ImprovedTransactionData improvedTransactionData = new LazyTransactionData(
        data);

    int delta = 0;

    //handle new friendships
    for (Relationship newFriendship : improvedTransactionData.
        getAllCreatedRelationships()) {
      if (newFriendship.isType(FRIEND_OF)) {
        delta += (int) newFriendship.getProperty(STRENGTH, 0);
      }
    }

    //handle changed friendships
    for (Change<Relationship> changedFriendship : improvedTransactionData.
        getAllChangedRelationships()) {
      if (changedFriendship.getPrevious().isType(FRIEND_OF)) {
        delta -= (int) changedFriendship.getPrevious().getProperty(STRENGTH,
            0);
        delta += (int) changedFriendship.getCurrent().getProperty(STRENGTH,
            0);
      }
    }

    //handle deleted friendships
    for (Relationship deletedFriendship : improvedTransactionData.
        getAllDeletedRelationships()) {
       if (deletedFriendship.isType(FRIEND_OF)) {
        delta -= (int) deletedFriendship.getProperty(STRENGTH, 0);
       }
    }

    Node root = database.getNodeById(0);
    root.setProperty(TOTAL_FRIENDSHIP_STRENGTH, (int) root.getProperty(
        TOTAL_FRIENDSHIP_STRENGTH, 0) + delta);

    return null;
  }

  @Override
  public void afterCommit(TransactionData data, Void state) {}

  @Override
  public void afterRollback(TransactionData data, Void state) {}
}
```

All that remains is registering this event handler on the database:

```
GraphDatabaseService database
  = new TestGraphDatabaseFactory().newImpermanentDatabase();

database.registerTransactionEventHandler(
  new TotalFriendshipStrengthCounter(database));
```

`TotalFriendshipStrengthCountingDemo` demonstrates the entire example.

Usage in Detail The API categorizes `PropertyContainers`, i.e., `Nodes` and `Relationships` into:

- created in this transaction

- deleted in this transaction
- changed in this transaction, i.e those with at least one property created, deleted, or changed
- untouched by this transaction

Users can find out, whether a `PropertyContainer` has been created, deleted, or changed in this transaction and obtain all the created, deleted, and changed PropertyContainers.

Properties that have been created, deleted, and changed in the transaction are categorised by the changed `PropertyContainer` they belong to. Users can find out, which properties have been created, deleted, and changed for a given changed `PropertyContainer` and check, whether a given property for a given changed `PropertyContainer` has been created, deleted, or changed.

Properties of created `PropertyContainers` are available through the actual created `PropertyContainer`. Properties of deleted `PropertyContainers` (as they were before the transaction started) are available through the snapshot of the deleted `PropertyContainer`, obtained by calling `getDeleted(Node)` or `getDeleted(Relationship)`. Properties of created and deleted containers will not be returned by `changedProperties(Node)` and `changedProperties(Relationship)` as these only return changed properties of changed `PropertyContainers`.

Changed `PropertyContainers` and properties are wrapped in a `Change` object which holds the previous state of the object before the transaction started, and the current state of the object (when the transaction commits).

All created `PropertyContainers` and properties and current versions of changed `PropertyContainers` and properties can be accessed by native Neo4j API and the traversal API as one would expect. For example, one can traverse the graph starting from a newly created node, using a mixture of newly created and already existing relationships. In other words, one can traverse the graph as if the transaction has already been committed. This is similar to using TransactionData.

A major difference between this API and `TransactionData`, however, is what one can do with the returned information about deleted `PropertyContainers` and properties and the previous versions thereof. With this API, one can traverse a *snapshot* of the graph as it was before the transaction started. As opposed to the `TransactionData` API, this will not result in exceptions being thrown.

For example, one can start traversing the graph from a deleted `Node`, or the previous version of a changed `Node`. Such a traversal will only traverse `Relationships` that existed before the transaction started and will return properties and their values as they were before the transaction started. This is achieved using `NodeSnapshot` and `RelationshipSnapshot` decorators.

One can even perform additional mutating operations on the previous version (snapshot) of the graph, provided that the mutated objects have been changed in the transaction (as opposed to deleted). Mutating deleted PropertyContainers and properties does not make any sense and will cause exceptions.

To summarise, this API gives access to two versions of the same graph. Through created PropertyContainers and/or their current versions, one can traverse the current version of the graph as it will be after the transaction commits. Through deleted and/or previous versions of PropertyContainers, one can traverse the previous snapshot of the graph, as it was before the transaction started.

Batch Usage In case you would like to use the Neo4j BatchInserter API but still get access to ImprovedTransactionData during batch insert operations, TransactionSimulatingBatchInserterImpl is the class for you. It is a BatchInserter but allows TransactionEventHandlers to be registered on it. It then simulates a transaction commit every once in a while (configurable, please refer to JavaDoc).

As a GraphDatabaseService equivalent for batch inserts, this project provides TransactionSimulatingBatchGraphDatabase for completeness, but its usage is discouraged.

DTOs

In com.graphaware.propertycontainer.dto, you will find classes used for creating detached representations (or Data Transfer Objects (DTOs)) from Nodes, Relationships and Properties.

First, there are property-less representations of Relationships, useful for encapsulating a RelationshipType and (optionally) Direction. These can be found in the common sub-package. For instance, if you need an object that encapsulates RelationshipType and Direction, which can be constructed from a Relationship object, use

```
Relationship relationship = ... //get it in the database
HasTypeAndDirection relationshipRepresentation
  = new TypeAndDirection(relationship); //now you can store this, serialise it
     , or whatever
```

Of course, other commonly needed constructors are provided. For string-convertible encapsulation of RelationshipType and Direction, use SerializableTypeAndDirectionImpl, which provides a toString(...) method and a string-based constructor.

If your relationship representations need to contain properties as well, have a look at the plain.relationship sub-package. Specifically,there are immutable relationship representations, like ImmutableRelationshipImpl and ImmutableDirectedRelationshipImpl, and their mutable counterparts. The reason they are in the plain package is that no conversion is done on property values; they are represented as Objects.

In the `string.relationship` package, you will find some classes with the same names as above (`ImmutableRelationshipImpl` and `ImmutableDirectedRelationshipImpl,...`). These are essentially the same except that property values are represented as strings. This is useful, for instance, when you want to convert a `Relationship` representation (including its properties) to and from a single string. For that purpose, use `SerializableRelationshipImpl` and `SerializableDirectedRelationshipImpl`.

Please note that when using this feature, it would be good not to name anything (properties, relationships) with names that start with "*GA*". Also, please refrain from using the "#" (or anything else you choose to be your information separator in string representations of relationships, nodes, and properties) symbol altogether, especially in relationship and node property values.

Neo4j does not allow null keys for properties, but *does allow* empty strings to be keys. GraphAware *does not allow this*, please make sure you do not use empty strings as property keys. Empty strings (or nulls) as values are absolutely fine.

Miscellaneous Utilities

The following functionality is also provided:

- Arrays (see `ArrayUtils`)
  - Determine if an object is a primitive array
  - Convert an array to a string representation
  - Check equality of two `Objects` which may or may not be arrays
  - Check equality of two `Map<String, Object>` instances, where the `Object`-typed values may or may not be arrays
- Property Containers (see `PropertyContainerUtils`)
  - Convert a `PropertyContainer` to a Map of properties
  - Delete nodes with all their relationships automatically, avoiding a `org.neo4j.kernel.impl.nioneo.store.ConstraintViolationException: Node record Node[xxx] still has relationships`, using `DeleteUtils.deleteNodeAndRelationships(node);`
- Relationship Directions
  - The need to determine the direction of a relationship is quite common. The `Relationship` object does not provide the functionality for the obvious reason that it depends on "who's point of view we're talking about". In order to resolve a direction from a specific Node's

point of view, use

`DirectionUtils.resolveDirection(Relationship relationship, Node pointOfView);`

- Iterables (see `IterableUtils` in tests)
  - Count iterables by iterating over them, unless they're a collection in which case just return `size()`
  - Randomise iterables by iterating over them and shuffling them, unless they're a collection in which case just shuffle
  - Convert iterables to lists
  - Check if iterable contains an object by iterating over the iterable, unless it's a collection in which case just return `contains(..)`

... and more, please see JavaDoc.

Appendix C

Relationship Count Module User Guide

This user guide has been created for the Relationship Count Module. It is intended as a public manual for the module users and does not assume familiarity with any other part of this report, except Appendix B. This user guide as well as all the code can be downloaded from https://github.com/graphaware.

C.1 GraphAware Relationship Count Module

In some Neo4j applications, it is useful to know how many relationships of a given type, perhaps with different properties, are present on a node. Naive on-demand relationship counting quickly becomes inefficient with large numbers of relationships per node.

The aim of this GraphAware Relationship Count Module is to provide an easy-to-use, high-performance relationship counting mechanism.

C.1.1 Download

Node: In order to use this module, you will also need the GraphAware Framework.

Releases Releases are synced to Maven Central repository. To use the latest release, download it and put it on your classpath. When using Maven, include the following snippet in your pom.xml:

```
<dependencies>
    ...
    <dependency>
```

```
        <groupId>com.graphaware</groupId>
        <artifactId>neo4j-relcount</artifactId>
        <version>1.9-1.3</version>
    </dependency>
    ...
</dependencies>
```

Snapshots To use the latest development version, just clone this repository, run `mvn clean install` and put the produced .jar file (found in target) into your classpath. If using Maven for your own development, include the following snippet in your pom.xml instead of copying the .jar:

```
<dependencies>
    ...
    <dependency>
        <groupId>com.graphaware</groupId>
        <artifactId>neo4j-relcount</artifactId>
        <version>1.9-1.4-SNAPSHOT</version>
    </dependency>
    ...
</dependencies>
```

Note on Versioning Scheme The version number has two parts, separated by a dash. The first part indicates compatibility with a Neo4j version. The second part is the version of the module. For example, version 1.9-1.2 is a 1.2 version of the module compatible with Neo4j 1.9.x.

Compatibility This module is compatible with Neo4j v. 1.9.x and GraphAware Framework v. 1.9-1.7.

C.1.2 Usage

Once set up (read below), it is very simple to use the API.

```
Node node = ... //find a node somewhere, perhaps in an index

RelationshipCounter relationshipCounter = ... //instantiate some kind of
    relationship counter

int count = relationshipCounter.count(node); //DONE!
```

A few different kinds of relationship counters are provided. There are two categories: *simple* relationship counters, found in `com.graphaware.relcount.simple`, only deal with relationship types and directions, but ignore relationship properties. *Full* relationship counters, found in `com.graphaware.relcount.full`, on the other hand, are more powerful as they take relationship properties into account as well.

C.1.3 Simple Relationship Counters

If the only thing of interest are relationship counts per type and direction (relationship properties do not matter), it is best to use simple relationship counters, from simplicity and performance point of view.

Simple Caching Relationship Counter

The most efficient simple counter is the `SimpleCachedRelationshipCounter`. As the name suggests, it counts relationships by reading them from "cache", i.e., nodes' properties. In order for this caching mechanism to work, you need to be using the GraphAware Framework with `SimpleRelationshipCountModule` registered.

The simplest default setup looks like this:

```
GraphDatabaseService database = ... //your database

GraphAwareFramework framework = new GraphAwareFramework(database);
framework.registerModule(new SimpleRelationshipCountModule());
framework.start();
```

Now, let's say you have a very simple graph with 10 people and 2 cities. Each person lives in one of the cities (relationship type LIVES_IN), and each person follows every other person on Twitter (relationship type FOLLOWS).

In order to count all followers of a person named Tracy, who is represented by node with ID = 2 in Neo4j, you would write the following:

```
Node tracy = database.getNodeById(2);

RelationshipCounter followers
  = new SimpleCachedRelationshipCounter(FOLLOWS, INCOMING);

followers.count(tracy); //returns 9
```

For graphs with thousands (or more) relationships per node, this way of counting relationships can be an significantly faster than the "naive" approach of traversing all relationships.

Simple Naive Relationship Counter

It is possible to use the `RelationshipCounter` API without any caching at all, using the "naive" approach.

The following snippet will count all Tracy's followers by traversing and inspecting all relationships:

```
Node tracy = database.getNodeById(2);

RelationshipCounter followers
  = new SimpleNaiveRelationshipCounter(FOLLOWS, INCOMING);

followers.count(tracy);
```

C.1.4 Full Relationship Counters

Full relationship counters are capable of counting relationships based on their types, directions, and properties.

Full Caching Relationship Counter

The most efficient full counter is the `FullCachedRelationshipCounter`. As the name suggests, it counts relationships by reading them from "cache", i.e., nodes' properties. In order for this caching mechanism to work, you need to be using the GraphAware Framework with `FullRelationshipCountModule` registered.

The simplest default setup looks like this:

```
GraphAwareFramework framework = new GraphAwareFramework(database);
FullRelationshipCountModule module = new FullRelationshipCountModule();
framework.registerModule(module);
framework.start();
```

Now, let's say you have a very simple graph with 10 people and 2 cities. Each person lives in one of the cities (relationship type LIVES_IN), and each person follows every other person on Twitter (relationship type FOLLOWS). Furthermore, there can be an optional "strength" property on each FOLLOWS relationship indicating the strength of a person's interest into the other person (1 or 2).

In order to count all followers of a person named Tracy, who is represented by node with ID = 2 in Neo4j, you would write the following:

```
Node tracy = database.getNodeById(2);
FullRelationshipCounter followers = module.cachedCounter(FOLLOWS, INCOMING);
followers.count(tracy); //returns 9
```

Alternatively, if you do not have access to the module object from when you've set things up, you can instantiate the counter directly:

```
Node tracy = database.getNodeById(2);

FullRelationshipCounter followers
  = new FullCachedRelationshipCounter(FOLLOWS, INCOMING);

followers.count(tracy); //returns 9
```

The first approach is preferred, however, because it simplifies things when using the module (or the Framework) with custom configuration.

If you wanted to know, how many of those followers are very interested in Tracy (strength = 2):

```
Node tracy = database.getNodeById(2);

FullRelationshipCounter followersStrength2
  = module.cachedCounter(FOLLOWS, INCOMING).with(STRENGTH, 2);

followersStrength2.count(tracy);
```

When counting using `module.cachedCounter(FOLLOWS, INCOMING)`, all incoming relationships of type FOLLOWS are taken into account, including those with and without the strength property. What if, however, the lack of the strength property has some meaning, i.e., if we want to consider "undefined" as a separate case? This kind of counting is referred to as "literal" counting and would be done like this:

```
Node tracy = database.getNodeById(2);
FullRelationshipCounter followers = module.cachedCounter(FOLLOWS, INCOMING);
followers.countLiterally(tracy);
```

For graphs with thousands (or more) relationships per node, this way of counting relationships can be an order of magnitude faster than a naive approach of traversing all relationships and inspecting their properties.

How does it work? There is no magic. The module inspects all transactions before they are committed to the database and analyses them for any created, deleted, or modified relationships.

It caches the relationship counts as properties on each node, both for incoming and outgoing relationships. In order not to pollute nodes with meaningless properties, a `RelationshipCountCompactor`, as the name suggests, compacts the cached information.

Let's illustrate that on an example. Suppose that a node has no relationships to start with. When you create the first outgoing relationship of type `FRIEND_OF` with properties `level` equal to `2` and `timestamp` equal to `1368206683579`, the following property is automatically written to the node:

```
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#1368206683579 = 1
```

Let's break it down:

- `_GA_` is a prefix for all GraphAware internal metadata.
- `FRC_` is the default ID of the FullRelationshipCountModule. This can be configured on per-module basis and is useful for registering multiple modules performing the same functionality with different configurations.
- `FRIEND_OF` is the relationship type
- `#` is a configurable information delimiter GraphAware uses internally.
- `level` is the key of the first property
- `2` is the value of the first property (level)
- `timestamp` is the key of the second property
- `1368206683579` is the value of the second property (timestamp)
- `1` is the cached number of relationships matching this representation (stored as a value of the property)

NOTE: None of the application level nodes or relationships should have names, types, labels, property keys or values containing the following strings: * `_GA_` * `#` (can be changed if needed)

That includes user input written into properties of nodes and relationship. Please check for this in your application and encode it somehow.

Right, at some point, after our node makes more friends, the situation will look something like this:

```
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#1368206683579 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#1#timestamp#1368206668364 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#1368206623759 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#1368924528927 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#0#timestamp#1368092348239 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#1368547772839 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#1#timestamp#1368542321123 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#1368254232452 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#1#timestamp#1368546532344 = 1
```

```
_GA_FRC_FRIEND_OF#OUTGOING#level#0#timestamp#1363234542345 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#0#timestamp#1363234555555 = 1
```

At that point, the compactor looks at the situation finds out there are too many cached relationship counts. More specifically, there is a threshold called the *compaction threshold* which by default is set to 20. Let's illustrate with 10.

The compactor thus tries to generalise the cached relationships. One such generalisation might involve replacing the timestamp with a wildcard (*GA**), generating representations like this:

```
_GA_FRC_FRIEND_OF#OUTGOING#level#0#timestamp#_GA_*
_GA_FRC_FRIEND_OF#OUTGOING#level#1#timestamp#_GA_*
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#_GA_*
```

Then it compacts the cached relationship counts that match these representations. In our example, it results in this:

```
 _GA_FRC_FRIEND_OF#OUTGOING#level#0#timestamp#_GA_* = 3
 _GA_FRC_FRIEND_OF#OUTGOING#level#1#timestamp#_GA_* = 3
 _GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#_GA_* = 5
```

After that, timestamp will always be ignored for these relationships, so if the next created relationships is

```
_GA_FRC_FRIEND_OF#OUTGOING#level#0#timestamp#1363266542345
```

it will result in

```
_GA_FRC_FRIEND_OF#OUTGOING#level#0#timestamp#_GA_* = 4
_GA_FRC_FRIEND_OF#OUTGOING#level#1#timestamp#_GA_* = 3
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#_GA_* = 5
```

The compaction process uses heuristics to determine, which property is the best one to generalise. In simple terms, it is the most property with most frequently changing values (measured per relationship type).

That's how it works on a high level. Of course relationships with different levels of generality are supported (for example, creating a `FRIEND_OF` relationship without a level will work just fine). When issuing a query like this

```
RelationshipCounter counter
  = new FullCachedRelationshipCounter(FRIEND_OF, OUTGOING);

int count = counter.count(node);
```

on a node with the following cache counts

```
_GA_FRC_FRIEND_OF#OUTGOING#level#3#timestamp#1368206683579 = 1
_GA_FRC_FRIEND_OF#OUTGOING#level#2#timestamp#_GA_* = 10
_GA_FRC_FRIEND_OF#OUTGOING#level#1#timestamp#_GA_* = 20
_GA_FRC_FRIEND_OF#OUTGOING = 5 (no level or timestamp provided on these relationships)
```

the result will be... you guessed it... 36.

On the other hand, counting pure outgoing FRIEND_OF relationships with no properties would be done like this:

```
FullRelationshipCounter counter
  = new FullCachedRelationshipCounter(FRIEND_OF, OUTGOING);

int count = counter.countLiterally(node);
```

and result in 5.

However, if you now issue the following query:

```
RelationshipCounter counter = module.cachedCounter(FRIEND_OF, OUTGOING)
  .with("level", 2)
  .with("timestamp", 123456789);

int count = counter.count(node);
```

an `UnableToCountException` will be thrown, because the granularity needed for answering such a query has been compacted away. There are three ways to deal with this problem, either

- configure the compaction threshold so that this does not happen, or
- manually fallback to naive counting, or
- use `FullNaiveRelationshipCounter`, which falls back to naive counting approach automatically

Advanced Usage

There are a number of things that can be tweaked here. Let's talk about the compaction threshold first.

Compaction Threshold Level What should the compaction threshold be set to? That depends entirely on the use case. Let's use the people/places example from earlier with FOLLOWS and LIVES_IN relationships. Each node will have a number of LIVES_IN relationships, but only incoming (places) or outgoing (people). These relationships have no properties, so that's 1 property for each node.

Furthermore, each person will have incoming and outgoing FOLLOWS relationships with 3 possible "strengths": none, 1, and 2. That's 6 more properties. A compaction threshold of 7 would, therefore be appropriate for this use case.

If you know, however, that you are not going to be interested in the strength of the FOLLOWS relationships, you could well set the threshold to 3. One for the LIVES_IN relationships, and 2 for incoming and outgoing FOLLOWS relationships.

The threshold can be set when constructing the module by passing in a custom configuration:

```
GraphAwareFramework framework = new GraphAwareFramework(database);

//compaction threshold to 7
RelationshipCountStrategies relationshipCountStrategies =
  RelationshipCountStrategiesImpl.defaultStrategies().with(7);

FullRelationshipCountModule module
  = new FullRelationshipCountModule(relationshipCountStrategies);

framework.registerModule(module);
framework.start();
```

Relationship Weights Let's say you would like each relationship to have a different "weight", i.e., some relationships should count for more than one. This is entirely possible by implementing a custom `RelationshipWeighingStrategy`.

Building on the previous example, let's say you would like the FOLLOWS relationship with strength = 2 to count for 2 relationships. The following code would achieve just that:

```
GraphAwareFramework framework = new GraphAwareFramework(database);

RelationshipWeighingStrategy customWeighingStrategy = new
    RelationshipWeighingStrategy() {
  @Override
  public int getRelationshipWeight(Relationship relationship, Node pointOfView
      ) {
    return (int) relationship.getProperty(STRENGTH, 1);
  }
};

RelationshipCountStrategies relationshipCountStrategies
  = RelationshipCountStrategiesImpl.defaultStrategies()
    .with(7) //threshold
    .with(customWeighingStrategy);

FullRelationshipCountModule module
  = new FullRelationshipCountModule(relationshipCountStrategies);
```

```
framework.registerModule(module);
framework.start();
```

Excluding Relationships To exclude certain relationships from the count caching process altogether, create a strategy that implements the `RelationshipInclusionStrategy`. For example, if you're only interested in FOLLOWS relationship counts and nothing else, you could configure the module as follows:

```
GraphAwareFramework framework = new GraphAwareFramework(database);

RelationshipInclusionStrategy customRelationshipInclusionStrategy
  = new RelationshipInclusionStrategy() {
    @Override
    public boolean include(Relationship relationship) {
      return relationship.isType(FOLLOWS);
    }
};

RelationshipCountStrategies relationshipCountStrategies
  = RelationshipCountStrategiesImpl.defaultStrategies()
    .with(customRelationshipInclusionStrategy);

FullRelationshipCountModule module = new FullRelationshipCountModule(
   relationshipCountStrategies);

framework.registerModule(module);
framework.start();
```

Excluding Relationship Properties Whilst the compaction mechanism eventually excludes frequently changing properties anyway, it might be useful (at least for performance reasons) to exclude them explicitly, if you know up front that these properties are not going to be used in the counting process.

Let's say, for example, that each FOLLOWS relationship has a "timestamp" property that is pretty much unique for each relationship. In that case, you might choose to ignore that property for the purposes of relationship count caching by setting up the module in the following fashion:

```
GraphAwareFramework framework = new GraphAwareFramework(database);

RelationshipPropertyInclusionStrategy
    customRelationshipPropertyInclusionStrategy
  = new RelationshipPropertyInclusionStrategy() {
    @Override
    public boolean include(String key, Relationship propertyContainer) {
      return !"timestamp".equals(key);
    }
};

RelationshipCountStrategies relationshipCountStrategies
  = RelationshipCountStrategiesImpl.defaultStrategies()
    .with(customRelationshipPropertyInclusionStrategy);

FullRelationshipCountModule module
  = new FullRelationshipCountModule(relationshipCountStrategies);
```

```
framework.registerModule(module);
framework.start();
```

Deriving Relationship Properties Sometimes, it might be useful to derive relationship properties that are not explicitly there for the purposes of relationship counting. Let's say, for example, that you want to count the number of followers based on the followers' gender. In that case, each FOLLOWS relationship could get a derived "followeeGender" property, value of which is the gender of the followed person. Such a requirement would be achieved with the following setup:

```
GraphAwareFramework framework = new GraphAwareFramework(database);

RelationshipPropertiesExtractionStrategy customPropertiesExtractionStrategy
  = new RelationshipPropertiesExtractionStrategy() {
       @Override
       public Map<String, String> extractProperties(Relationship relationship,
            Node pointOfView) {
           //all real properties
           Map<String, String> result = PropertyContainerUtils.
              propertiesToStringMap(relationship);

           //derived property from the "other" node participating in the
              relationship
           if (relationship.isType(FOLLOWS)) {
               result.put(GENDER, relationship.getOtherNode(pointOfView).
                  getProperty(GENDER).toString());
           }

           return result;
       }
   };

RelationshipCountStrategies relationshipCountStrategies
  = RelationshipCountStrategiesImpl.defaultStrategies()
           .with(IncludeAllNodeProperties.getInstance()) //no node properties
              included by default!
           .with(customPropertiesExtractionStrategy);

FullRelationshipCountModule module = new FullRelationshipCountModule(
   relationshipCountStrategies);

framework.registerModule(module);
framework.start();
```

Counting would be done as usual:

```
Node tracy = database.getNodeById(2);

RelationshipCounter maleFollowers
  = module.cachedCounter(FOLLOWS, INCOMING).with(GENDER, MALE);

maleFollowers.count(tracy); //returns only male followers count

RelationshipCounter femaleFollowers
  = module.cachedCounter(FOLLOWS, INCOMING).with(GENDER, FEMALE);

femaleFollowers.count(tracy); //returns only female followers count
```

Full Naive Relationship Counter

It is possible to use the `RelationshipCounter` API without any caching at all. You might want to fall back to the naive approach of traversing through all relationships because you caught an `UnableToCountException`, or maybe you simply do not have enough relationships in your system to justify the write-overhead of the caching approach.

It is still advisable to obtain your `RelationshipCounter` from a module, although the module might not need to be registered with a running instance of the GraphAware framework. Even when using the naive approach, it is possible to use custom strategies (`RelationshipWeighingStrategy`, `RelationshipPropertiesExtractionStrategy`, etc.) explained above.

The following snippet will count all Tracy's followers by traversing and inspecting all relationships:

```
FullRelationshipCountModule module = new FullRelationshipCountModule();

Node tracy = database.getNodeById(2);

RelationshipCounter followers = module.naiveCounter(FOLLOWS, INCOMING);
followers.count(tracy);
```

If you're using this just for naive counting (no fallback, no custom config), it is possible to achieve the same thing using the following code, although the former approach is preferable.

```
Node tracy = database.getNodeById(2);
RelationshipCounter following
  = new FullNaiveRelationshipCounter(FOLLOWS, OUTGOING);
following.count(tracy);
```

Full Falling Back Relationship Counter

Although it is recommended to avoid getting `UnableToCountExceptions` by configuring things properly, there is an option of an automatic fallback to the naive approach when the caching approach has failed, because the needed granularity for counting some kind of relationship has been compacted away.

The following code snippet illustrates the usage:

```
GraphAwareFramework framework = new GraphAwareFramework(database);

RelationshipCountStrategies relationshipCountStrategies
  = RelationshipCountStrategiesImpl.defaultStrategies().with(3);

FullRelationshipCountModule module
  = new FullRelationshipCountModule(relationshipCountStrategies);

framework.registerModule(module);
framework.start();

populateDatabase();
```

```
Node tracy = database.getNodeById(2);

RelationshipCounter followers = module.fallingBackCounter(FOLLOWS, INCOMING);
assertEquals(9, followers.count(tracy));           //uses cache

RelationshipCounter followersStrength2
  = module.fallingBackCounter(FOLLOWS, INCOMING).with(STRENGTH, 2);

assertEquals(3, followersStrength2.count(tracy));  //falls back to naive
```