# Comparative Analysis of PropertyFirst vs. EntityFirst Modeling Approaches in Graph Databases

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Milan Bogunovic

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan, S7N 5C9

Canada

## ABSTRACT

While relational databases still hold the primary position in the database technology domain, and have been for the longest time of any Computer Science technology has since its inception, for the first time the relational databases now have valid and worthy opponent in the NoSQL database movement.

NoSQL databases, even though not many people have heard of them, with a significant number of Computer Science people included, have spread rapidly in many shapes and forms and have done so in quite a chaotic fashion. Similarly to the way they appeared and spread, design and modeling for them have been undertaken in an unstructured manner. Currently they are subcategorized in 4 main groups as: Key-value stores, Column Family stores, Document stores and Graph databases.

In this thesis, different modeling approaches for graph databases, applied to the same domain are analyzed and compared, especially from a design perspective.

The database selected here as the implemented technology is Neo4J by Neo Technologies and is a directed property graph database, which means that relationships between its data entities must have a starting and ending (or source and destination) node.

This research provides an overview of two competing modeling approaches and evaluates them in a context of a real world example.

The work done here shows that both of these modeling approaches are valid and that it is possible to fully develop a data model based on the same domain data with both approaches and that both can be used later to support application access in a similar fashion. One of the models provides for faster access to data, but at a cost of higher maintenance and increased complexity.

ACKNOWLEDGMENTS

I would like to take this opportunity and express my sincerest gratitude to my supervisor, Dr. Ralph Deters who introduced me to the whole new and brave world of NoSQL databases and without whose continuous encouragement and guidance throughout this research my thesis could not be completed. Members of the Committee Dr. Gord McCalla, Dr. Chris Zhang and Dr. Julita Vassileva deserve special thanks for all their help and suggestions made along the way. I would also like to thank my friends and a colleagues, Vivian Mahoney and Richard Gibbins, for proof reading this text, which could not have been an easy task. And last but not the least, my wife Jelena, because without her support I could not even get started.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Figure page

# LIST OF ABBREVIATIONS

| | |
|---|---|
| 1NF | First Normal Form |
| 2NF | Second Normal Form |
| 3NF | Third Normal Form |
| ACID | Atomicity, Consistency, Isolation, Durability |
| AGM | Annual General Meeting |
| BASE | Basically Available, Soft state, Eventually consistent |
| BUT | Building-Unit-Tenant relationship |
| DB | Database |
| DBMS | Database Management System |
| DGV | Database Graph Views |
| EFGDBM | Entity First Graph Database Model |
| EFM | Entity First Model |
| FK | Foreign Key |
| G-Base | Graph base |
| GDB | Graph Database |
| GDM | Graph Data Model |
| GGL | Graph Database System for Genomics |
| G-Log | Graph Log |
| GMOD | Graph-oriented Object Manipulation |
| GOAL | Graph-based Object and Association Language |
| GOOD | Graph Object Oriented Data |
| GOQL | Graph Object Oriented Query Language |

| | |
|---|---|
| GRAM | Graph Data Model and Query Language |
| GRAS | Graph Oriented Software Engineering |
| GROOVY | Graphically Represented Object-Oriented Data Model with Values |
| GUI | Graphical User Interface |
| LDM | Logical Data Model |
| NoSQL | Not only SQL |
| OLAP | Online Analytical Processing |
| OLTP | Online Transaction Processing |
| PaMaL | Pattern Matching Language |
| PFGDBM | Property First Graph Database Model |
| PFM | Property First Model |
| PK | Primary Key |
| RDBMS | Relational Database Management System |
| RDF | Resource Description Framework |
| SPARQL | Simple Protocol and RDF Query Language |
| SQL | Structured Query Language |
| XML | Extensible Markup Language |

# CHAPTER 1
# INTRODUCTION

At this point in time the mainstream database world still belongs to the Relational Database Management Systems (RDBMS) model and several big vendors like Oracle (Oracle database and MySQL), Microsoft (SQL Server), IBM (DB2) along with some smaller ones like PostgreSQL and Sybase. That has been the case for more than 20 years since the establishment of several major players in the database market (RDB, Oracle, IBM, Informix) who followed the proposition and conceptualization of the relational model created by E.F. Codd [3]. Relational databases handle structured data very well, but in recent years the need for a different kind of database technology presented itself, with the appearance of Big Data, Social Media, Cloud Computing and increased usage of the internet, which produce huge amounts of unstructured and semi structured data.

That is where the NoSQL database movement entered the picture and quickly gained popularity, though still not close to jeopardizing the relational databases' dominant position.

NoSQL databases are now is use in some of the biggest companies, some of which are shaping the field of applied Computer Science and even world economy order. In many cases they are developed by those companies for a special need as a full or partial solution. NoSQL databases, whose current count sits at around 150 [7], have been popping up in quite a chaotic way, without any standardization or rules, whenever need or want presented itself.

Similarly to the chaotic fashion in which NoSQL databases have been popping up here and there, design and modeling for them have been undertaken in an unstructured manner and in most cases based on "common sense" or "best fit for current situation" approach. This is understandable if their age and the speed at which technology changes currently are taken into

consideration. However, some major performance improvements by using NoSQL databases in certain domains over using traditional relational database have been achieved and that trend is expanding.

While relational databases are fully ACID [11, 12, 22] compliant, which means that they process transactions in a reliable manner, most NoSQL databases are BASE [13, 56] compliant, which means that they are bit less reliable in dealing with transactions, but allow for higher scalability, performance and distribution. However there are some, like graph databases, which are fully ACID compliant. NoSQL databases are easily horizontally scalable which presents a major advantage over relational databases in "big data" market.

## 1.1 NoSQL Databases

As mentioned, the NoSQL database world is made up of four major groups, key-value stores, column family stores, document stores and graph databases. While currently the best categorization available, this is still rough as lines are in many cases blurry and sometimes products could go in more than one category. Sometimes they are just put in one although they don't really belong in any of the current categories. Each of these groups requires further explanation.

Figure 1-1 graphically represents the categories into which NoSQL databases are currently divided into four quadrants.

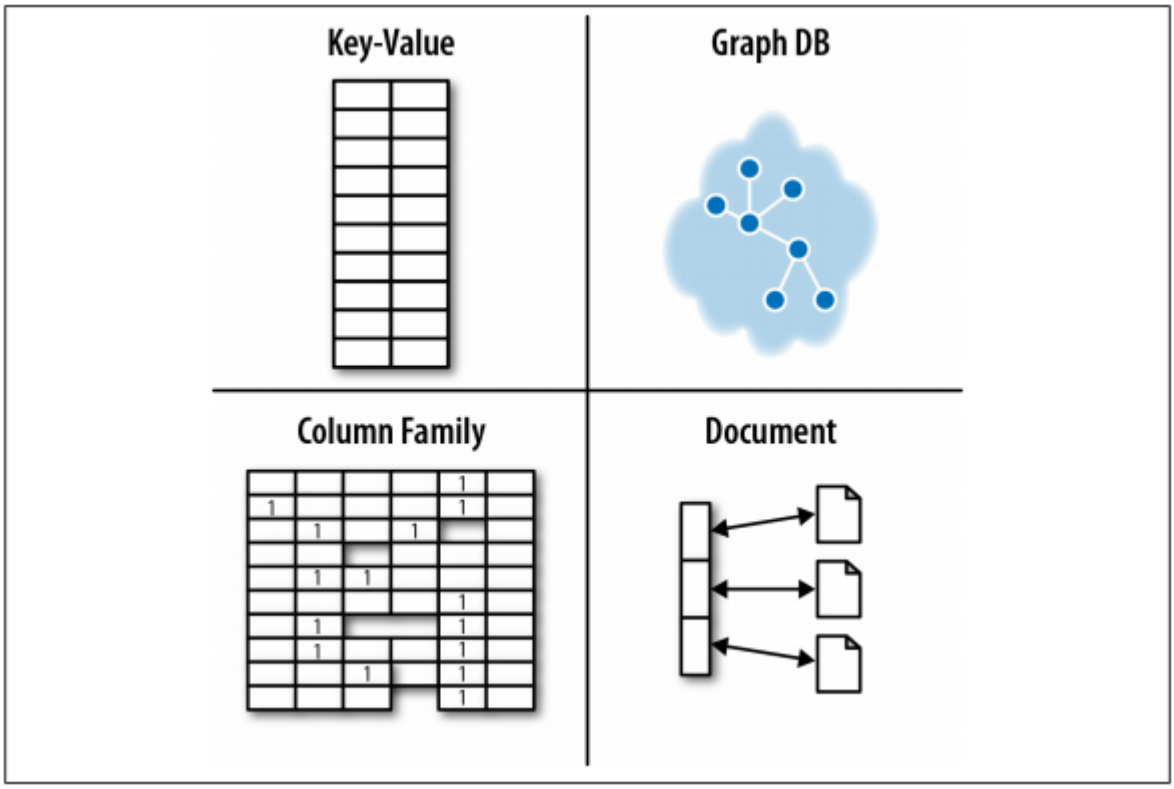


Figure 1-1. NoSQL database quadrants [2]

### 1.1.1 Key-Value Stores

These databases allow for data storage in a schemaless way and in what most looks like a simple table with two fields, a key and its value.

Key-Value stores act like large distributed hashmap data structures that store and retrieve opaque values by key [2]. They are simple structures that are easy to use, providing easy high availability, scalability and performance through load balancing. Its implementation technology helps to avoid hotspots. As with most NoSQL databases, consistency is outsourced to the application level. The Key-Value stores were preceded by Amazon's Dynamo database whose team described it by saying that it should work even "if disks are failing, network routes are flapping, or data centers are being destroyed by tornados".

### 1.1.2 Column family stores

The easiest way to understand these databases is to think of the entire database as one huge table, in which each row can “choose” which columns to have and which not to have. Even for the columns that it chooses to have, it can choose for which to have values and for which to not have values and which values can and can’t be used. Consistency and logic are outsourced to the application level. Column family stores are easy to scale and offer very good performance for large data sets.

### 1.1.3 Document stores

The easiest way to understand Document stores, is to think of documents - which is how data is stored here - as rows in a relational database, without any restrictions and schema limitations, which can contain any data and data types. This makes them almost independent from other data, i.e. other documents. Consistency and logic are also outsourced to the application level. Document stores are easy to scale and offer very good performance for large data sets. The term document comes from the fact that these databases store and retrieve “documents” by id. Document in this context can be almost any combination of data, which gives the database its schemaless nature.

### 1.1.4 Graph databases

Graph databases, although part of the NoSQL movement and one of its categories, are quite different than any other NoSQL database. In addition to being ACID compliant, they store data in entities (represented as nodes and relationships between nodes) and properties (that can be set for both nodes and relationships). They have picked up some of the best features from both the relational and NoSQL worlds to present a full solution for most real world situations. There is a

saying in the graph database world: “anything that can be drawn on a whiteboard can be stored in a graph database in a natural way”.

Graph databases are designed for highly inter-connected data, with a number of relationships and data that can be represented as a graph (data entities connected with relationships between them). Graph databases can be further subcategorized into three groups [2]: Property graphs, Hypergraphs and Triple stores.

Graph databases are the focus of this thesis and are explained in more detail in following sections.

## 1.2 Introduction to Graph Databases

Graph databases are based on graphs and graph theory [14, 15], but this thesis will not go into details about graph theory, but rather keep the focus on the graph database.

### 1.2.1 What are graphs?

Formally, a graph is just a collection of vertices and edges or, in less intimidating language, a set of nodes and the relationships that connect them. [2]

In reality, graphs are all around us. Every two objects that have a relationship or connection between them represent a graph. For instance, the relationship between a person and that person’s shoes presents a graph as seen in Figure 1-2.

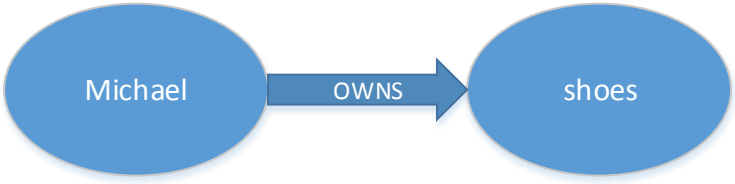


Figure 1-2. Simple graph with 2 objects

In this example the connection OWNS is used, but there could be multiple connections (relationships) between two objects (nodes) like, wears, cleans, ties, etc.

Similarly multiple connection graphs could have more than one object, as shown in Figure 1-3 which represents a graph with 3 objects.

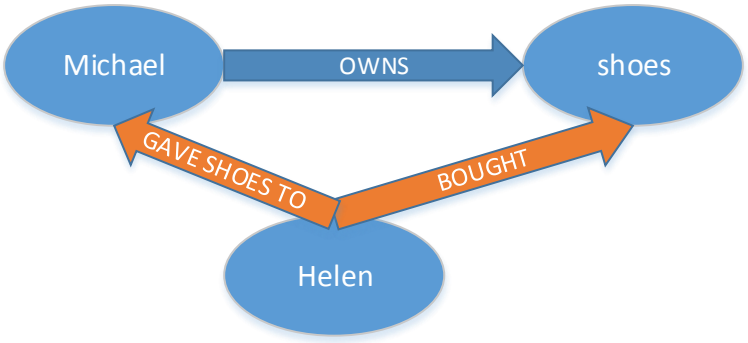


Figure 1-3. Simple graph with 3 objects

Figure 1-3 represents as a graph a situation in which Helen bought shoes, gave them to Michael as a gift and now Michael owns them. This graphically represents the objects Michael, Helen and shoes and the current relationships that exist between them.

Possibilities with graphs are endless, such as the number of objects and the connections between them. Practically every situation can be presented as a graph. It is only a question of what is appropriate for your needs and if it is feasible.

### 1.2.2 What are graph databases?

A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way. [8]

Graph Databases are part of the much larger movement of NoSQL databases which are mainly driven by big data needs in the contemporary environment supported by relatively cheap storage.

Graph databases consist of two major elements: objects, which are also called nodes or vertices, and the relationships between them, which are also called edges. Collectively these two major elements can be referred to as entities. In addition to that, data in graph databases can be stored as properties, which can exist for both nodes and relationships.

Graph databases can further be divided into three groups:

- Property Graphs [2]
- Hypergraphs [2]
- Resource Description Framework (RDF) triples [2]

More in depth information about the different graph database models can be found in Survey of Graph Database Models [1]. The focus of this thesis is Property graph databases, more precisely directed property graph databases and in particular Neo4J [5] by Neo Technology [6] as one of most prominent databases in that domain.

### 1.2.3 Neo4J

Neo4j is an open-source graph database which stores data in nodes connected by directed, typed relationships with properties on both, also known as a property graph. [9]

While graph databases share a lot of positive features (scalability, schemaless nature, speed) with the rest of the NoSQL databases, they also have one of the crucial relational database features that most other NoSQL databases don’t: graph databases are fully ACID compliant. Most other NoSQL databases are BASE compliant. The Neo4J database also (in its own way) shares some of the best features that the rest of the NoSQL quadrants have, like functioning as

key-values pairs, document stores or column families while keeping full ACID compliancy. One of the advantages other NoSQL quadrants have over Neo4J is easier and more robust clustering for write operations, while keeping at par with clustering for read operations.

## 1.3 Introduction to the Database and Data Modeling for Graph Databases

Modeling in graph databases can be simply described in one sentence: whatever is drawn on a whiteboard, any objects and connections that appear there will also appear in a graph database model. Only the terminology changes a bit as objects and connections become nodes (vertices) and relationships (edges) respectively.

### 1.3.1 Simple 2-node model

So, the real life situation (Michael owns shoes) that was drawn in figure 1-2 on a whiteboard, can be presented as graph database model in Figure 1-4.

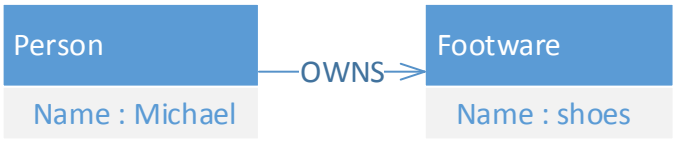


Figure 1-4. Graph database model of a simple graph with 2 nodes

In this model nodes are represented by squares, divided in two parts. The first part (upper in blue) represents object's type (could be person, animal, building, vehicle, etc.) in this case person and his footwear and second part (lower in grey) represents the object's properties (in the case of a person, it could be name, address, date of birth, height, color of eyes, etc.). In this example it is

the person's first name and shoes. The relationship in this model is represented with a line that connects two nodes with an arrow pointed in the direction of the relationship (directed line), that contains the relationship name. This is the simplest possible model. It is possible to have standalone nodes, but not standalone relationships or relationships without a starting and ending nodes.

### 1.3.2 Simple 3-node model

Another real life situation (Helen bought the shoes and gave them as gift to Michael, who now owns them) presented in Figure 1-3 is presented as graph database model in Figure 1-5.

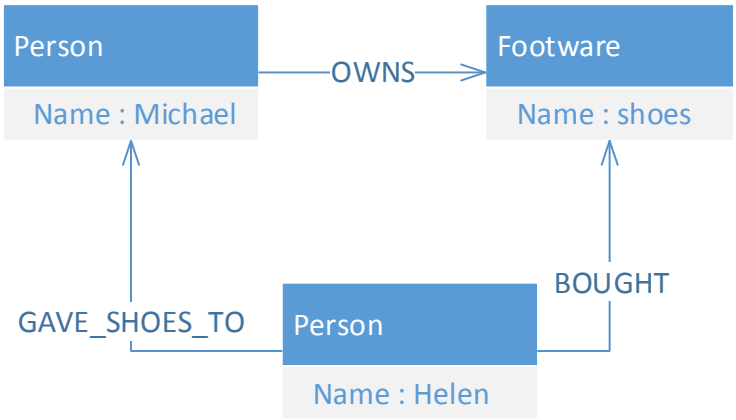


Figure 1-5. Graph database model with 3 nodes and 3 relationships

### 1.3.3 Extended 3-node model

Relationships between people (and objects) are never that simple. Nodes could have multiple relationships that connect them. For instance, the relationship between the person Helen and the person Michael, in addition to Helen giving shoes to Michael, could be that Helen is married to Michael. A relationships between the person Michael and the footwear shoes, in addition to Michael owning the shoes, could be that Michael wears the shoes; this could very well be only a temporary relationship.

The model represented in Figure 1-5 shows only one object with two people. In the case of multiple objects, multiple relationships between people and objects are possible and likely as well as multiple relationships between objects.

Figure 1-6 shows additional relationships.

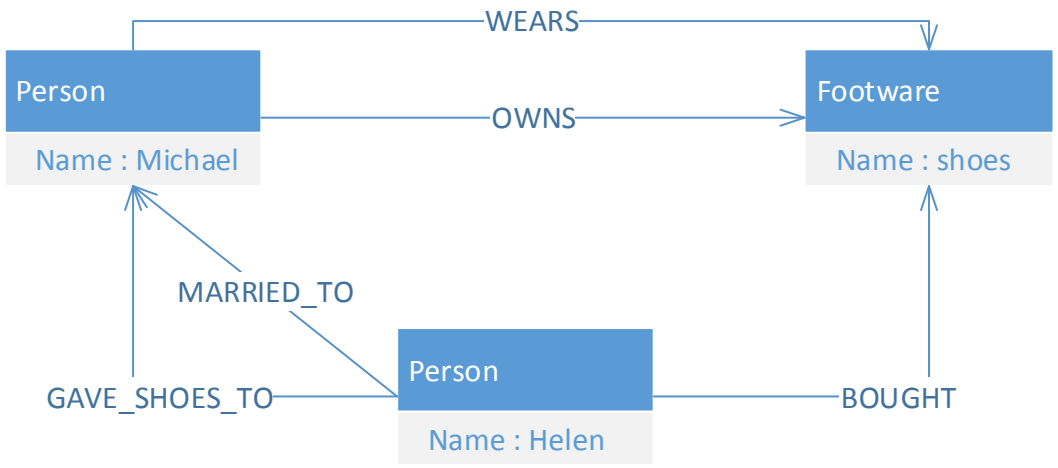


Figure 1-6. Graph database model with 3 nodes and 5 relationships

Some of these relationships could be related and combined and used as one, while others remain independent. An example of this is two relationships, OWNS and WEARS, between the person Michael and the footwear shoes could be combined into OWNS_WEARS. The relationships MARRIED_TO and GAVE_SHOES_TO, being pretty much independent, could not be combined. The above is represented in Figure 1-7.

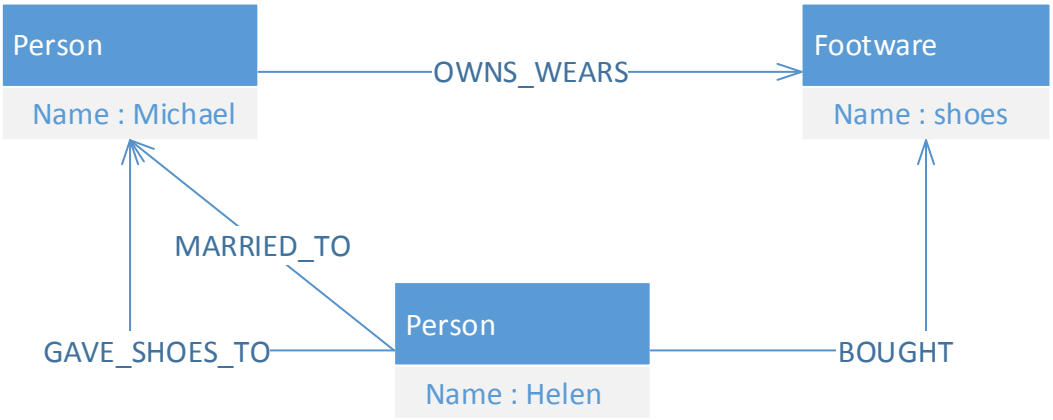


Figure 1-7. Graph database model with combined relationships

As well, the direction of multiple relationships between two nodes could be the same or different, depending on relationship type. Some could even be redirected, while some have to stay the same. For instance, the relationship MARRIED_TO is bidirectional and can either be directed from Helen towards Michael or vice versa. On the other hand, the relationship MARRIED_TO could be broken into two relationships (if more detailed information is required by the application domain or even just to make querying simpler or more efficient) called HUSBAND_OF and WIFE_OF. This is represented in Figure 1-8, in which other elements (footwear and relationship GAVE_SHOES_TO) have been removed for clarity.

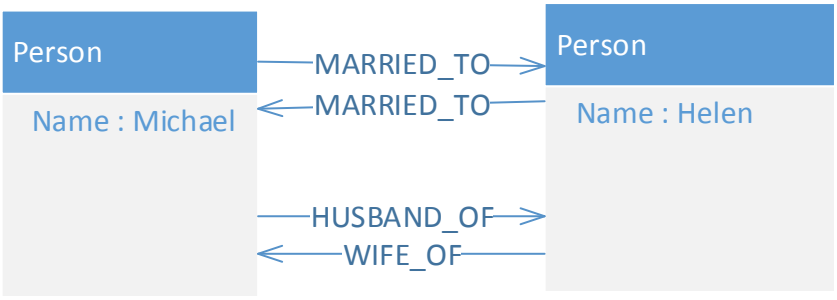


Figure 1-8. Graph db model with reversed direction relationships and broken into two

## 1.4 Additional attributes

All of these examples are very simple and present very basic information that a person or an object like shoes could have. For instance, the person will have a first and last name, age (or date of birth), foot size, favorite brand, any special requirements for shoes (like insoles) and shoe type preference (like sports, dress, casual, open, etc.). These are attributes related only to person identification and shoes preferences. Shoes, on the other hand, could also have additional information, like model name, size, color, type, brand, price, etc.

In addition to further object information (object in this case includes people) there could be more information about relationships between objects (between people and between people and things). The relationship between Michael and Helen could be that they are married, but they could also be just friends or colleagues. Their relationship could further be quantified by the number of years they have known each other or by activities that they like to do together. Michael could dislike certain types of shoes, for instance dress shoes, that Helen likes very much (quantification of the relationship).

### 1.4.1 Properties

All this additional information and so much more can easily be presented in a graph database. In fact, it can be presented in two very specific, different ways. The information can be presented and stored as properties (Neo4J database being a directed property graph database) for both nodes and relationships or as separate nodes and relationships with relationships to the original nodes.

Figure 1-9 represents additional information modeled as properties, as a simple graph with 2 nodes and one relationship.

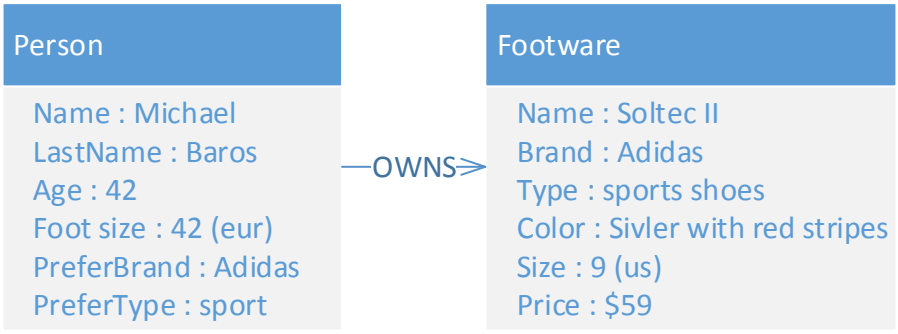


Figure 1-9. Graph database model with additional attributes stored as properties

### 1.4.2 Entities

As opposed to putting additional attributes into properties, they can be included in a database model with additional nodes and relationships (which could be together called entities) and Figure 1-10 shows exactly that, for a simple graph with 2 original nodes.

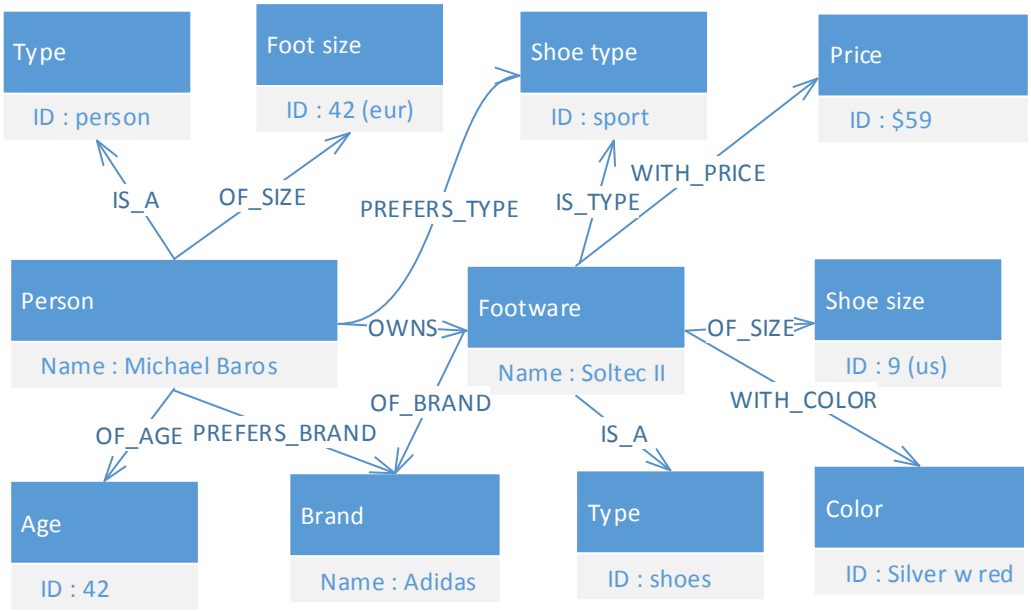


Figure 1-10. Graph database model with additional attributes stored as entities

### 1.4.3 Properties vs. entities

It can be immediately noticed that the number of nodes and relationships (considered first-class "citizens" in graph databases) grew significantly in this modeling approach, while each node, including the original two, kept only one property.

Some of the differences per entity type are:

- The number of the nodes increased from the original two to eleven and only because some of nodes were reused (like node Adidas for brand, which in this case match since both brand of Soltec II shoes and Michael's preferred brand is Adidas, but that does not always have to be the case and usually it isn't. Another example of a matching and reusable node is Shoe type with value sports. An example of nodes not matching is the node Size, in this case because European size number 42 and US size number 9 do not match exactly. It is also a good reason not to match them because of the different measuring types although they could also be mapped based on the closest number).
- The number of relationships increased from the original one to twelve (ten unique relationships as IS_A and OF_SIZE are used in two places). It is interesting to note that even for matching nodes (like Brand and Shoe type ) the relationships are different (OF_BRAND vs. PREFERS_BRAND and IS_TYPE vs. PREFERS_TYPE) in order to reflect the difference between actuality and preference.

Modeling for the three original nodes would be done in similar way in models oriented to storing additional information as properties or as entities. In the case of storing information as properties, a few more properties could be added to the node Helen (which could be exactly the same as or completely different from those stored for the node Michael, or a combination of the two, with some properties the same, and some different). In the case of storing information as

entities, several new nodes would be created (some could be used more than once, depending on what information it is) which could be connected by new or existing relationships (again, depending on the nature of the information).

# CHAPTER 2
# PROBLEM DEFINITION

As seen in the previous chapter, graph databases can model data in different ways. Which way is “better”? In real life relationships are never as simple as the person Michael owning shoes that were given to him by Helen, that being the only relevant information. Data captured and stored would be so much more extensive, even some that initially could be considered irrelevant. In relational databases there is not much unknown, data is stored in tables and the only challenge is to decide what to collect and how to link relevant information. In graph databases this data could be stored in two opposite ways, which are property and entity centric (or some combination of them that could be considered a third way).

## 2.1 Modeling Approaches

The first way is called the PropertyFirst graph database model (PFGDBM or just PF model) where additional data is stored as properties in major nodes. This model looks very close to a relational model, where we can match one person (or any other object) and its attributes to one row in a relational table. This model is just way more flexible as it is not necessary to predefine which kind of information you want to store.

The second way where the data is stored as entities (nodes and relationships) and that one is called the EntityFirst graph database model (EFGDBM or just EF model).

In addition to these two data models, data could be stored as a combination of these two, storing some of the data as properties and some of the data as entities, based on “where it makes sense”. That is probably how most modeling in graph databases is done now.

## 2.2 Benefits

Every document, paper or book about graph databases or Neo4J, as one of the more popular and most documented in the field, states that they are excellent when dealing with multiple depths of relationships (friend of a friend of a friend of friend of a friend and so on) where relational databases perform miserably (or fail to complete queries) due to its use of expensive joins. But if that was be the only area where graph databases could be applied they wouldn't be very much used. Interest in them (followed by investment in development and maintenance) would start to fall and it would inevitably mean their demise. There must be other areas where graph databases are superior or at least competitive with other forms of databases. One of the most important features which makes this particular database suitable for a particular domain or application is its ability to adapt and provide models that match domain needs as closely as possible.

But what has been done so far to explain and digest modeling approaches with graph databases? Going a little bit deeper into modeling and the different approaches could prove valuable and provide some guidance as to when to use which technique.

Although using the method of "common sense" to determine where to put certain data would work decently in many cases, there could be situations where using the PropertyFirst or the EntityFirst model (or not even pure breeds but the "as much as possible" approach) would make more sense and could provide gains in performance, disk and resource usage, scalability, ability to expand applications and plug in different options (regions, sections, use cases) more easily.

## 2.3 Research Goals

### 2.3.1 First goal

Implement the chosen domain with both modeling approaches into separate database instances. Make sure that both database instances can support identical operations like creation and manipulation of data. At same time make sure that both models can support and store identical data, although modeled differently.

### 2.3.2 Second goal

Compare creation statement and query performance against both databases instances and measure differences. Analyze any discovered differences and discuss possible causes.

### 2.3.3 Third goal

Expand the model with additional data sets and reapply the second and third step to both instances. Discuss possible differences in performance between instances with one and multiple data sets for both models.

### 2.3.5 Fourth goal

Compare overall differences between the two implementations, including size on disk, possible creation of hot nodes (nodes with the most amount of incoming or outgoing relationships), data duplication and the query complexity due to the modeling approach used.

# CHAPTER 3
# LITRATURE REVIEW

While graph databases are part of the NoSQL movement and its "full member", they are actually the black sheep in the herd, with a very uncommon model in the NoSQL world, if we can talk about "common" in NoSQL at all. Graph databases even have some similarities with relational databases, like full ACID [11, 12, 22] compliancy, while almost all other NoSQL are BASE [13, 56] compliant. Graph databases still share some similarities with the other three major NoSQL quadrants, for instance, their schemaless nature. The following Table 3-1 shows the similarities and differences between graph databases, relational databases and rest of NoSQL databases. Note that each aspect applies to the majority of databases in a group but may not apply to every single one.

| | Relational databases | Graph databases | Other NoSQL |
|---|---|---|---|
| Transaction | ACID | ACID | BASE |
| Query language | SQL | SQL like Cypher | No SQL like |
| Consistency enforced | DB level | DB level | App level |
| Scalability | Expensive | Cheap | Cheap |
| Data | Structured | Non-structured | Non-structured |
| Schema | Hard schema | Schemaless | Schemaless |

Table 3-1. Commonalities and differences between database models

Only some of the most well-known parameters are listed here. This is just meant as an example, not a complete reference of the differences between these databases. All these parameters suggest differences between database models and the data modeling processes.

ACID compliance means that data consistency is guaranteed by the database management system. But that is only one of the ACID properties. ACID refers to properties of transaction management in database systems. All Relational DBMS are ACID compliant, while of NoSQL database systems only graph databases are ACID compliant. ACID stands for:

- Atomicity - transactions are atomic (all or nothing [22]) which means that all of transaction is successful or all of it is unsuccessful. All of it either commits or rolls back.
- Consistency – refers to mentioned data consistency; once it completes, a transaction takes the database from one consistent state to another
- Isolation – in the case of two concurrent transactions, refers to the inability of one transaction to see what the other transaction is doing, and vice versa, until the work has been completed successfully and committed. In modern relational databases it is possible to set the isolation level and change its default state of not being able to see anything that is not committed.
- Durability – refers to the database's ability to guarantee that once a transaction commits, it will be preserved even in case of system crashes, which forces uncommitted transactions to be rolled back.

While ACID properties for relational databases sound great and very reliable, they tend to affect database availability and ability to scale, especially in distributed environments. Since availability and scalability are couple of the biggest reasons for the inception of NoSQL databases, consistency had to be sacrificed and BASE was developed. BASE stands for:

- Basically Available
- Soft state
- Eventually consistent

Basically available, soft state, eventually consistent model allows for partial failures, for a subsets of users, data inconsistencies and delayed consistency under the assumption that if there are no new updates, eventually the data will be synchronized and consistent. In reality the data never is consistent as there are always new changes that keep parts of the data inconsistent. These features, on the other hand, have achieved availability, scalability and performance as never seen before.

This thesis focuses on graph database models and data modeling. Available and published scholarly work on this subject is extremely limited or not available at all. All the modeling work for graph databases currently available (since graph databases are relatively new), focuses on "how would you model this domain" and the particulars of specific modeling goals. There is no attempt to define the modeling approach or dwelling on the possibility that modeling the same data could be done in some way other than the "default". All other mention of the word model, with regards to graph databases, is done in relation to databases models (even when they call it data models), not data models or approaches to modeling data.

## 3.1 Relational Databases

The inception of Relational databases started with the 1970 paper "A Relational Model of Data for Large Shared Data Banks [3]" by E.F. Codd. Since then they have gained almost the entire database market and dominated all aspects of the database development domain to the present day. Multiple commercial vendors exist like:

- Oracle RDBMS [16]
- Microsoft SQL Server [17]
- IBM DB2 [18]
- Oracle's MySQL [19]
- PostgreSQL [20]
- Sybase [21]

The list goes on, but these are the current main names. Almost all of them claim that they are "market leaders" in some aspect, and they support their claim with different parameters (most new licenses, most revenue from new licenses, most support revenue, most installations, most total revenue, etc.). It is very hard to find out the truth, but in general Oracle has the most large installations and the most revenue, SQL Server has the most (mainly smaller) installations, while PostgreSQL and MySQL are leaders in the open source database domain. An additional obstacle to really finding out who is the leader in what is the fact that the numbers are changing constantly and also, most companies use more than one (sometimes more than two or three) database technology.

All that aside, all of these database technologies are still relational, and only lately non-relational databases have started to enter the market and occupy several percentage points in the market share. But even that is a big success because for the longest time for any new project that was done, choosing database technology would only mean to choose which of the mentioned vendors to select, not actually choosing between relational or some other technology. Selection is usually made based on the size of the database, required availability and performance and license and support costs. In large part that was due to the fact that the relational models have worked well for a long time and fulfilled most needs that applications had, in most cases directly, and

only sometimes with workarounds. Developing entire new technologies to fulfill minority needs (that in most cases could be resolved with workarounds) never really made much sense.

That has started to change with emergence of heavier internet use, social networking and big data in general.

### 3.2 Relational Model

As mentioned, for the formal start of the relational database model we can take the year 1970, when Codd presented it in a paper [3]. Database models existed before that and the relational model was preceded by the hierarchical (that stores data in a tree like structure) data model [22] and the network model [22]. The term database model has been used in the Computer Science community interchangeably with different meanings. Generally, the database model can be described as a concept that describes a collection of conceptual tools for representing real-world entities to be modeled and the relationships among these entities [24]. Figure 3-1 [1] shows the evolution of database models with rectangles representing database models, arrows the influences and the ellipses theoretical developments.

The relational model, as proposed by Codd, introduced the idea of data organized in relations, in the form of two dimensional tables. As discussed in his paper, this model was designed to deal with the following issues:

- Hide from users how data is organized in a machine
- Protect user and application activities from internal data changes and growing data types
- Remove data inconsistency

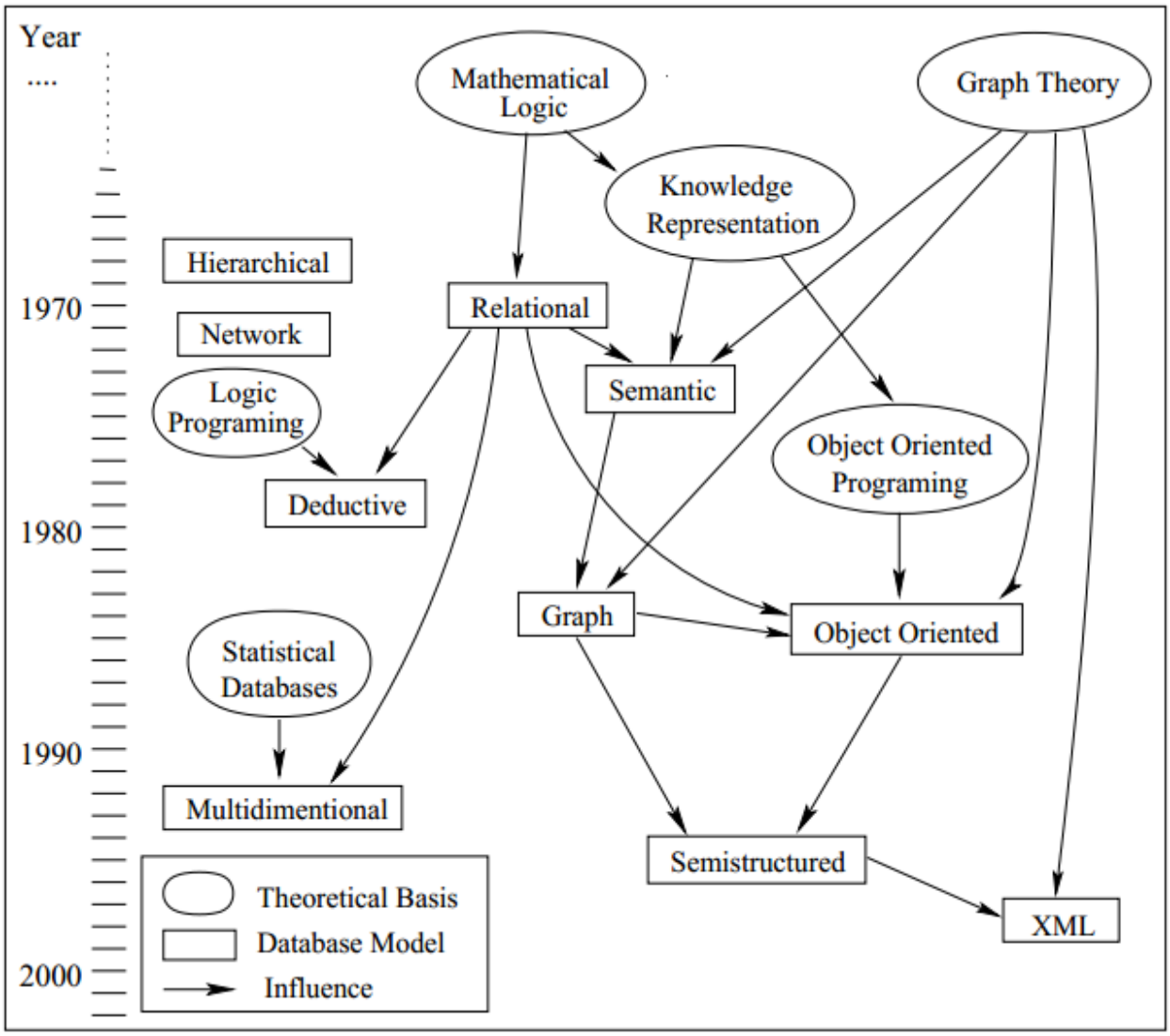


Figure 3-1. Database models evolution [1]

The relational model introduced many terms to explain its components, like entity, domain, relation, attribute, tuple, etc. Table 3-2 illustrates some of them and makes it easier to understand using some sample data.

Tenant

| UNIT | UNIT_TYPE | TENANT_NAME | TENANT_TYPE | ADDRESS |
|---|---|---|---|---|
| 509 | Residential | Michael Baros | Owner | 399 5th Ave |
| 11 | Commercial | Coffee Shop | Renter | 500 3rd Ave |
| 780 | Residential | John Travolta | Renter | 212 Main St |
| 311 | Commercial | Hair Salon | Owner | 1001 Wide Blvd |

Table 3-2. Sample data in relational model

In this example Tenant represents a relation with the name tenant as an entity. UNIT, UNIT_TYPE, TENANT_NAME, TENANT_TYPE and ADDRESS represent relational attributes. UNIT, UNIT_TYPE, TENANT_NAME, TENANT_TYPE and ADDRESS with assigned data are domains. The horizontal set of data ("780", "Residential", "John Travolta", "Renter", "212 Main St") represents a tuple. The number of tuples is called cardinality, while the number of attributes is called degree.

This example is only to illustrate relational model terminology, not to show how this particular data would be modeled (in relational model it would actually be broken into several tables, tied together with keys. Modeling like this more corresponds to NoSQL Column family model).

In the relational model the order of rows and columns (tuples and attributes) is not important and each row has to be unique.

Table 3-3 illustrates relational model terminology and their meanings.

| Relational model | Meaning |
|---|---|
| relation | table |
| domain | type of column |
| tuple | row |
| degree | number of columns |
| cardinality | number of rows |
| attribute | column |
| entity | table name |

Table 3-3. Relational model terminology translation

### 3.2.1 Primary-Foreign key concept

By default in the relational model identical rows are not allowed in relational tables to avoid ambiguity [52]. A Primary key (PK) is defined as a column (which can actually be a single or a set of columns) with a unique value (or a unique combination of values from different columns, while some of them can be repeated) that uniquely identifies each row in a relational table. The purpose of Primary keys is to guarantee that redundant data does not exist in the tables. In reality, this is sometimes bypassed by disabling primary keys for a specific need and redundant data sometimes does exist in tables, purposely or accidentally. Also, not all relational tables have primary keys set.

Primary keys in a relational model very often work with Foreign keys, who reference them. A foreign key value represents a reference to the tuple containing the matching candidate key value (the referenced tuple) [22].

A table could have multiple foreign keys that reference multiple primary keys in multiple tables. And multiple remote primary key values can exist in local foreign key tables, while the rest of data must differ (if a Primary Key exist in local table) to ensure that data is not redundant. Foreign keys ensure that data can't exist in a local table that already does not exist in a remote table with primary key on. As with Primary Keys, in reality Foreign keys also get disabled for specific procedures, which allows redundant data to enter tables. Sometimes that is cleaned up, but quite often is not.

### 3.2.2 Normalization

In [3] Codd described complex domains problem which should be broken down into independent sub-domains, but still related to each other by a series of Primary-Foreign key relationships. This guaranties data isolation in each subdomain and that each of them contain only data related to themselves, but as mentioned, keeping relationships with others through primary-foreign keys. This process is named normalization and means that instead of one big table, containing all of the data (which, depending on the domain, could result in massive data redundancy), there would be multiple, sub-domain related tables, tied together with relationships, which simplifies database operations.

This was further discussed in Codd's papers [53, 54, 55] that followed the initial one and introduced three types of normal forms. Later work expanded this even further, but in reality most of the work in actual databases gets resolved sufficiently with the original three normal forms.

As mentioned, this process was conceived and formalized by Codd. The first step in the normalization process is called the first normal form (1NF) and requires that all values in a

column of a table are atomic, which means they cannot be broken down further. A table is in first normal form if and only if each attribute of the table is atomic.

The second step in normalization is the second normal form (2NF). The second normal form requires that the relation must be in first normal form and any non-key column is dependent on the entire primary key (a primary key that could be a comprised of a set of columns).

The third normal form (3NF) states that a table is in third normal form if the table is in 2NF and that the non-primary key columns of the table are only dependent on the primary key and not on each other.

This process greatly enhanced the relational model and resulted in the huge success relational databases have had since then. But, as with everything, it can't be applied everywhere, and the first example where it does not really work is in data warehouses, but there are others. If it worked perfectly in every situation, there would be no need for NoSQL databases.

## 3.3 Graph Database Models

At the moment there are quite a few graph databases with different models which could be (currently) subdivided into three groups:

- Hypergraphs
- RDF (Resource Description Framework) triples
- Property Graphs

### 3.3.1 Hypergraphs

All graph databases are represented with nodes and relationships. Contrary to the property graph model, in which one relationship can connect only two nodes, one on each side, in the

hypergraph model one relationship, called a hyper edge, can connect to multiple nodes on both sides.

This is very useful in domains with many-to-many relationships which would be harder to design in the property graph model and would require more resources and higher maintenance costs. Figure 3-2 [2] shows an example of a many to many relationship, where multiple people own multiple cars.

Relationships between all of the nodes is represented by a single hyper edge, or hyper relationship, while in the property graph model multiple relationships would be used. This could be applied in similar way to represent relationships between parents and children. The advantage is simplicity, but the disadvantage is fine grained description. After all, the car needs to be registered to somebody and children in most cases have a mother and a father, who in turn have daughters and sons.

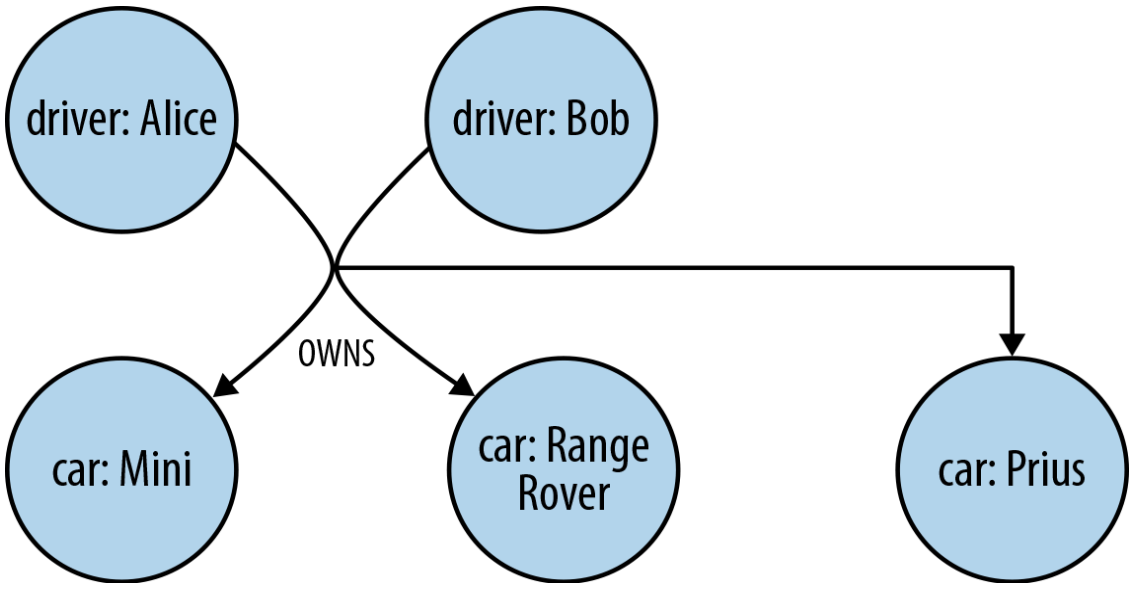


Figure 3-2. Simple hypergraph [2]

Because hyper-edges are multidimensional, hypergraphs comprise a more general model than property graphs. That said, the two models are isomorphic; it is always as possible to represent the information in a hypergraph as it is to use a property graph (albeit using more relationships and intermediary nodes). Whether a hypergraph or a property graph is best for you is going to depend on your modeling mindset and the kinds of applications you're building. [2]

### 3.3.2 RDF Triples

"A triple is a subject-predicate-object data structure. Using triples, we can capture facts, such as "Ginger dances with Fred" and "Fred likes ice cream." Individually, single triples are semantically rather poor, but en-masse they provide a rich dataset from which to harvest knowledge and infer connections. Triple stores typically provide SPARQL capabilities to reason about and stored RDF data" [2].

The origins of triple store databases are in the semantic web movement which has an interest in large knowledge inference with added semantics to links used in web resources.

"Triple stores fall under the general category of graph databases because they deal in data that—once processed—tends to be logically linked. They are not, however, "native" graph databases, because they do not support index-free adjacency, nor are their storage engines optimized for storing property graphs" [2].

"Triple stores store triples as independent artifacts, which allows them to scale horizontally for storage, but precludes them from rapidly traversing relationships. To perform graph queries, triple stores must create connected structures from independent facts, which adds latency to each query. For these reasons, the sweet spot for a triple store is analytics, where latency is a secondary consideration, rather than the OLTP (responsive, online transaction processing systems)" [2].

### 3.3.3 Property Graphs

Property graph models are models with simple and easy to understand characteristics:

- Comprised of vertices and edges, called nodes and relationships
- Nodes consist of (at least one) properties, which store attributes
- Named and directed relationship that mush have origin and destination nodes
- Standalone nodes are possible but not very useful in most cases
- Relationships could have properties, but not required like nodes.

Figure 3-3 represents the situation that was modeled with hypergraphs in Figure 3-1, this time modeled as a property graph and with relationship properties that give weight to the relationships, for a better fine grained description than possible with hypergraphs.

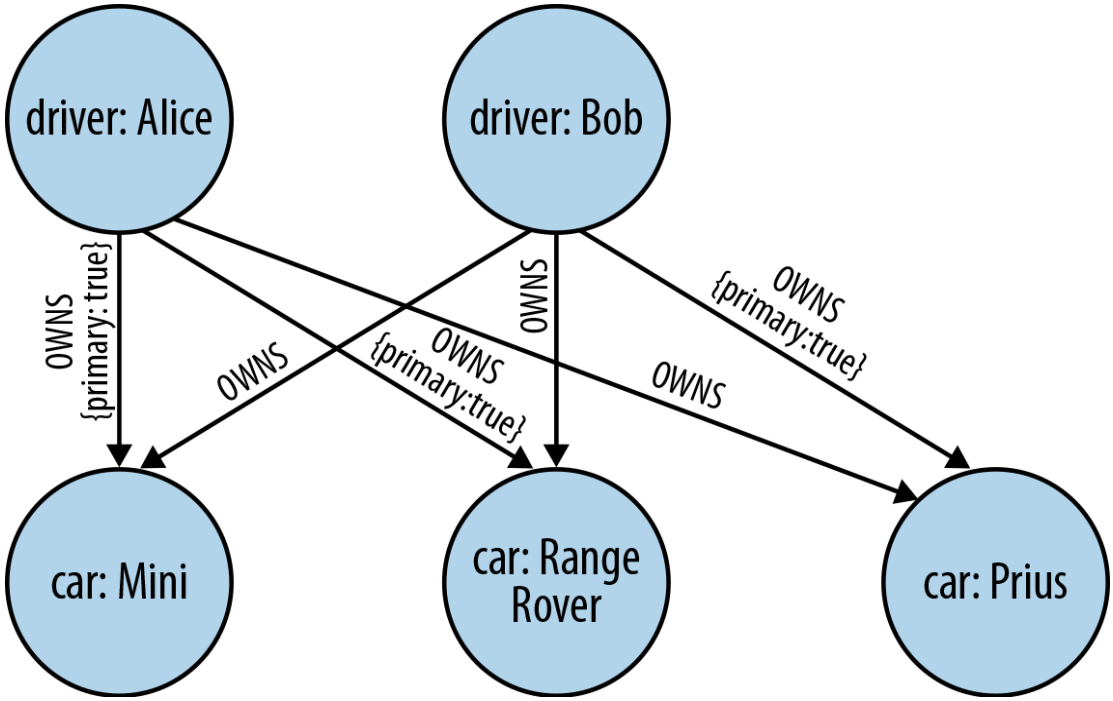


Figure 3-3. Simple property graph [2]

Primary ownership (car registered to) is described with the property on the relationship OWNS, while the secondary ownership is described with a simple OWNS relationship without additional properties.

Property (directed) graph databases are the focus of this thesis and are further discussed in detail in the following chapters.

### 3.3.4 Other graph database models

The models mentioned in this chapter are only those with current implementations and in use (even if only for the academic purposes), classified loosely and without specific databases being put in certain category. There are many more models that are not directly related to this thesis, among which many have never reached any form of implementation or any other work other than theoretical conceptualization. They are just mentioned here as summarized in [1]:

- DGV – Database Graph Views – this model proposes "the definition of personalized graph views of the data with management and querying purposes, and independent of its implementation" [1, 25]
- GDM – Based on GOOD (see below) and "adds explicit complex values, inheritance and n-ary

  symmetric relationships" [1, 26]
- GGL – Graph Databases System for Genomics – it comes from biology community "and highlights the advantage of storing Genome maps as graphs" [1, 27].
- GMOD – focused on user interfaces and "schema and instance are labeled

digraphs" [1, 28]

- GOAL [29]
- GOOD – Graph Object Oriented Data; in this model "schema and instances are represented by directed labeled graphs, and the data manipulation is expressed by graph transformations" [1, 30, 31, 32]
- GOQL – a model that proposed " SQL-Style query languages with explicit path expressions" [1, 33]
- Gram – a model in which data is organized as graph and "the schema is a directed labeled multigraph, where each node is labeled with a symbol called a type, which has associated a domain of values" [1, 34]
- GRAS – is a model that "uses attributed graphs for modeling complex information from software engineering projects" [1, 35]
- GraphDB – is an explicit model "which allows simple modeling of graphs in an object oriented environment" [1, 36]
- GROOVY - Graphically Represented Object-Oriented data model with Values is a "proposal of object-oriented db-model which is formalized using hypergraphs, that is, a generalization of graphs where the notion of edge is extended to hyperedge, which relates an arbitrary set of nodes" [1, 37]
- G−Base - is a proposed model "for representing complex structures of knowledge" [1,38]
- G−Log – is a proposed model with "a declarative query language for graphs" [1, 39]
- Hypernode - A hypernode model is based on notion that "hypernode is a directed graph whose nodes can themselves be graphs (or hypernodes), allowing nesting of graphs. Hypernodes can be used to represent simple (flat) and complex objects (hierarchical, composite, and cyclic) as well as mappings and records" [1, 40, 41, 42]].
- Hy+ - Based on the graph generalization concept and was used in the context of query

and visualization system [1, 43]

- LDM [44, 45] – Logical Data Model – This model "generalizes the relational, hierarchical and network models. The model describes mechanisms to restructure data plus a logical and an algebraic query languages" [1, 44, 45].
- O2 – is an "object oriented db model based on a graph structure." [1, 46]
- PaMaL – uses "uses patterns (represented as graphs) to specify the parts of the instance on which the operation has to be executed" [1, 47]
- R&M – Started with "the objective of modeling information whose structure is a graph" and proposed as "a semantic network to store data about the database" [1, 48]
- Simatic-XT – Developed for transport networks and communications this db model was proposed and it "merges the concepts of graph and object oriented paradigm, focusing in the (graph) structure of the data but not on the behavior of entities to be modeled" [1, 49]
- Model for Hypertext – Data Model for Flexible Hypertext Database Systems where "nodes represent web pages and hyperedges represent user state and browsing" [1, 50]
- W&S - Based on the graph generalization concept and was used in the context of "modeling of data instances and access to them" [1, 51]

Figure 3-4 represents all these models where nodes indicate models, arrows citations and dashed nodes represent related works in graph db-models [1].

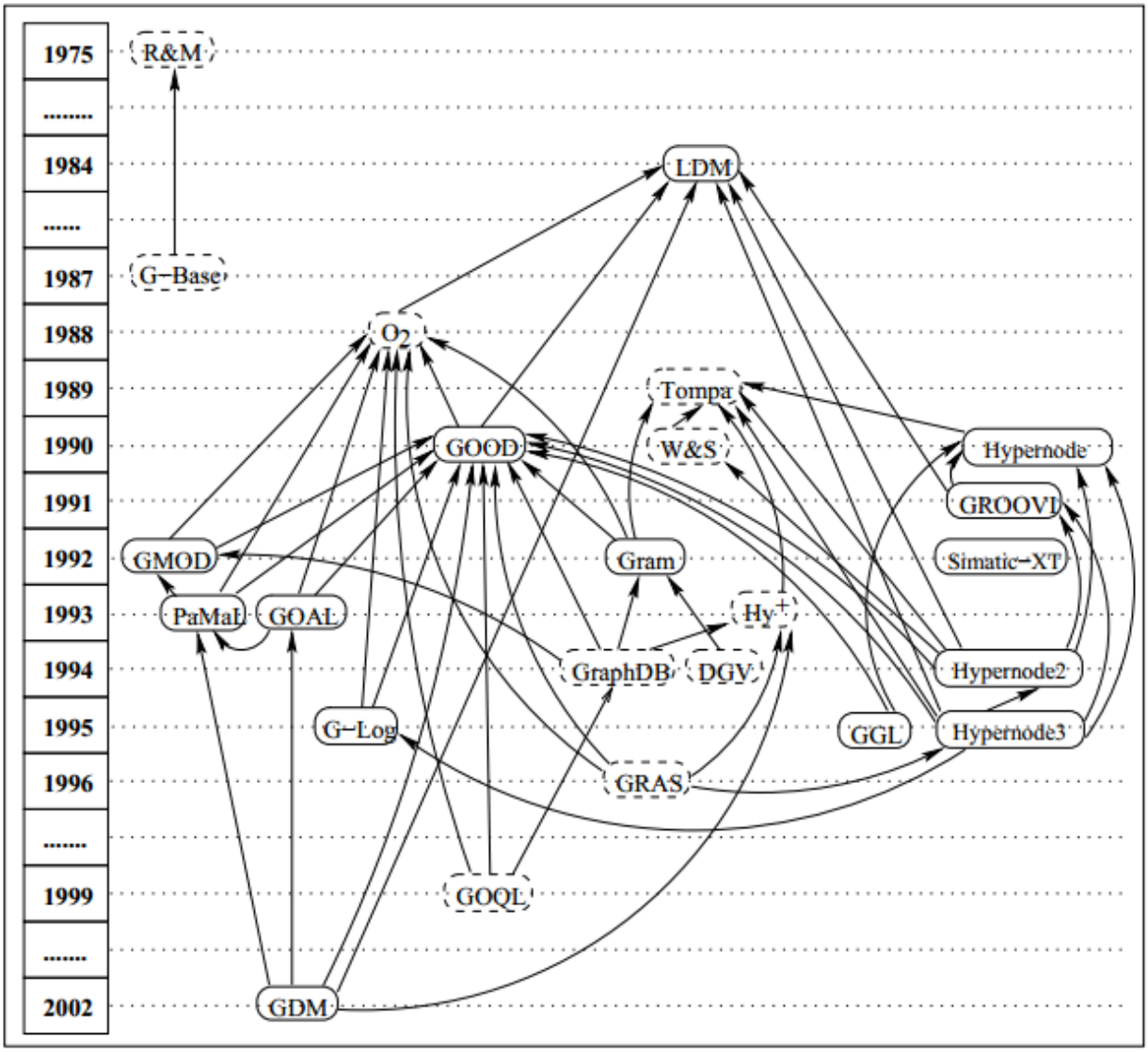


Figure 3-4. Graph database-models development [1]

### 3.4 Graph Database Modeling Best Practices and Rules

Since there is very limited amount of work done on formal data(base) modeling in the graph database domain, it is hard to expect many hard rules, advice or best practices. Some general rules are mentioned in [2] and they are listed below:

- Use nodes to represent entities, that is, the things that are of interest to us in our domain.

- Use relationships both to express the connections between entities and to establish semantic context for each entity, thereby structuring the domain.
- Use relationship direction to further clarify relationship semantics. Many relationships are asymmetrical, which is why relationships in a property graph are always directed. For bidirectional relationships, we should make our queries ignore direction.
- Use node properties to represent entity attributes, plus any necessary entity meta-data, such as timestamps, version numbers, etc.
- Use relationship properties to express the strength, weight, or quality of a relationship, plus any necessary relationship metadata, such as timestamps, version numbers, etc.

This is only place that mentions any of these suggestions, but without any justification or reasoning behind it nor how to differentiate between different elements of that model and what would happen if some other approach was taken.

### 3.5 Relation from Literature to Research Goals

In relational databases, with defined modeling rules, it is still left to individuals doing the modeling to solve particular problems and decide how to break data. But regardless of the approach taken, the final model must support required operations for any particular application. Similarly, to conceptualize different modeling approaches in graph databases, they must first support all operations required by the application. And to be able to compare and analyze them, they must support the same operations, to store, support and return identical data when queried.

Performance for relational databases is measured by way of capturing how fast data is returned and manipulated. Once it is confirmed that data and performed operations are the same,

the same principle applies to graph databases and the faster data is returned and/or manipulated, better the performance is considered to be.

How easy it is to expand a database and how well it scales is the next goal. This applies in a similar way to both the relational and the graph databases.

The remaining goals are all relatively same between the relational and the graph databases with the exception of hot nodes. They can't be measured in the relational databases, however a similar concept of highly referenced data, used by multiple foreign relations does exist.

# CHAPTER 4
# CASE STUDY

So far, all situations/graphs that were modelled as graph databases are pretty simple examples, two or three nodes that have few relationships and a handful of properties for each of the nodes. But in real life domains to be modelled can be quite complex with thousands, hundreds of thousands , even hundreds of millions of nodes and as many or more relationships between them, in domains like social networking. And both can grow by thousands a day or a minute. The domain of choice, in order to properly test the graph database and its uses would be one with numerous elements and relationships between them. Not necessarily a social network domain (although it would be very appropriate) for at least two reasons, it is not easy to generate or simulate user-generated content and it would maintain the mainstream opinion that graph databases are good for only a small set of applications and social networking is already one of them.

## 4.1 Domain

The domain chosen for creating graph databases with different approaches PropertyFirst and EntityFirst is the Condo Association Administration, or CondoConnect®. It has extensive relationships between many of its elements, but it is not strictly a social networking application (although it could easily fulfill that need between tenants if necessary, but it is unlikely) and it avoids the mentioned stereotypes. In addition, it could be as easily developed with a relational database solution, it makes a pretty general application. This kind of database application could be used for communication with tenants by the Condo board or by Property Managers, keeping a

list of tenants and their active contact information, units, board and general meetings, common property, employed personnel, requests for and the status of repairs, parking assignments and other assets, and other use cases. Such an application could start with local use, for one Condo Corporation, but it could be easily adopted to much wider use, from one community, city, region, country, or even wider, with very little effort on technical side to make room for new “tenants”.

### 4.1.2 Domain elements

Generally, elements that can be represented in the database model are, but not limited to:

- Building
- Tenant (owners and renters)
- Units (residential and commercial)
- Management (building and property)
- Board of Directors
- Maintenance tasks
- Meetings (Board and AGMs)
- Common property
- Externals (renting managers and service providers)

Elements in this application architecture could be divided in two groups, humans and non-humans, but would be equally treated as database elements, by both the database and the application, the only difference would be whether interaction is needed or required.

People in this domain are tenants (who can be owner or renters and owners who can be tenants or non-tenants), board members (who can also be tenants or non-tenants owners) from presidents to

members-at-large, building management (building manager who is also tenant, but not an owner and property manager that is neither a tenant nor an owner).

Non humans in this domain are units (residential units of different sizes where people live, commercial units, which are either rented from building or owned by non living-in-building commercial tenants), the building itself, the Board of Directors (but not board members who are humans) and any sort of meetings related to building or maintenance tasks requested by tenants and performed by building manager or external providers.

This example is focused on a single building. If we would add more buildings in the same city the street address becomes important. Add more buildings in other cities and the city location becomes important. The same applies for province, country and even continent. Nonetheless, no matter how much more information is added to the model, the back bone relationship remains the one between tenant-unit and building. Everything else gets built on top of that.

As discussed, the domain of condo corporation was chosen for this work, as it contains lots of information that is (or can be) highly connected and interdependent, with its various elements that all could be modelled as entities or properties. Not all relationships that were developed belong to this strict model (like marital status and other relationships between tenants, and some other, like military rank of some tenants or their TV show preference) but were intentionally included to enrich the model and make experiments more robust. This does not mean that the workable model could not contain this information, but they would need to be better protected due to possible security and privacy issues, and available only to those “who need to know”. This security access part could also be done with graph databases as they work very well for access management systems.

### 4.1.2 Domain model convention

When choosing an identifier for properties and entities the following convention is used for the simple reason of maximizing readability when writing queries in the implementation phase.

- For properties, a single term coined from one or more words, all lowercase, no underscores (i.e. buildingaddress from building address)
- For relationships, one or more words with underscores, all uppercase (i.e. LIVES_IN)
- For node labels, all lowercase single term
- Actual data follows rules of grammar

## 4.2 PropertyFirst Graph Database Model (PFM)

The Central point in this domain is Building; condos are located in it, people live in condos, management maintains it, and people are members of its board. Everything is connected to it, if not directly then through an intermediary. In addition to Building, two other main elements of this domain are Units and Tenants.

### 4.2.1 Building

Figure 4-1 shows sample information for the chosen attributes for this particular case. The beautiful thing with the schema free nature of graph databases is that any other building could have some of the attributes (or most of them) different without any issues for database itself. Nothing needs to be added, changed or removed to accommodate the introduction of the new entity (building). It can just be created with whatever attributes it needs. For instance, if the new building is three storey, walk-up building and does not have elevators and/or underground parking, those properties can simply be omitted for its model. And if it has some attribute which

is important for it and that this sample building does not (like placement in community like Willowgrow or Erindale) it can simply be added to a new building while not touching existing buildings at all. This is a big difference from relational databases, where building information would typically be stored in a table, with a predetermined structure and any changes that are necessary for new entities would mean that the current table structure for all buildings must be changed to accommodate new attributes. Null values (or some other approach) would have to be used for non-existent attributes of the new entities. Similarly to node attributes (properties) the same applies to relationships. If they are necessary, any new relationships including their properties can simply be added or removed, and no ground work is necessary. In relational databases these kinds of things are usually done through parent-child records with primary-foreign key relationships which need to be maintained and ground work is necessary ahead of time in case of any structural changes.


Figure 4-1. Sample building and its attributes in PFM

Similarly, models of Units and Tenants are presented in figures 4-2 and 4-3, respectively, with their own set of attributes, chosen for this case, but as with Building, they can be seamlessly changed for other cases, in this building or in any other.

### 4.2.2 Unit

Shown below in Figure 4-2 is a unit and its attributes. In this case its type is residential (which can also be commercial or something else in some other case), its absolute size is given in square feet (it could be sized in some other measurement type, like SI, depending on local preference) and its relative size is given in bedrooms and bathrooms, amenities and assets like assigned parking spot, storage availability and floor position location (which in this case is irrelevant as it can be deduced from the unit number, since it starts with 5, but can be useful in cases where unit numbers are not related to floors). Again, adding or removing attributes is easy and does not impact anything.


Figure 4-2. Sample Unit and its attributes in PFM

### 4.2.3 Tenant

Shown below in Figure 4-3 is Tenant and its attributes, chosen for this particular case. Many other attributes can be added to this or any other tenant without affecting any existing tenants (or any other entities, for that matter) in the database, which has never really been possible in relational databases.


Figure 4-3. Sample Tenant and its attributes in PFM

For instance, tenant Helen has the additional attribute that she is member of the board of directors, which Michael isn't and that is shown below in Figure 4-4.

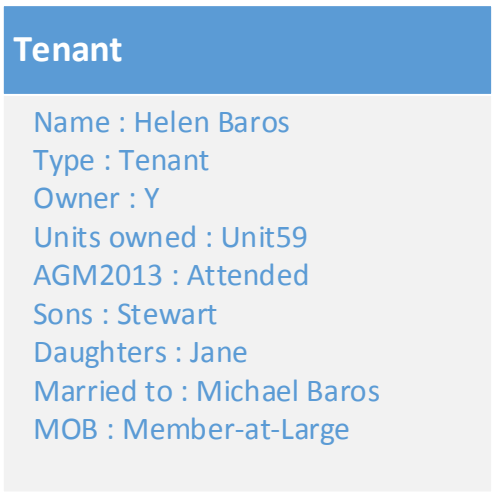


Figure 4-4. Sample Tenant with additional attributes in PFM

In the same way, children that are part of the same family and live in same the condo, can be represented in a very simple way, with just a few attributes as seen in Figure 4-5. Even the negative value for the attribute Owner is not really necessary; it was just put in for improved readability. Children could have some other attributes (like parents or mother and father, siblings or brother and sister) the choice was made not to show that here in order to present a simple tenant model with a minimum number of attributes used.

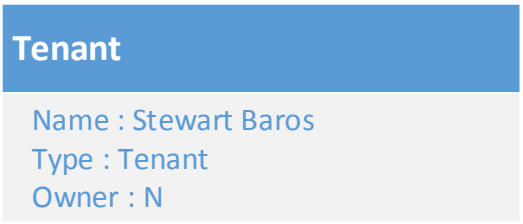


Figure 4-5. Sample Tenant with minimal attributes in PFM

### 4.2.4 Other elements

It would be beneficial to explain in detail models for all the other entities involved in this domain. For the purpose of this research, which is to show differences in modeling approaches and their impact on data stored in graph databases to save space and time, only a sample of the most prominent entities were detailed. If the goal was to develop an application for this particular domain with graph databases and this was a detailed technical how-to document all of them would be explained in detail. Here, these entities are only mentioned and include:

- Property and Building Managers and the Company that employs them, Property Management Company with all of their contact details.
- Board of directors, its members and the meetings they organize, schedule and hold, and distribution of notifications, minutes and decisions made in them.

- Maintenance requests with details that include issues, statuses, steps, dates, and people involved in them, from submitters to tasked people, either internal (building manager) or external (if escalation was necessary and external providers and/or partners were required)
- Renter information, including organized renting by the private Company that is in possession of multiple units and is involved in this in a commercial way and personal renting by individual owners who possess 1 or 2 units and can be living in or out of building.
- Commercial tenants, usually located on the ground floor and who can either be renting common property from the building directly (including rent amount) or could be in possession of said commercial units.

The full PropertyFirst model, with all "other elements" represented can be seen in Figure 4-7. By full model, it is meant that all elements are represented (building, residential and commercial tenants, residential and commercial units, building managers, maintenance requests, board of directors and its members, etc.), but not every tenant, unit, board member or maintenance request is represented. That would be equivalent to representing every row in the relational database in an ER model which would mean model representation for entire databases (in graph database as well) and that is highly impractical and could be considered impossible in real life.

### 4.2.5 Building-unit-tenant relationship (BUT relationship)

As mentioned, the relationship between a building and its units and tenants of various types is essential in this domain and consequently to its model, whatever the approach is, and deserves special attention. Figure 4-6 represents an excerpt from the full domain model and shows relationships between Building, Units and Tenants (BUT) occupying units in PFM.
The Model shows four tenants (Helen, Michael, Stewart and Jane) that live in the same unit, Condo 59, and that is represented by the relationship LIVES_IN (upper case relationships and lower case node identification is the unofficial standard in the Neo4J world, but it is not required, just recommended for easy distinction between the two). Also condo 59 is part of this building and that is represented with relationship PART_OF.

This is also a sample building-unit-tenant relationship which contains sample tenants (that are also owners, with exception of children) and a sample unit (that is a residential unit, inhabited by its owners, not renters). It can be expanded to include some of other elements discussed earlier but for simplicity and easier understanding of the full model in the following section, it is better to leave those out for the moment.

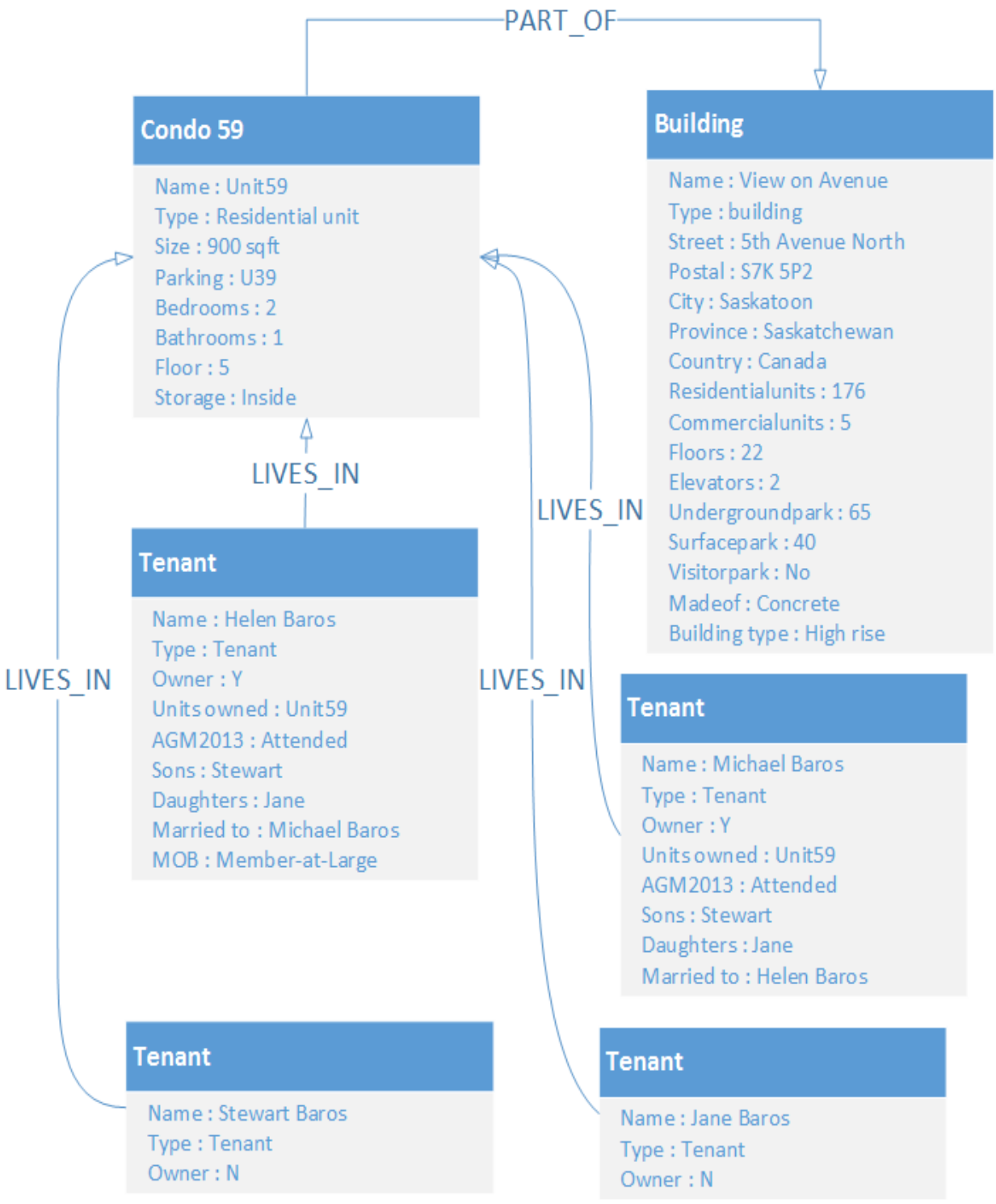


Figure 4-6. Building-unit-tenant relationship (BUT) in PFM

### 4.2.6 PropertyFirst full model

The PropertyFirst modelling approach follows (as much as possible) several rules:

- Identify the main node entities that can't be modeled as properties (Building, Units, Tenants, Board, etc.) and break them into separate nodes. For instance, even though for each tenant the property livesin can be set to the unit number that the tenants live in, that should not be done as unit is the main entity and needs to be modeled as such, since it has its own set of properties. Modeling all of that as properties for other nodes (tenant in this case) would be a nightmare and most of relationships would be impractical to follow and way harder, next to impossible, to query.
- Identify the main relationship entities that can't be modeled as properties (as relationships need to exist between nodes to make proper use of graph database and not to needlessly duplicate data). Some of this will overlap with first rule (mentioned livesin attribute instead of LIVES_IN relationship), but not everything.
- Most of the attributes need to be used as properties, unless it would mean missing a relationship, even if not one of the previously identified main ones.
- There can't be standalone nodes; at least one relationship is required. Although in graph databases they can exist (unlike relationships without both starting and ending node) it does not make sense to have them and is not at all practical for normal usage (and would add to data duplication and therefore its maintenance).

Figure 4-7 represents the full PropertyFirst model for CondoConnect® domain with intentionally increased font for node titles and relationships for better visibility.

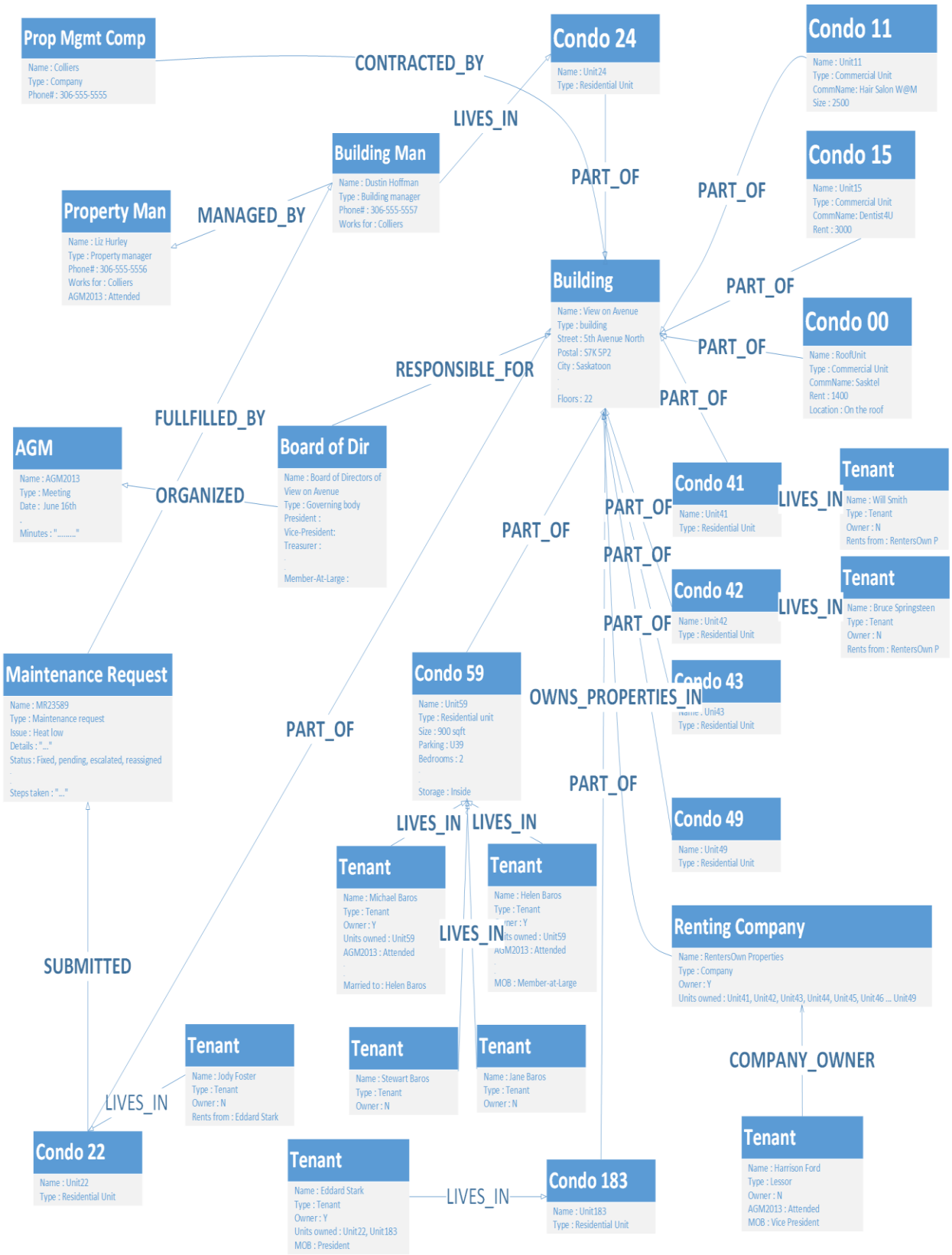


Figure 4-7. Full PropertyFirst model (PFM)

Some of the elements mentioned before but not shown graphically represented include:

- Commercial tenants (condos 11-15 and 00 which is the roof unit) that rent space in common areas
- Renting Company (owns condos 41-49) that is renting out condos that it owns
- Private renter that lives in condo 22 and rents from the owner who lives in condo 183 and owns both (relationship created as property in tenant node)
- Property Management Company which is contracted by the building and that employs Property Manager (who manages building and building manager, but does not live in building) and Building Manager
- Building Manager who is the building caretaker and takes care of Maintenance requests (submitted by tenant in condo 22), and lives in building
- Board of Directors that organizes Annual General Meetings (AGM) and is responsible for buildings state of affairs and financial standing

While this list seems extensive, it is never final, something new can always be added and a current element can be adjusted if it needs to serve some other purpose.

It might be interesting to compare excerpts from the code used to create the PF and EF database models. Here is the code (in Cypher [4], native SQL-like Neo4J language) used to create (nodes in blue, relationships in green) elements of building-unit-tenant section:

- Building

```
(building { name: 'View on the Avenue', type: 'Building', street: '399 5th Avenue North', postalcode: 'S7K 5P2', city: 'Saskatoon', province: 'Saskatchewan', country: 'Canada', continent: 'North America', residentialunits: 176, commercialunits: 5, floors: 22, elevators: 2, undergroundpark: 65, surfacepark: 40, visitorpark: 0, madeof: 'Concrete', buildingtype: 'High rise'}),
```

- Unit

```
(condo59  { name: 'Unit59',type: 'Residential unit',size: '900 sqft',bedrooms: 2,bathrooms: 1,parking: 'U39',storage: 'Inside',floor: 5 }),
```

- Tenants

  (michaelb { name: 'Michael Baros',type: 'Tenant', owner: true, unitsowned: 'Unit59', marriedto : 'Helen Baros', sons: 'Stewart Baros', daughters: 'Jane Baros', agm2013: 'attended' }),

  (helenb { name: 'Helen Baros',type: 'Tenant', owner: true, unitsowned: 'Unit59', marriedto : 'Michael Baros', sons: 'Stewart Baros', daughters: 'Jane Baros', memberofboard: 'Member at Large', agm2013: 'attended' }),

  (stewb { name: 'Stewart Baros',type: 'Tenant', owner: false }),

  (janeb { name: 'Jane Baros',type: 'Tenant', owner: false }),

- Relationships between them

  (condo59)-[:PART_OF]->(building),

  (michaelb)-[:LIVES_IN]->(condo59),

  (helenb)-[:LIVES_IN]->(condo59),

  (stewb)-[:LIVES_IN]->(condo59),

  (janeb)-[:LIVES_IN]->(condo59),

## 4.3 EntityFirst Graph Database Model (EFM)

Contrary to the PropertyFirst modeling approach, here the goal is to model and fit as many attributes as possible as entities, not properties. Basically, attributes are transformed into relationships and their values into nodes. One thing that can be noticed is that if the value for a particular attribute is generic enough in its nature, then the node could be reused by other parts of the data and connected to it with relationships.

### 4.3.1 Building

Expanded entities, being nodes and relationships, are represented in figure 4-8, and it is easily noticeable how they affect building modeling with this technique.

The first thing to notice is the increased number of nodes and relationships that connects them. In addition to the one original node there are now ten new ones, and only because not all the new nodes were presented in this model (done to make it more readable, with all the existing

new nodes and relationships it would be pretty hard to recognize what is what). Otherwise there would be 15 new nodes. The missing nodes are three nodes about parking, building type (high rise), building material (concrete) and floors, all accompanied with appropriate relationships to building.

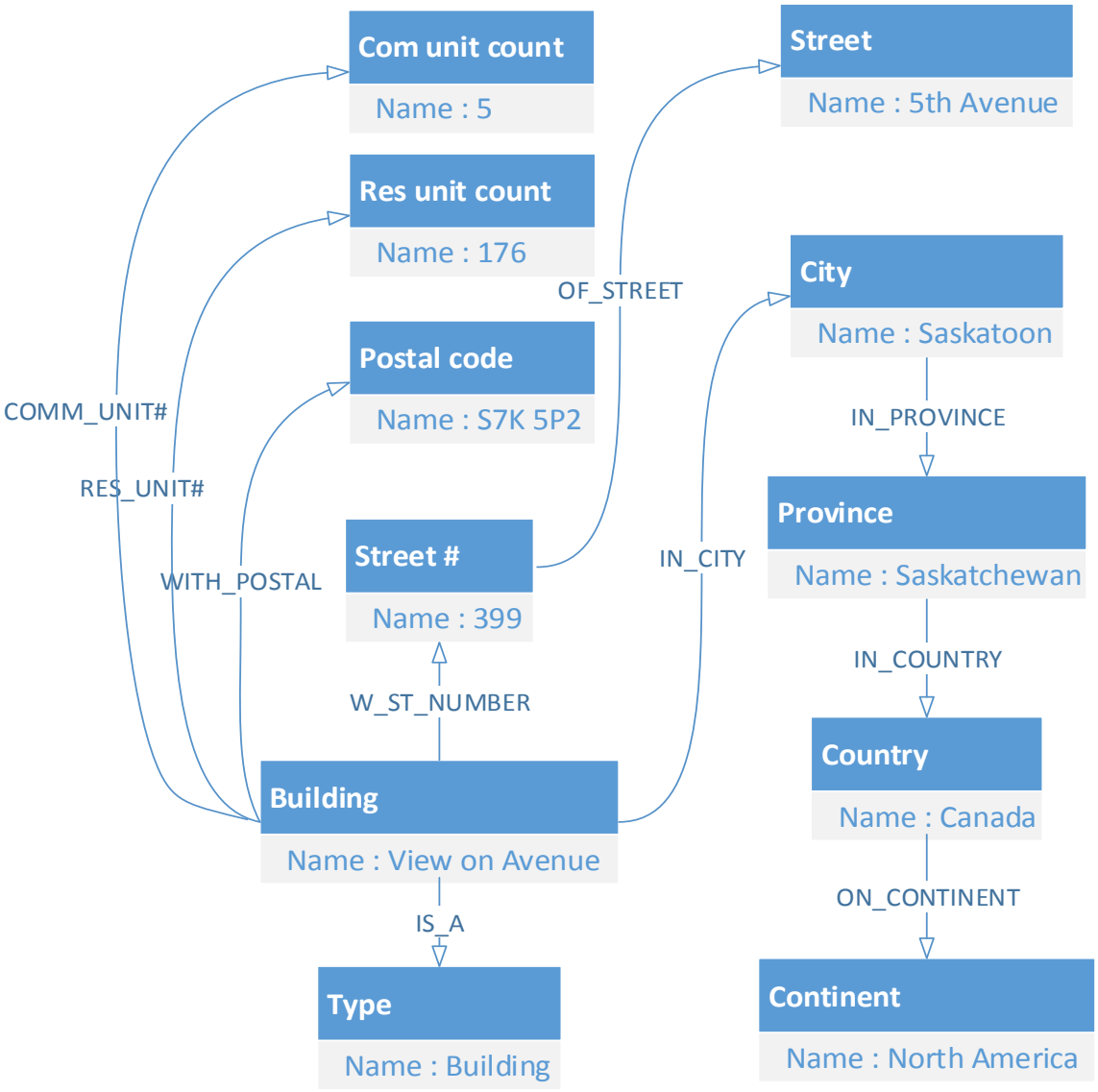


Figure 4-8. Sample building and its attributes in EFM

Another entity that significantly increased are relationships that the building is involved in directly. Although most new nodes have one relationship, most of them are connected to the building node, even those related to location. Node entities Street#–street and city–province–country–continent, which are related to each other in this case, could be related directly to the building node. Therefore the number of relationships the building node is directly involved with jumps significantly. However, several of the nodes can be reused for other buildings that might be added to database. They include entity type node (IS_A relationship), location nodes (city for buildings from the same city, province for buildings in other cities in same the province and so on), all parking nodes (if they match in assigned value and/or naming, if not new ones need to be created), building type, building material, floors (again, only applicable if values match, floors node can't be reused for buildings with 15 or 27 floors).

Everything that was mentioned for PFM about the easy way to add different information applies here too, only instead of adding new properties, new nodes and relationships need to be added or exiting ones reused, if possible and practical.

### 4.3.2 Unit

Modeled unit in EFM is presented in Figure 4-9.

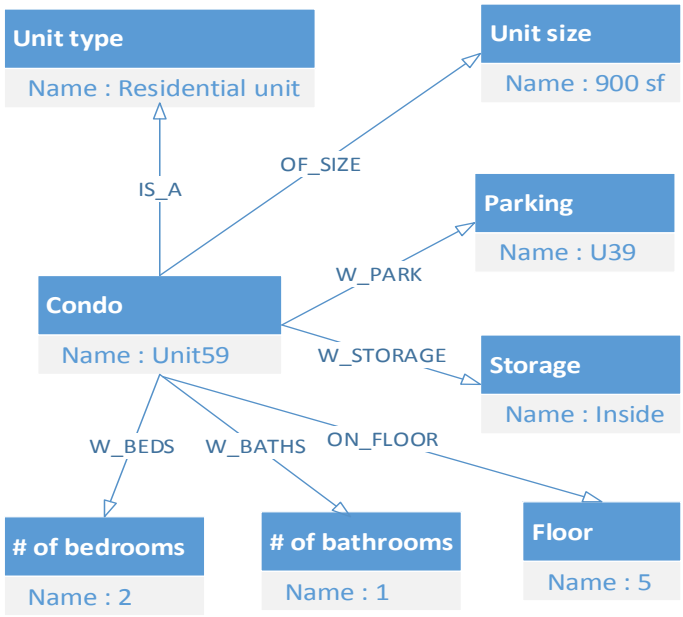


Figure 4-9. Sample Unit and its attributes in EFM

Again, several new entities (one node and one relationship for each pair of attribute and its value) are added to the model, but many of them can be reused for other units. While assigned parking and external storage are unique and can't be reused, unit type, unit size, internal storage, bedrooms and bathrooms can all be reused (for matching values only). Floor can be reused by all units on the same floor in the same building and depending on design, could be used to model any other buildings, as well.

### 4.3.3 Tenant

Figure 4-10 represents tenants with properties converted into entities. As discussed for the other main nodes, an increased number of entities can be observed as well as the model's ability to reuse some of them. For readability reasons some external nodes were added (board, AGM

and unit) because those were set as node properties in PFM. On the other hand, some attributes presented in PFM (those between tenants, related to family ties) were omitted as those relationships would just complicate this picture.

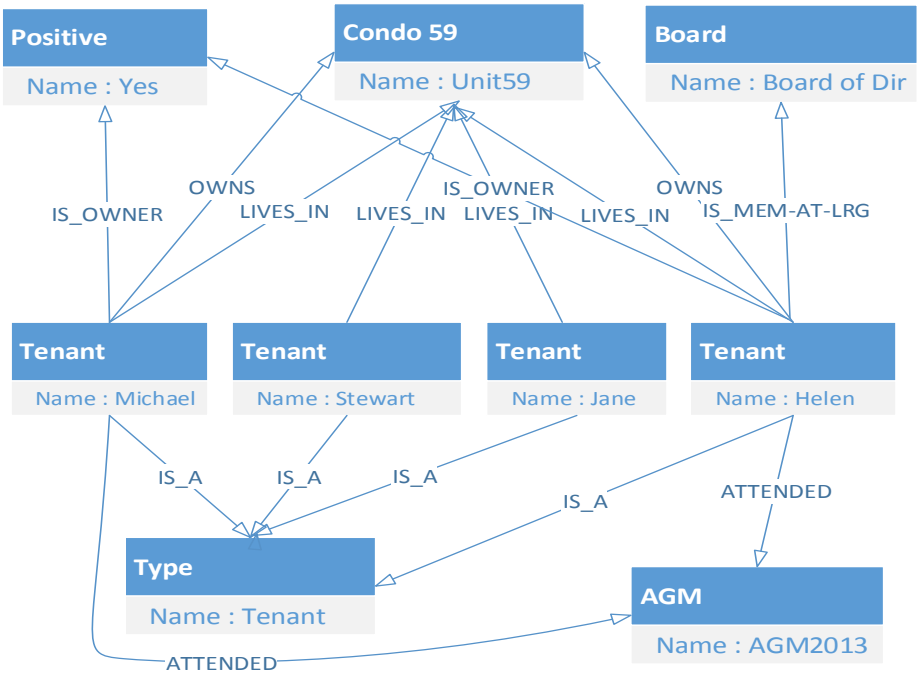


Figure 4-10. Sample Tenant and its attributes in EFM

### 4.3.4 Other entities

Everything mentioned for other entities in PFM (as listed in 4.2.4) and for the three main nodes in EFM in the modeling sense applies to other nodes in the domain too.

### 4.3.5 Building-unit-tenant relationship

Figure 4-11 represents an excerpt from the full domain model and shows the relationships between Building, Units and Tenants (BUT) occupying units in EFM.

Following the same standard, some of the nodes and relationship were omitted from this model, for better understanding and readability, and this applies to all three “sections” of the model.

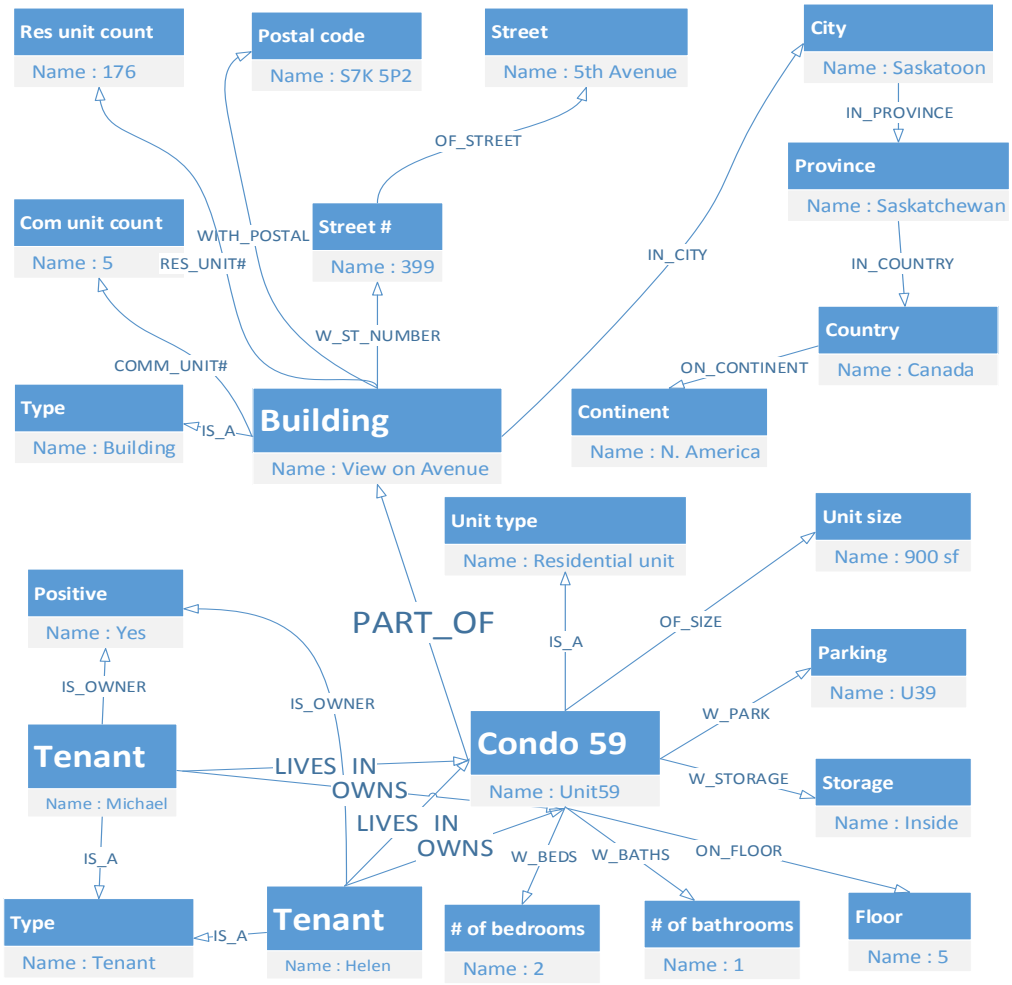


Figure 4-11. Building-unit-tenant relationship (BUT) in EFM

### 4.3.6 EntityFirst full model

The EntityFirst modelling approach follows a set of slightly different rules:

- In addition to main nodes, identify all the attributes of the main nodes that could be modeled as entities (node/relationship pair instead of property/value pair) and break them into nodes. The goal is to, wherever possible, have just one remaining attribute (property) per node as it is not possible nor useful to have a node without any properties. During this process, choose meaningful relationships to connect new nodes to the original main nodes
- Identify any relationship properties that could be modeled as entities and apply same logic as for nodes. An example here could be the relationship MEMBER_OF_BOARD with the property mob type being President or whatever broken into a new relationship called IS_PRESIDENT_OF_BOARD.
- There can't be standalone nodes, at least one relationship is required.
- Reuse nodes wherever appropriate, i.e. whenever possible without the chance of harming anything in the existing model or anything that we could already be aware of for future model or addition. Even if mistakes are made and nodes are reused that later prove to be incompatible for reuse, it is not difficult to change that, but does require additional work.
- Reuse relationships in a similar fashion, only be aware of their nature. If there is a need to differentiate very similar relationships (HUSBAND_OF or WIFE_OF instead of MARRIED_TO) as the application demands or it is a requirement for quicker and more precise queries, do as much. If it even turns out not to be the best approach, it is not that difficult to accomplish. Like in case of nodes, can be relatively easy to change, but it does require additional work.

Figure 4-12 represents full EntityFirst model.

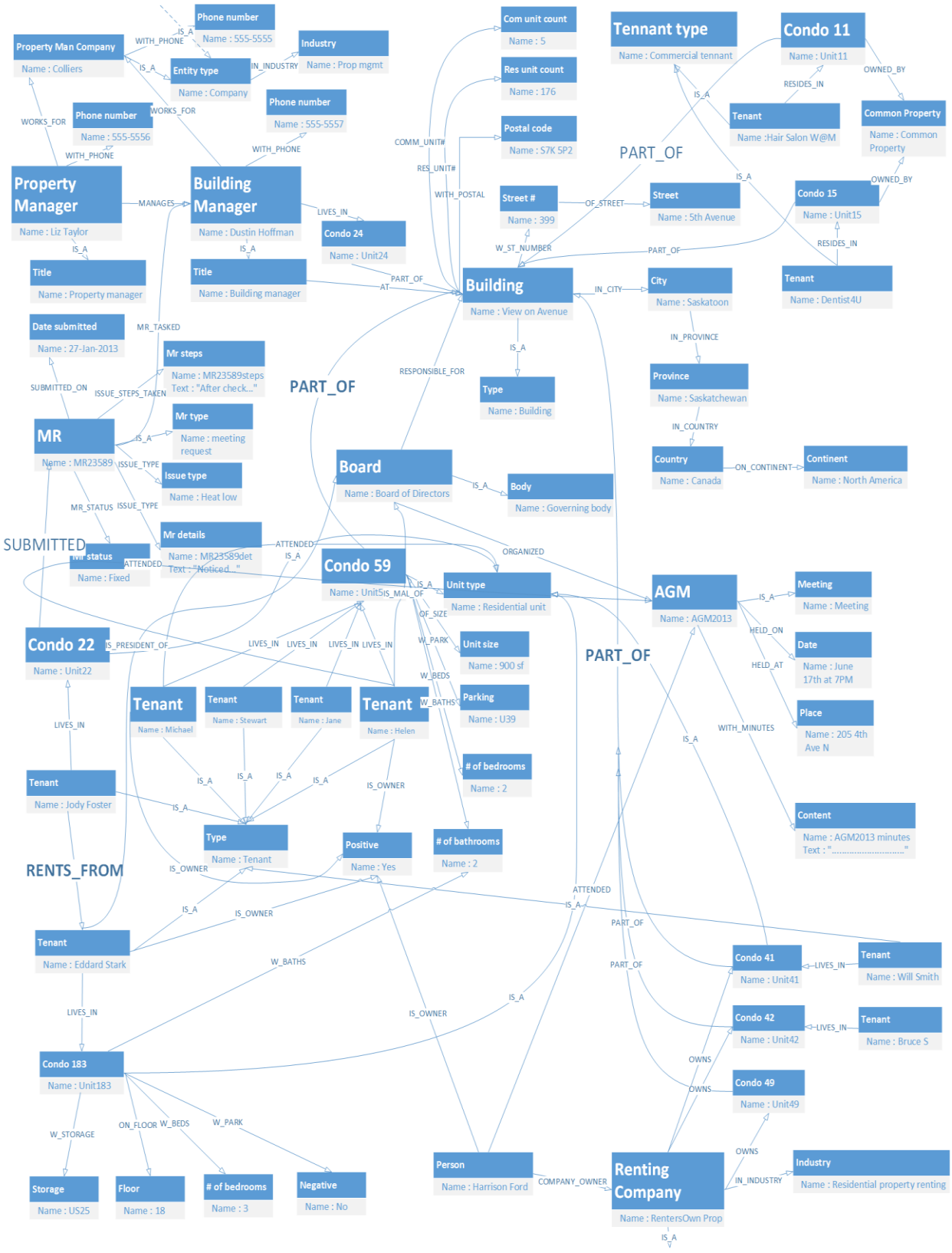


Figure 4-12. Full EntityFirst model (EFM)

Almost all of the attributes that were modeled as properties in PFM are now modeled as entities in EFM. The only exceptions in this particular case are “text” attributes in the maintenance requests (for maintenance request details and feedback) and the minutes for the AGM, because it would be difficult to uniquely identify and apply text or parts of text if separated from its identifier (which is set as “name” in all cases for simplicity faster query writing and maintenance), and above all impractical. The model also clearly identifies several cases of node reuse (like IS_A tenant, OF_SIZE unit size, W_BEDS relative condo size, etc.) which saves space, helps prevents fragmentation and potentially improves performance, but increases graphical model representation complexity.

The following is an excerpt from the Cypher [4] code used to create (nodes in blue, relationships in green) EFM model’s original three nodes (and entities created from their properties in PFM):

```
(nodetypebuilding { name: 'building' }),
(buildingstreet5thaven { name: 'Fifth Ave N' }),
(city { name: 'Saskatoon' }),
(province { name: 'Saskatchewan' }),
(country { name: 'Canada' }),
(continent { name: 'North America' }),
(number0 { name: 0 }),
(number1 { name: 1 }),
(number2 { name: 2 }),
(number5 { name: 5 }),
(number22 { name: 22 }),
(number40 { name: 40 }),
(number65 { name: 65 }),
(number176 { name: 176 }),
(number399 { name: 399 }),
(unitsize900 { name: '900 sqft' }),
(concrete { name: 'Concrete' }),
(highrise { name: 'High rise' }),
(nodetyperesunit { name: 'Residential unit' }),
(storageinside { name: 'Inside' }),
(tenant { name: 'Tenant' }),
(affirmative { name: 'Yes' }),
(negative { name: 'No' }),
(agm2013 { name: 'AGM 2013'}),
(mobmematlarge { name: 'Member-at_Large' }),
(building { name: 'View on the Avenue' })
(postalcode { name: 'S7K 5P2' }),
(building)-[:IS_A]->(nodetypebuilding),
(building)-[:WITH_STREET_NUMBER]->(number399),
```

```
(number399)-[:ON_STREET]->(buildingstreet),
(building)-[:WITH_POSTAL]->(postalcode),
(building)-[:IN_CITY]->(city),
(city)-[:IN_PROVINCE]->(province),
(province)-[:IN_COUNTRY]->(country),
(country)-[:IN_CONTINENT]->(continent),
(building)-[:RESIDENTIAL_UNITS_NUMBER]->(number176),
(building)-[:COMMERCIAL_UNITS_NUMBER]->(number5),
(building)-[:NUMBER_OF_FLOORS]->(number22),
(building)-[:NUMBER_OF_ELEVATORS]->(number2),
(building)-[:NUMBER_OF_UNDERG_PARKING]->(number65),
(building)-[:NUMBER_OF_SURFACE_PARKING]->(number40),
(building)-[:NUMBER_OF_VISITOR_PARKING]->(number0),
(building)-[:BUILDING_MATERIAL]->(concrete),
(building)-[:TYPE_OF_BUILDING]->(highrise),
(condo59 { name: 'Unit59' }),
(parkingu39 { name: 'U39' }),
(condo59)-[:PART_OF]->(building),
(condo59)-[:IS_A]->(nodetyperesunit),
(condo59)-[:SIZE_OF]->(unitsize900),
(condo59)-[:WITH_BEDROOM]->(number2),
(condo59)-[:WITH_BATHROOM]->(number1),
(condo59)-[:WITH_PARKING]->(parkingu39),
(condo59)-[:WITH_STORAGE]->(storageinside),
(condo59)-[:ON_FLOOR]->(number5),
(michaelb{name:'Michael Baros' }),
(helenb { name: 'Helen Baros' }),
(stewb { name: 'Stewart Baros' }),
(janeb { name: 'Jane Baros' }),
(michaelb)-[:IS_A]->(tenant),
(michaelb)-[:IS_OWNER]->(affirmative),
(michaelb)-[:OWNS]->(condo59),
(michaelb)-[:LIVES_IN]->(condo59),
(michaelb)-[:MARRIED_TO]->(helenb),
(michaelb)-[:IS_FATHER]->(stewb),
(michaelb)-[:IS_FATHER]->(janeb),
(michaelb)-[:ATTENDED]->(agm2013),
(helenb)-[:IS_A]->(tenant),
(helenb)-[:IS_OWNER]->(affirmative),
(helenb)-[:OWNS]->(condo59),
(helenb)-[:LIVES_IN]->(condo59),
(helenb)-[:MARRIED_TO]->(michaelb),
(helenb)-[:IS_MOTHER]->(stewb),
(helenb)-[:IS_MOTHER]->(janeb),
(helenb)-[:MEMBER_OF_BOARD]->(mobmematlarge),
(helenb)-[:ATTENDED]->(agm2013),
(stewb)-[:IS_A]->(tenant),
(stewb)-[:IS_OWNER]->(negative),
(stewb)-[:LIVES_IN]->(condo59),
(stewb)-[:IS_SON_OF]->(michaelb),
(stewb)-[:IS_SON_OF]->(helenb),
(janeb)-[:IS_A]->(tenant),
(janeb)-[:IS_OWNER]->(negative),
(janeb)-[:LIVES_IN]->(condo59),
(janeb)-[:IS_DAUGHTER_OF]->(michaelb),
(janeb)-[:IS_DAUGHTER_OF]->(helenb),
```

It is shown a bit differently than for PFM as nodes and relationships are not separated, almost intermingled, order based on each dependency. There is immediately a visible increase in number of nodes, and a surge in the number of relationships and lines of the code.

### 4.4 Natural or Mixed Graph Database Model

When approaching a new application in just about any conceivable domain, neither of these modeling techniques would normally be used in its pure form. One would use some combination or mix. The most that can be hoped when adopting these models, would be PropertyFirst or EntityFirst centric models, where the best from each model would be taken, but with elements from the other.

For instance, the very first steps in this research proved something similar. The initial PFM model was later reviewed, reconsidered and re-done and replaced with the current PFM model, while the original PFM model was renamed as merged, mixed, combined or natural. Four relationships were removed from it (OWNS, RENTS_FROM, ATTENDED and MEMBER_OF_BOARD) and replaced with properties on nodes (unitsowned, rentsfrom, agm2013 and memberofboard respectively, on the tenant nodes). At same time, this helped to develop and conceptualize more precise rules to establish different models. For PFM, as much as possible, information is to be stored as properties inside nodes, with at least one relationship for each node and maximum two outgoing relationships, without restrictions on incoming relationships. There can't be any restrictions on the incoming relationships, for practical reasons. For instance, all condos will have an outgoing relationship with building, which makes 175 incoming relationships for building, but all of the same relationship type. Also, each tenant will have just one outgoing relationship to condo, but that condo will have incoming relationships

from each of the tenants. All other relationships that condos or tenants might have between each other should be set as properties.

For EFM, the rules are that each node must have at least one property (like name or ID) and two at most (to accommodate long text values that don't work well if set as separate entities).

The mixed or natural model was not represented here as it is, basically, any combination of the two presented extremes. The model developed in the initial phase of this research was just one example. It would probably look quite different if done by somebody, or even by the same person at a different time or under the different circumstances.

The two models presented were developed and designed as opposite extremes to evaluate the different modeling approaches and to help determine the best uses for one or another approach for different modeling goals and to better apply different techniques for different uses and use cases.

## 4.5 Modeling Conclusions and Lessons Learned

During the entire process, domain selection, initial modeling attempts and consequent remodeling steps, several details were observed, some expected and some unexpected.

### 4.5.1 Approach

The domain of the condo corporation was chosen for this work as it contains lots of information that is (or can be) highly connected and interdependent, with its residential and commercial tenants and units and their attributes, the different features of buildings (high rise, walk up, frame, concrete, number of elevators, parking availability) and their location, the management that maintains the building, the maintenance work being done, the board that runs

the building and their meetings, and so on. All of this could be modeled as relationships or properties. As mentioned, additional information was included (marital status and other relationships between tenants or military rank and TV show preference of some of the tenants) which does not strictly belong in this domain, but could be added for different reasons in real life. They were added here to increase complexity and robustness of database and allow wider database queries. This does not mean that a workable model could not contain this information. That kind of the information would be just need to be better protected due to security and privacy issues and available only to those “who need to know” which could also be accomplished with graph databases as they work very well for access management systems.

### 4.5.2 Modeling lessons learned

It is very well possible to do the modeling on a "positive" basis only. Positive in this case means, when there is no data or if the value is "no" or "false" the property can be left out and in application code development phase assume that it does not exist or that is not applicable.
For instance, when modeling condos which do not have parking stalls or storage assigned, properties for parking and storage can be omitted, instead of being set to “false”:

- with parking and storage excluded

```
(condo179  { name: 'Unit179',type: 'Residential unit',size: '900 sqft',bedrooms: 2,bathrooms: 1, floor: 17 }),
```

- with parking and storage included

```
(condo179  { name: 'Unit179',type: 'Residential unit',size: '900 sqft',bedrooms: 2,bathrooms: 1,parking: 'No',storage: 'No',floor: 17 }),
```

But it was left in on purpose, for two reasons. It is slightly easier and more readable to write queries, and when present they add to the total number of properties which better illustrates "PropertyFirst" concept of modeling. Incidentally, they also add to the total size of the data and

in EFM create more entities. Another reason to keep them might be to use them in data de-duplication in future work.

Some of the tenants have military ranks of Captain, Colonel or Lieutenant. These were included with a property “rank” in PFM (with specific values of Captain, Colonel, etc.) or relationship in EFM (with specific values as nodes) “WITH_RANK”. At same time the title of Dr. for some tenants has been included with tenant’s name, although it was possible to follow the same logic and represent it with a property or relationship.

Properties that represent the same information can be set on different nodes. For instance, property ownership of commercial units in building can be set in nodes representing units as "ownedby" or in building itself with properties "unitsowned" (that is same as one used in modeling residential units and tenants, set for tenants) or "comunitsowned" to be able to logically separate it easier from residential units. Separation can also be achieved by involving the property "type" but that would complicate search queries and most likely affect performance as additional processing is necessary.

Example:

a) modeled in building node:

(building { name: 'View on the Avenue', type: 'Building', street: '399 5th Avenue North', postalcode: 'S7K 5P2', city: 'Saskatoon', province: 'Saskatchewan', country: 'Canada', residentialunits: 180, commercialunits: 4, floors: 22, elevators: 2, undergroundpark: 65, surfacepark: 40, visitorpark: 0, madeof: 'Concrete', buildingtype: 'High rise', comunitsowned: 'Unit11, Unit15, Unit17, Unit19' }),

b) modeled in unit node:

(condo11 { name: 'Unit11',type: 'Commercial unit',size: '2500 sqft', commercialname: 'Hair Salon W@M', ownedby: 'View on the Avenue', parking: 'Metered street parking', storage: 'Inside', floor: 1, rent: 2200, location:'West side' }),

When modeling building in the EntityFirst model it is possible to model the building address in one node with the one property or multiple nodes, distilled by street name, street number or even street direction (North, South, East, West or even further NW, SE, etc.).

Example:

a) modeled in one node:

```
(buildingaddress { name: '399 - 5th Ave North' }),
```

b) modeled in multiple nodes:

```
(buildingnumber { name: 399 }),
(buildingstreet { name: 'Fifth Ave ' }),
(buildingstreetdirection { name: 'North' }),
```

Modeling for geographical location (city, province, country, continent) can be done directly from building to all of these nodes or "in a row" (in a row here means in sequential way, from the first entity to the second, than from second to the third and so on) from building to city, and then from city to province, from province to country and finally from country to continent. While doing it directly from building to each of the location nodes might be little faster for searching as there is just one degree of separation (instead one for city, two for province and four for continent) , doing it “in a row” makes for a better opportunity to test the Neo4J ability for fast traversal. It also allows for some relationship reusability as we only need to make relations from city to continent once and each new building needs only to establish relationship to city (for same city, for different city that that city first needs to establish a connection  to province, but that is all). Similar reusability applies for modeling any kind of node that has numbers for values, like the number 175 for number of residential units.

When modeling for parking and storage, they are not considered as reusable since they most likely apply only to this building, or at least their labels do (even though they could be shared between some buildings). So while size and other entities that are always represented by numbers only (like number of bedrooms or floor number) can be shared, parking and storage (with S01, U23 or US28 as their labels) are considered unique. That only applies to distinctive parking and storage; 'Inside' and 'No' are reusable. The thing to consider when doing this is the fact that reusable nodes tend to have much more (mostly) incoming relationships, as they replace several other nodes and accumulate all of their relationships.

One of differences between PFM and EFM modeling approaches can be seen in the fact that some properties in the PFM model that can be modeled as one (for instance unitsowned: 'Unit41, Unit 42, Unit43, Unit49') while in EFM model they have to be modelled as one entity for each distinct property value (for instance, each of properties for unitsowned has to be modeled as one relationship between owner and unit like this:

(rntcmpny)-[:OWNS]->(condo41),
(rntcmpny)-[:OWNS]->(condo42),
(rntcmpny)-[:OWNS]->(condo43),
(rntcmpny)-[:OWNS]->(condo49))

An example of the opposite situation is when properties like marriedto or roommate. Property roommate with value 'Darth Maul' is set for node with label JarJarBinks and property roommate with value 'Jar Jar Binks' is set in node labeled DarthMaul in the PFM, see below.

(JarJarBinks { name: 'Jar Jar Binks',type: 'Tenant', owner: true, unitsowned: 'Unit125', roommate: 'Darth Maul' }),
(DarthMaul { name: 'Darth Maul',type: 'Tenant', owner: true, unitsowned: 'Unit125', roommate: 'Jar Jar Binks' }),

All this can be replaced with a single relationship in the EFM:

((marthas)-[:ROOMMATE]->(arose))

This is only true for relationships where direction is not important; where it is important (father->daughter) two relationships must be used.

# CHAPTER 5
# IMPLEMENTATION

Implementing these modeling approaches is done in several steps that depend on each other and are explained as follows.

The first step is to setup hardware to be comparable for both models and best way to accomplish this is through two identical virtual machines hosted on the same physical machine, with only one booted at the time of conducting experiments.

The second step is to implement a full database for one full data set for each modeling approach. In this domain, the data set is considered to be one full implementation for one building with all of its modeled entities and properties as previously explained.

The third step is to conduct experiments on each virtual machine, to make sure that both implementations can support running the same operations (queries), note the results and compare them.

The fourth step is to alter the databases and add multiple data sets to its model, i.e. multiple buildings, accompanied with all of its entities and properties.

The fifth step is to repeat the experiments on the expanded databases, note the results, and compare them to each other and to the results from the third step. Some of the experiments in fifth step are identical to those in the third step, but some are different as they can only be applied to data with multiple data sets that will produce aggregate data as results.

The sixth step is to compare all other viable results and note differences between implementations, including size on disk, possible creation of hot nodes, occurrences of data duplication due to the modeling approach used and query complexity.

## 5.1 Hardware and Software

Both instances have been implemented on a Windows 8 host (64-bit, dual core i7 CPU with 16GB RAM) with a VMware Workstation 9.0.4 running two virtual machines with a Windows 8.1 guest OS (64-bit with 8GB RAM and all available CPU) and a Neo4J 2.15 Enterprise Edition databases release.

## 5.2 Tools

The tools used here are entirely native to Neo4J. The Cypher query language is used for both running statements to create and modify any and all parts of each modeling approach and running queries to search or modify existing data. Admin web console is also a Neo4J tool, which comes with the installation by default.

### 5.2.1 Cypher

This very powerful, SQL-like, query language that can be used for just about anything in Neo4J database.

Sample code shown below is an excerpt of the code used to create the first data set in PFM:

```
CREATE (building { name: 'View on the Avenue', type: 'Building', street: '399 5th Avenue North', postalcode: 'S7K 5P2', city: 'Saskatoon', province: 'Saskatchewan', country: 'Canada', continent: 'North America', residentialunits: 176, commercialunits: 5, floors: 22, elevators: 2, undergroundpark: 65, surfacepark: 40, visitorpark: 0, madeof: 'Concrete', buildingtype: 'High rise'}),
        (condo21  { name: 'Unit21',type: 'Residential unit',size: '900 sqft',bedrooms: 2,bathrooms: 1,parking: 'S01',storage: 'No',floor: 2 }),
        (condo21)-[:PART_OF]->(building),
        (travolta { name: 'John Travolta',type: 'Tenant', owner: true, unitsowned: 'Unit21' }),
```

```
(travolta)-[:LIVES_IN]->(condo21),
(condo22  { name: 'Unit22',type: 'Residential unit',size: '800 sqft',bedrooms: 1,bathrooms: 2,parking:
'S09',storage: 'No',floor: 2 }),
(condo22)-[:PART_OF]->(building),
(foster   { name: 'Jody Foster',type: 'Tenant', owner: false, rentsfrom: 'Eddard Stark' }),
(foster)-[:LIVES_IN]->(condo22),
(condo23  { name: 'Unit23',type: 'Residential unit',size: '1200 sqft',bedrooms: 2,bathrooms: 2,parking:
'S19',storage: 'US15',floor: 2 }),
(condo23)-[:PART_OF]->(building),
```

Sample code  shown below is a sample of code used to create the first data set in EFM:

```
CREATE  (nodetypebuilding { name: 'building' }),
(buildingstreet { name: 'Fifth Ave N' }),
(city { name: 'Saskatoon' }),
(province { name: 'Saskatchewan' }),
(country { name: 'Canada' }),
(continent { name: 'North America' }),
(condo21  { name: 'Unit21' }),
(parkings01  { name: 'S01' }),
(condo21)-[:PART_OF]->(building),
(condo21)-[:IS_A]->(nodetyperesunit),
(condo21)-[:SIZE_OF]->(unitsize900),
(condo21)-[:WITH_BEDROOM]->(number2),
```

Nodes and their properties are in blue while relationships are in green, making them immediately identifiable in code.

### 5.2.2 Admin web console

Many functions can be accomplished through Admin web console, such as create data, graphically search and represent it, and run Cypher queries (previous versions supported Gremlin, general graph traversal language, which was dropped as of Neo4J 2.0) manually or through scripts, etc.

Figure 5-1 shows the main page of the admin web console and some metadata about existing data stored in the database.

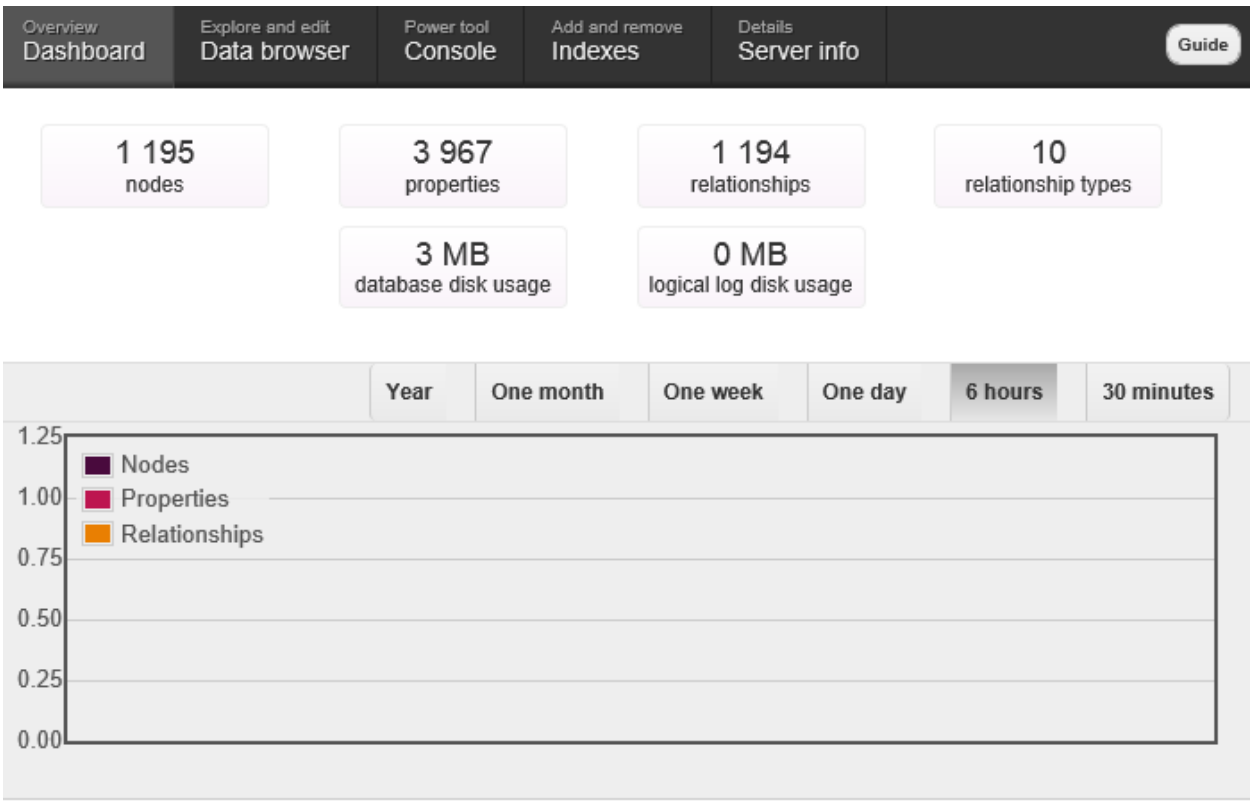


Figure 5-1. Admin web console main page

Here it is possible to see that 1195 nodes currently exist in the database with a total number of 3967 properties. As well, there are 1194 relationships in total, but only 10 of them unique. All of this data takes about 3 Mb on disk.

Figure 5-2 shows the graphical representation of sample data, selection of nodes and relationships between them.


Figure 5-2. Admin web console GUI representation of sample entities

This graphically represents the data entered in the database and looks a lot like the original white board modeling done at the beginning. Several tenants live in the same unit (225), while only one lives in the other (224). At the center of everything is the building which all of the units

are part of. Some other entities are present too, like maintenance requests, building manager, property manager, property management company, etc.

Figure 5-3 shows, in addition to nodes and relationships in general, the properties of a selected node or relationship, in this particular case the properties of the main node, building.

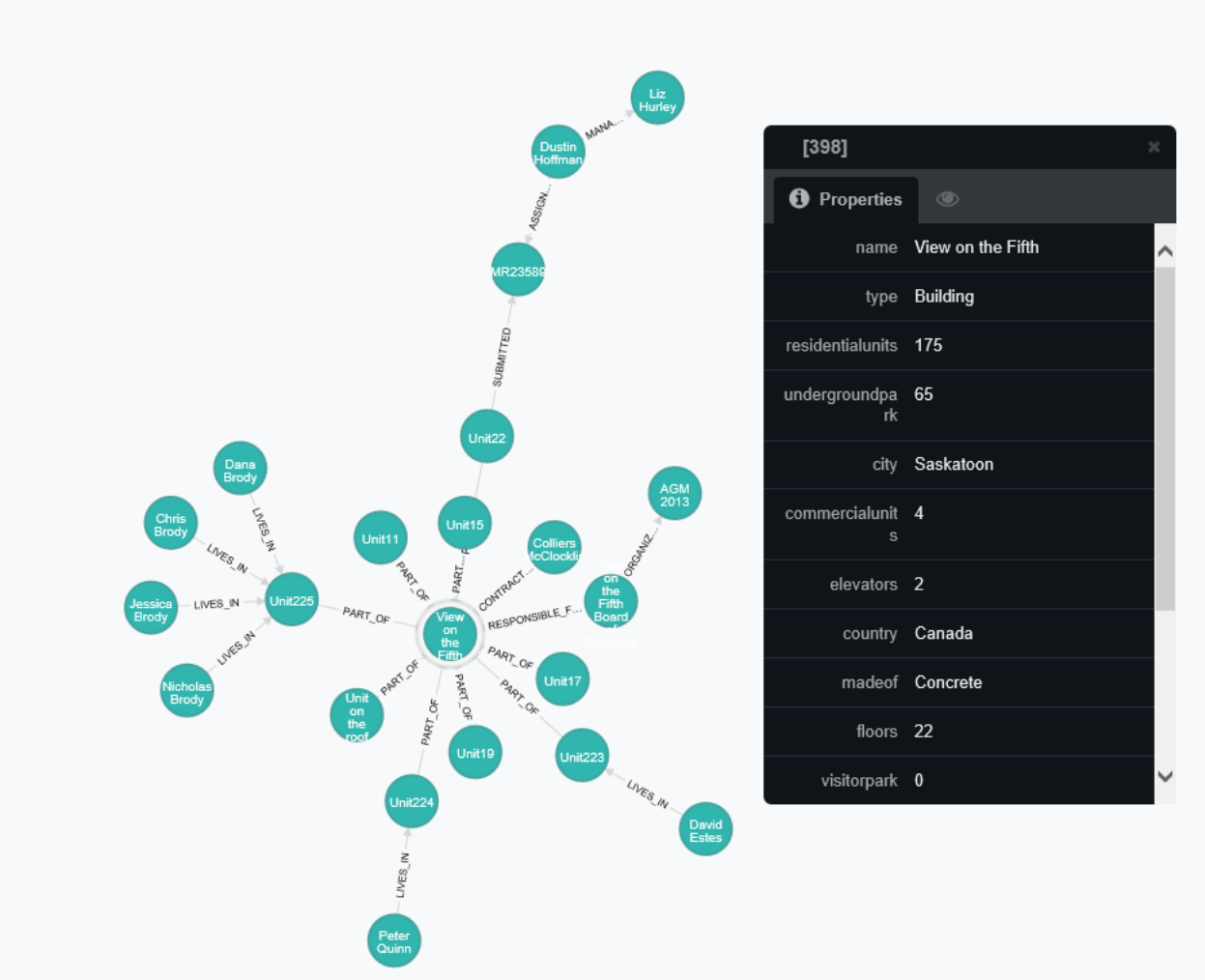


Figure 5-3. Admin web console GUI representation of sample entities and properties

Here, in the building properties, it is possible to see the name of the building, the number of units, parking, elevators, location, number of floors, etc.

In Figure 5-4 one full data set created in PFM is shown. Here too the building is in the center, but it is harder to distinguish amongst all the other data, although it is still recognizable.

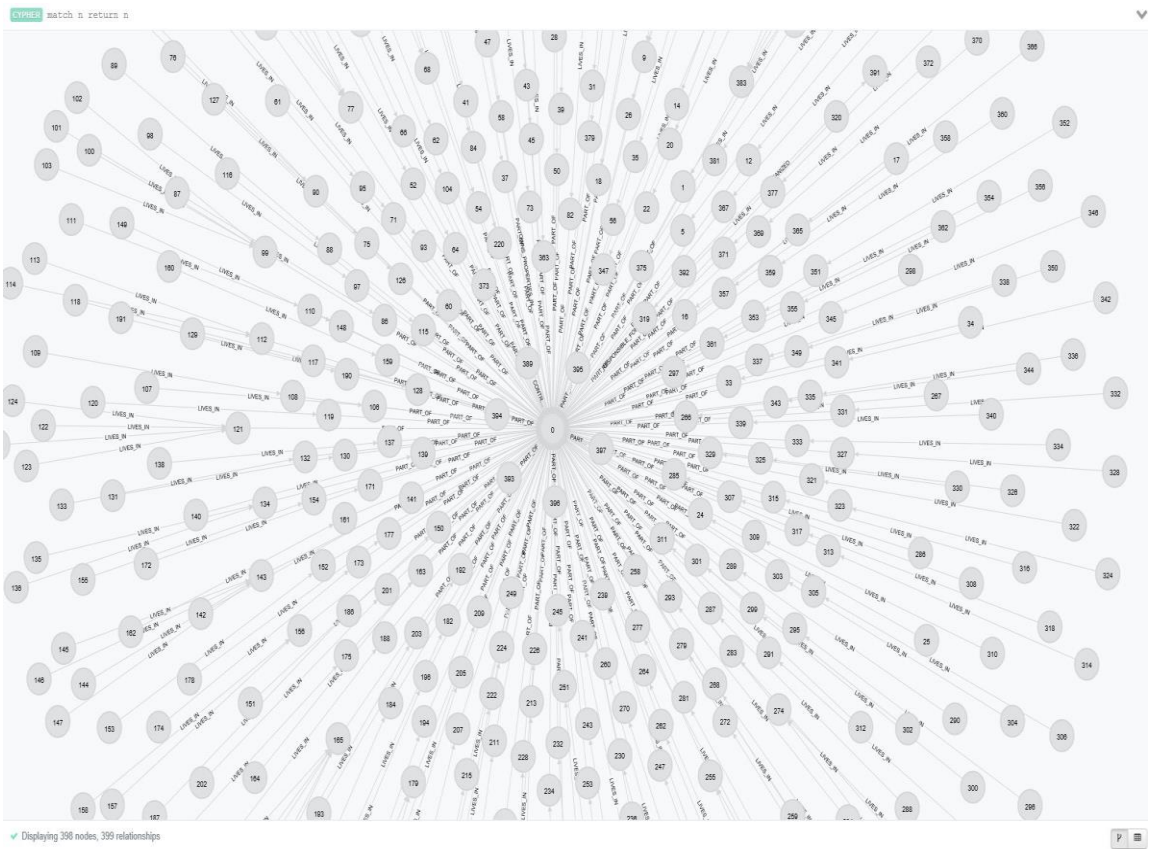


Figure 5-4. Admin web console GUI representation of one full data set in PFM

Figure 5-5 shows one full data set created in EFM. Building is in the center here as well, but it is impossible to distinguish because of all the other entities which are in much larger quantity, and are making the image much more complex. In addition to the main center, that is building, in EFM there are several other “regional” centers which represent reused nodes that are represented as properties in PFM.

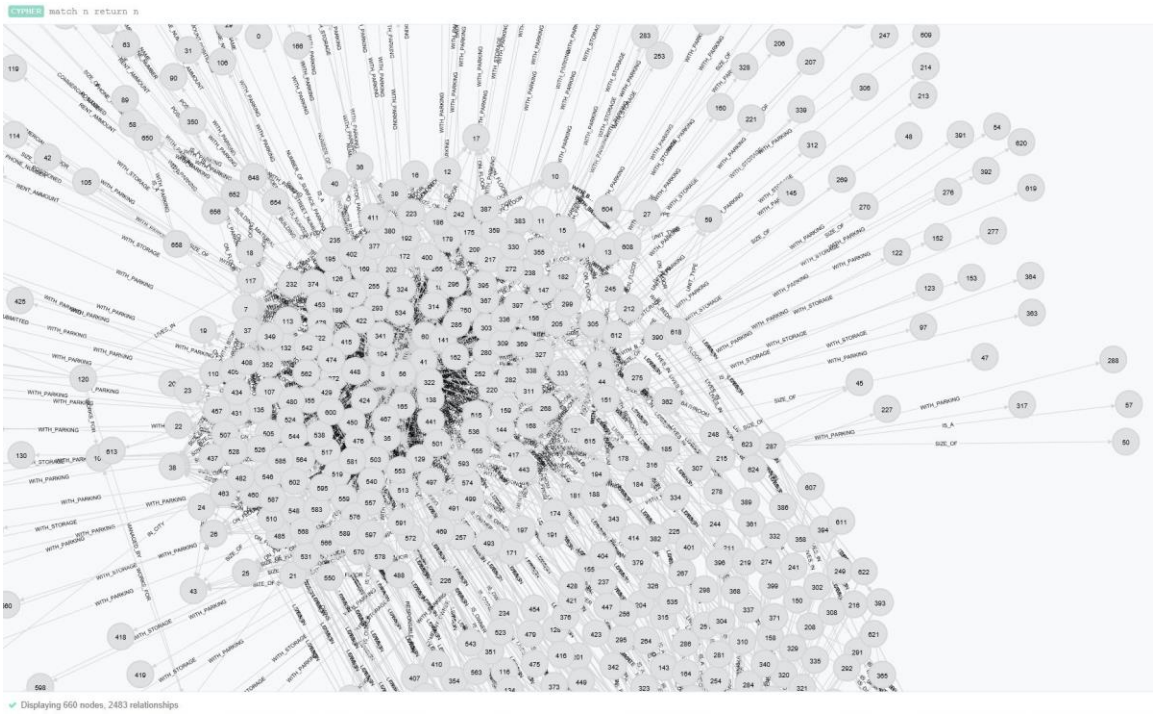


Figure 5-5. Admin web console GUI representation of one full data set in EFM

## 5.3 Indexes

Indexes were not used in either of the modeling approaches as their implementation could vary, and that could add a level of complexity to both models. That, in turn, could affect and skew the results as both models would be dependent on the level of index implementation. Indexes are interesting enough in graph databases and probably deserve some dedicated work.

## 5.4 Methodology

As previously mentioned, experiments were conducted on two identical virtual machines hosted on same physical machine, with only one booted at the time of executing individual

experiment tasks. But just as there are no two identical moments in history, there are no two identical moments in the life of virtual machines (or any other type of machine). Instead of running each query once, recording the result and using it for analysis, each query was run five times, the results recorded and the average value calculated. Only the average values were used in the performance analysis. The only exception to this is running queries in cold mode, because it is not possible to run the same query on a database twice and still consider it cold mode; therefore the results for cold mode queries are not averages, but single recorded values. The queries are based on the same questions and are the same in cold and warm mode, but are different for different models, as it was required to alter them, in some cases slightly and in some significantly, to reflect the differences in structure of the presented models. The questions represent a selection of different possible inquiries eventual users might have about the system. Execution times are recorded in milliseconds.

# CHAPTER 6
# RESULTS

## 6.1 Experiments

The experiments that were conducted were designed to match research goals presented in Chapter 2 and follow the methodology explained in Chapter 5, paragraph 5.4.

## 6.2 The Results of the Implementation of the First Research Goal

The first research goal was successfully achieved by developing and implementing both modeling approaches into separate databases for the chosen domain. All of the conceived elements were created in both models and there were no missing components. PFM was created first and it took 4 tries to perfect, including revisions based on the issues found while creating EFM model. EFM was created second and based on data from PFM, it took 17 tries to perfect it, including corrections made after some of the queries returned unexpected results. Finally after everything was implemented, both models fully supported all the same operations (create, add, remove ) and all the queries successfully completed execution and returned the same expected results as the identical data was stored in both models and databases.

## 6.3 The Results of the Implementation of the Second Research Goal

The second research goal was successfully achieved by capturing query performance against both models for the queries designed specifically for them. In the database world, performance of

the queries is measured solely based on the speed the correct data is returned. Queries for both models were designed based on questions we wanted answered and queries were always different between models, even though the questions were the same. The queries for different models were always different as a query from one model either could not be executed on the other model without modifications or would return incorrect data in some situations.

### 6.3.1 Performance data

Figure 6-1 represents performance data for queries run on one data set in cold mode. Cold mode means that the queries were executed for the first time on a freshly restarted database, where data had not had a chance to be placed in memory, while warm mode means a query was run at least once in the current database instance incarnation before the results were recorded. While there is data to support significant query performance difference between cold and warm mode, there is no evidence to suggest that there are differences for query performance between subsequent executions on a warm database. In other words, there is no pattern which suggests that the third execution is faster than the second, the fourth faster than the third and so on.

The data that was used to create Figure 6-1 is recorded data from query execution on a cold mode in PFM model and is presented in Table 6-1.

| Query | PFM Time (ms) | EFM Time (ms) |
|---|---|---|
| Q1 | 1577 | 897 |
| Q2 | 792 | 701 |
| Q3 | 415 | 980 |
| Q4 | 1011 | 722 |
| Q5 | 803 | 641 |
| Q6 | 745 | 307 |
| Q7 | 594 | 319 |
| Q8 | 660 | 271 |
| Q9 | 1262 | 888 |
| Q10 | 814 | 902 |
| Q11 | 751 | 912 |
| Q12 | 525 | 282 |
| Q13 | 715 | 678 |
| Q14 | 922 | 800 |
| Q15 | 424 | 415 |
| Q16 | 1455 | 595 |

Table 6-1. Query execution results for both models in cold mode

Queries are based on nineteen "questions" and are marked Q1to Q19. First sixteen questions are designed for a single data set, and remaining 3 are designed multiple data sets. First sixteen can be seen in the Figures 6-1 and 6-2 on horizontal axis, last three can be seen in the Figure 6-3. For example, query Q1 is based on the question "Who lives on and above the $15^{th}$ floor and on which floor?" and that is for the single building as it is assumed there is only one building. This query can run on multiple data sets database. It would not be very useful as it would return

results for all tenants that live above the 15th floor, in any building but without reporting which building it belongs to. The example for multiple data sets is query Q17, based on the question "Which buildings are in Canada, in what Cities and what are their names?" which assumes there are multiple buildings present in the database. This query can run on database with the single data set, but it would not be very useful as it would return only one result.

Q1 for PFM looks like this:

```
match (a)-[:LIVES_IN]->(b) where b.floor >= 15 return distinct a.name as Name, b.floor as Floor;
```

Q1 for EFM looks like this:

```
match (a)-[:LIVES_IN]->(b)-[:ON_FLOOR]->(c) where c.name >=15 return distinct a.name as Name,
c.name as Floor;
```

In this case the main difference is the second relationship (ON_FLOOR) in Q1 for the EFM, which was modeled as a relationship and a floor as a node, while in the PFM, the floor was set as a property of each unit that was represented as a node.

Q17 for PFM looks like this:

```
match (a)-[:PART_OF]->(b) where b.country='Canada' return  distinct b.name as Building ,
b.city as City, b.province as Province;
```

Q17 for EFM looks like this:

```
match (a)-[:IN_CITY]->(b)-[:IN_PROVINCE]->(c)-[:IN_COUNTRY]->(d) where
d.name='Canada' return  a.name as Building , b.name as City, c.name as Province;
```

In this case the main difference are the three relationships ( IN_CITY, IN_PROVINCE and IN_COUNTRY ) that connect the existing entity building and the new entities city, province and country which were all properties for the node building.

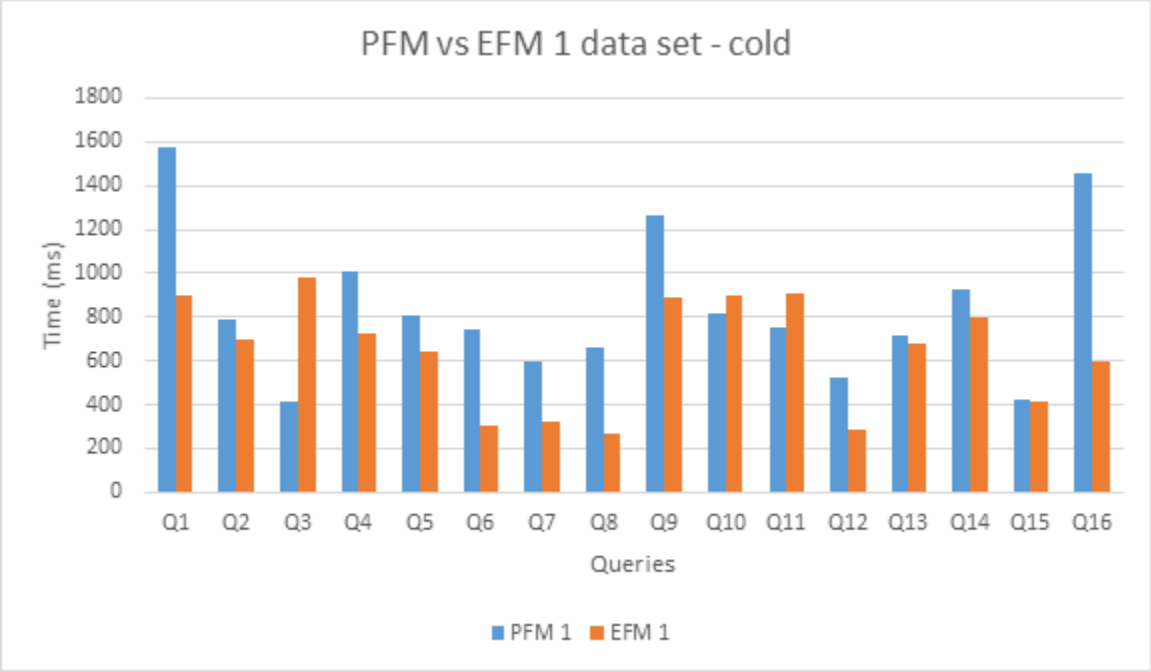


Figure 6-1. Performance comparison chart for one data set in cold mode

In cold mode (C1) queries against the EFM model completed execution faster in all but three cases (queries Q3, Q10 and Q11) with an average of 22.6% increase in speed. C1 here represents cold mode (C) with one (1) data set. Similarly, W1 and W5 represent warm mode on one and five data sets, respectively.

Questions that the queries were slower for are:

Q3 = How many people live on the 11th floor

Q10= Which units don't have parking assigned?

Q11= How many residential/commercial/total rental units?

And these deserve special attention in the analysis section.

Tables 6-2 and 6-3 contain data that was used to construct Figure 6-2. R1-R5 represent five runtimes or executions for each of the queries, while Avg represents the calculated average values all in milliseconds (ms).

| Q/R | R1 (ms) | R2 (ms) | R3 (ms) | R4 (ms) | R5 (ms) | Avg (ms) |
|---|---|---|---|---|---|---|
| Q1 | 874 | 344 | 460 | 345 | 465 | 498 |
| Q2 | 519 | 307 | 436 | 407 | 298 | 393 |
| Q3 | 397 | 352 | 328 | 419 | 359 | 371 |
| Q4 | 599 | 320 | 362 | 364 | 425 | 414 |
| Q5 | 525 | 453 | 421 | 415 | 378 | 438 |
| Q6 | 515 | 432 | 271 | 380 | 288 | 377 |
| Q7 | 506 | 403 | 299 | 289 | 443 | 388 |
| Q8 | 491 | 445 | 377 | 395 | 268 | 395 |
| Q9 | 427 | 537 | 483 | 733 | 428 | 521 |
| Q10 | 448 | 349 | 499 | 310 | 310 | 383 |
| Q11 | 255 | 278 | 431 | 412 | 425 | 360 |
| Q12 | 402 | 376 | 410 | 452 | 391 | 406 |
| Q13 | 416 | 317 | 445 | 624 | 466 | 456 |
| Q14 | 556 | 462 | 283 | 474 | 266 | 408 |
| Q15 | 340 | 436 | 422 | 485 | 419 | 420 |
| Q16 | 453 | 322 | 349 | 544 | 366 | 407 |

Table 6-2. PFM query performance data for one data set in warm mode

| Q/R | R1 (ms) | R2 (ms) | R3 (ms) | R4 (ms) | R5 (ms) | Avg (ms) |
|---|---|---|---|---|---|---|
| Q1 | 317 | 479 | 422 | 506 | 397 | 424 |
| Q2 | 223 | 404 | 291 | 307 | 399 | 324 |
| Q3 | 592 | 493 | 345 | 494 | 479 | 481 |
| Q4 | 252 | 414 | 469 | 408 | 332 | 375 |
| Q5 | 310 | 555 | 415 | 391 | 450 | 424 |
| Q6 | 244 | 288 | 372 | 335 | 416 | 331 |
| Q7 | 256 | 399 | 281 | 413 | 431 | 356 |
| Q8 | 284 | 389 | 297 | 402 | 398 | 354 |
| Q9 | 529 | 511 | 534 | 570 | 355 | 500 |
| Q10 | 314 | 497 | 510 | 523 | 373 | 443 |
| Q11 | 889 | 532 | 371 | 279 | 381 | 490 |
| Q12 | 217 | 251 | 288 | 175 | 169 | 220 |
| Q13 | 438 | 567 | 460 | 254 | 446 | 433 |
| Q14 | 595 | 355 | 299 | 349 | 320 | 384 |
| Q15 | 271 | 296 | 264 | 286 | 267 | 276 |
| Q16 | 332 | 305 | 269 | 255 | 246 | 281 |

Table 6-3. EFM query performance data for one data set in warm mode

The Figure 6-2 represents performance data for queries run on one data set in warm mode (W1). Data recorded here was when each of the queries were already run several times against the database and data was put in the cache. Just as for the cold mode, the queries against the EFM model completed execution faster in all but the same three cases. The average increase in speed, however, was only 8.0 %.

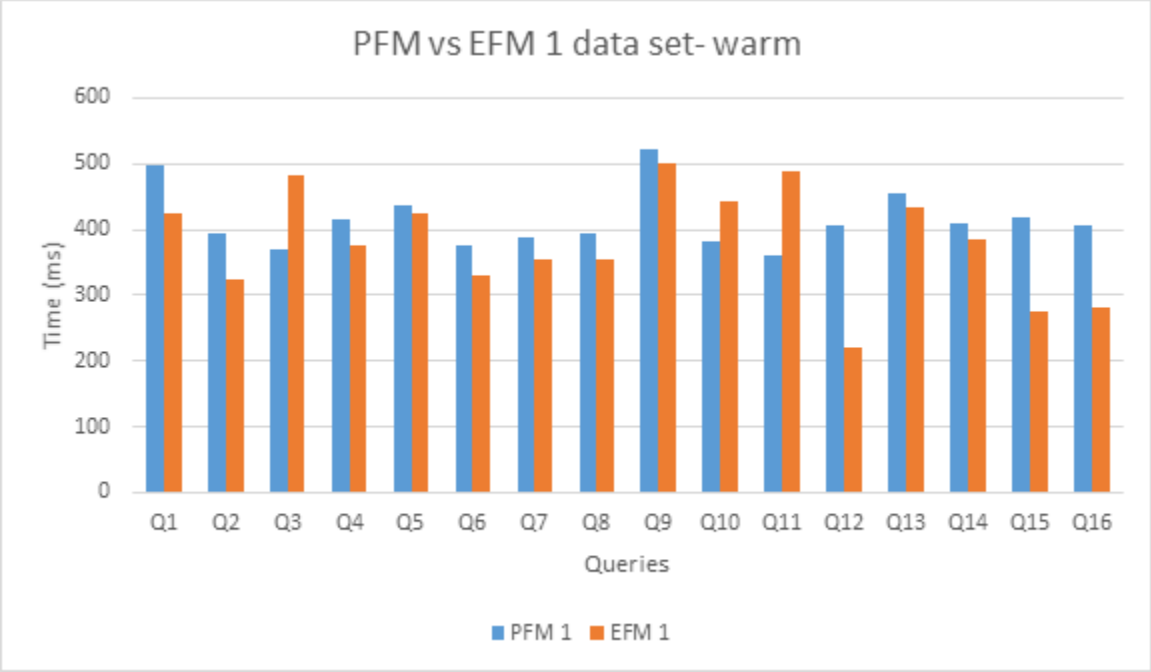


Figure 6-2. Performance comparison chart for one data set in warm mode

### 6.3.2 Performance analysis

Overall EFM queries performed faster than PFM queries, in some case slightly, in some other significantly and in few cases drastically. On a few occasions, several queries finished under 150 ms, with fastest being at 129 ms. All of the ten fastest queries from, all executions and both models, are from EFM. All of the numbers above are actually average numbers and each query response time is calculated as average from several executions. This is the case for both models in both cold and warm modes and it is pretty consistent. However, some of the queries were faster in PFM than EFM. The two cases were the calculation of how many of something and in one case looked for a certain type of units, which also implicitly includes the calculation of how many of those units there are. It would appear that PFM is faster if queries include calculation to determine counts. In other cases EFM performed faster. But that is not entirely true as there are two other cases (Q2 and Q9) where calculation is required and EFM performed better (in case of

Q2 the performance is very close). The best that can be said is that for the queries that involve calculation performance is comparable, while for all other cases EFM performs faster, at least for the cases examined in these experiments. That is partially expected as graph database developers claim that nodes and relationships are first class citizens and the generation of them, under certain circumstances, should improve database performance.

But that improvement comes at a significant price, which will be explained in more detail in following sections.

### 6.4 The Results of the Implementation of the Third Research Goal

The third research goal was successfully achieved by expanding both models with additional data sets and re-running performance checking queries. Both models were expanded with the same data, adjusted to reflect each of the models' structures. All the queries still completed successfully and returned the same data. The queries also needed to be modified to reflect the existence of multiple data sets (buildings in this case). Figure 6-3 shows query performance comparison between PFM and EFM for expanded databases with multiple data sets. Typically, the match clause statements (that is used for matching patterns in Neo4J, i.e. node1-relationship1-node2) had to be expanded from PFM to EFM and the where clause (used to narrow results from match clause) had to be extended to include the existence of additional data sets.

The queries selected for execution on multiple data sets were those that were either close (1, 5, 8) in single data sets or were significantly faster than their counterparts (3, 16). Three additional queries could only be executed on the databases that included multiple data sets as

they queried the data in different locations while the initial queries did not take location into account as there was only one location.

Table 6-4 contains the data that was recorded at execution time for PFM and used to construct Figure 6-3 representing performance for multiple data sets in warm mode.

| Q/R | R1 (ms) | R2 (ms) | R3 (ms) | R4 (ms) | R5 (ms) | Avg (ms) |
|---|---|---|---|---|---|---|
| Q1 | 1177 | 1222 | 1139 | 891 | 660 | 1018 |
| Q3 | 352 | 202 | 304 | 400 | 384 | 328 |
| Q5 | 318 | 382 | 387 | 403 | 413 | 380 |
| Q8 | 318 | 585 | 420 | 470 | 578 | 474 |
| Q16 | 744 | 489 | 476 | 554 | 457 | 544 |
| Q17 | 943 | 741 | 841 | 638 | 448 | 722 |
| Q18 | 542 | 688 | 593 | 587 | 324 | 547 |
| Q19 | 443 | 384 | 376 | 426 | 379 | 402 |

Table 6-4. PFM query performance data for multiple data sets in warm mode

Table 6-5 contains data that was recorded at the execution time for EFM and used to construct Figure 6-3 representing performance for multiple data sets in warm mode.

| Q/R | R1 (ms) | R2 (ms) | R3 (ms) | R4 (ms) | R5 (ms) | Avg (ms) |
|---|---|---|---|---|---|---|
| Q1 | 764 | 650 | 572 | 723 | 632 | 668 |
| Q3 | 800 | 656 | 571 | 609 | 621 | 651 |
| Q5 | 567 | 591 | 616 | 512 | 641 | 585 |
| Q8 | 501 | 467 | 497 | 474 | 494 | 487 |
| Q16 | 408 | 566 | 529 | 581 | 432 | 503 |
| Q17 | 433 | 373 | 351 | 391 | 496 | 409 |
| Q18 | 513 | 338 | 431 | 272 | 294 | 370 |
| Q19 | 463 | 416 | 341 | 292 | 297 | 362 |

Table 6-5. EFM query performance data for multiple data sets in warm mode

Figure 6-3 represents performance data for queries run on five data sets in warm mode (W5). The performance gain for the EFM model was preserved at 8.5%, mainly because of the three new queries (on aggregate data); without that, there is actually a loss of performance of 5.7%. Queries that performed better, in addition to Q3, are Q5 and Q8.

Q5= Find all the Cylons that live above 9th floor and which condo they live in

Q8= What is the total rent that the building receives from commercial tenants

In addition to this, performance generally and almost equally drops from W1 to W5 for both models, 33.1% for PFM and 32.3% for EFM.

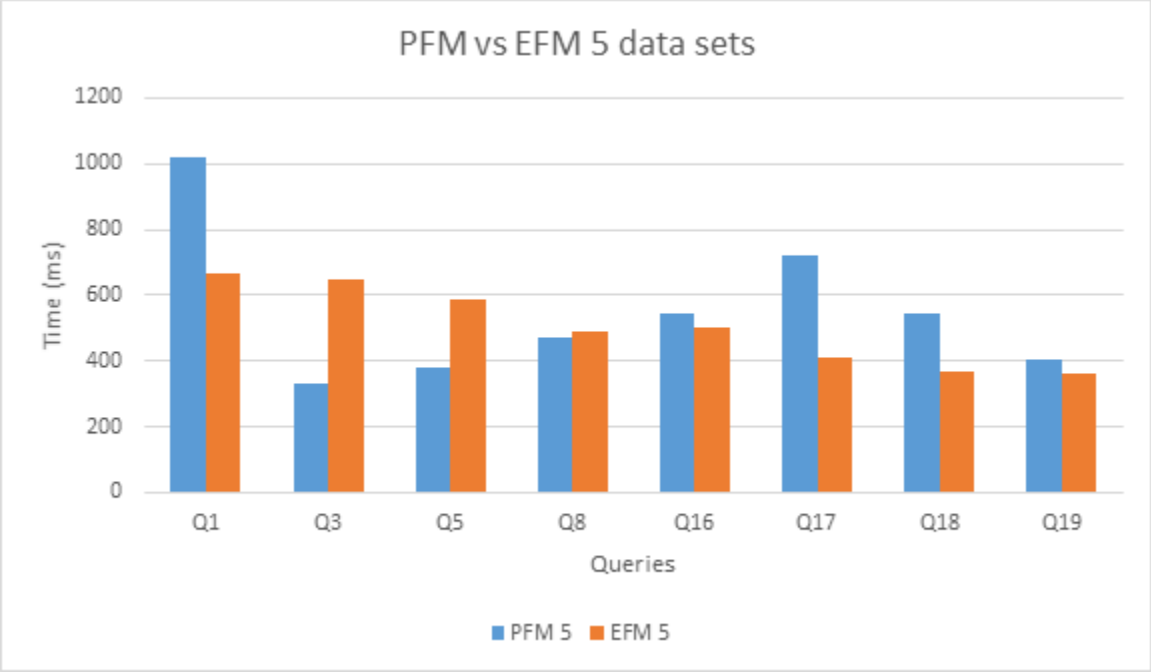


Figure 6-3. Performance comparison chart for five data sets in warm mode

### 6.5 The Results of the Implementation of the Fourth Research Goal

The fourth research goal was successfully achieved by the overall comparison of the models, including size on disk, reusability, query complexity, maintenance cost, etc.

EFM appears to allow for faster query performance (with some exceptions) but it comes with a price of significantly higher cost for maintenance which is reflected in more complex statements to create, alter, delete data and generally maintain data sets. For instance, if nodes are reused in EFM, they have to be maintained and each new relationship (incoming or outgoing) must be carefully implemented so as to not create new nodes, which would defeat the purpose of reusing nodes, nor would create additional nodes for the same purpose, which would cause incorrect data to be retrieved, in addition to data duplication. Doing this complicates code maintenance and requires more senior staff which increases the cost of maintenance, not just in

time, but in money directly as well. Maintaining the code data in PFM is relatively easy; instead of reusing a node, a new property needs to be created. It only applies for that node, does not affect anything else and always returns correct data.

Queries to retrieve data in EFM are more complex and harder to maintain if anything changes than for PFM. PFM data sets are pretty independent and easily maintained, even in the case of a database structural changes.

While EFM allows for data reusability (which PFM only does on very rare occasions) applying it adds to complexity.

EFM takes less space on disk if data is reused, but that significantly adds to complexity and maintenance cost. In case in questions disk space saved was approx. 11%, but that was without true and full data reusability, which if implemented can save even more space, but would be quite difficult to apply initially and to maintain. Disk space savings are almost exclusively achieved with node reusability, as relationships still had to be established and maintained in each case and could not be reused.

| Data set | PFM (ms) | EFM (ms) |
|---|---|---|
| 1 | 7038 | 9660 |
| 2 | 6490 | 9323 |
| 3 | 3749 | 8124 |
| 4 | 3596 | 8812 |
| 5 | 3735 | 7719 |

Table 6-6. Data set creation performance data for both models in warm mode

Data creation is faster in PFM and Table 6-6 represents the time needed to create the full data set in each model in one large create statement.

The creation of individual entities (nodes and relationships) in both models is very fast and almost instantaneous and comparing those times would not bring any value, especially since data is represented differently. What is represented with the two nodes and the one relationship in the PFM would typically be represented with the multiple nodes and relationships in the EFM.

Figure 6-4 is a graphical representation of Table 6-6 and shows the average of 43.6% performance degradation for the data set creation in EFM.

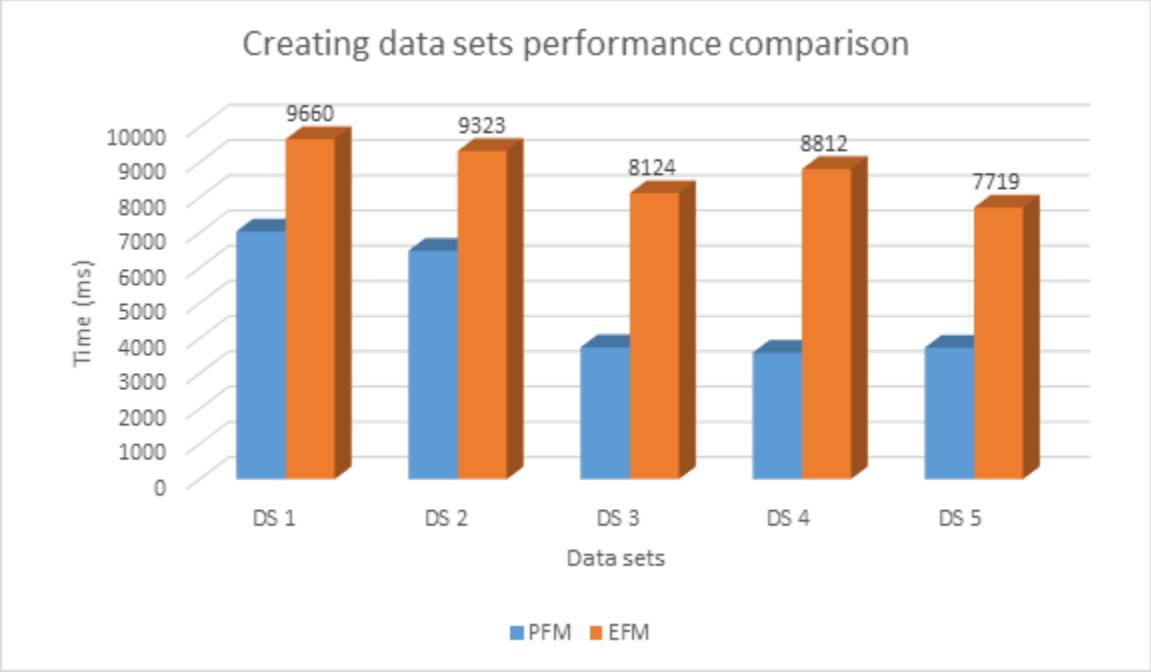


Figure 6-4. Performance comparison for creating data sets in warm mode

While PFM in this domain created only one hot node (building), the creation of hot nodes happens regularly in EFM. Although it is hard to prove and claim with absolute certainty that it affects performance, that is likely because it is difficult to achieve high query concurrency.

On the other hand, PFM creates a significant amounts of duplicated data, as it is not possible to reuse properties which have to be set for each entity. Data is duplicated to a certain extent in EFM as well. Creation of hot nodes and node reusability eliminates data duplication for node creation, but not for relationships as each relationship between two nodes has to be created, even if one or both nodes are reused.

# CHAPTER 7
# CONCLUSION AND FUTURE WORK

## 7.1 Conclusion

The main goal and focus of this thesis is to establish two valid distinct and opposed approaches for modeling data in graph databases. That goal was accomplished with the successful modeling and full implementation of the same data from the same domain for the same particular case in both modeling approaches: PropertyFirst and EntityFirst models. There was no omitted data or information that would be a deterrent to using the full model. In other words, the data did not have to be complete, but it did have to be identical in both models, just represented differently.

The second important goal is the comparison of the two modeling approaches and assurance that both models can support the same kind of operations against them. That goal was accomplished because queries could be executed for both models and returned the same data (that could be in a different format, but contain the same information). The queries were different for each model, tuned to each particular database and structure, but returned the identical data. They were typically more complex for EFM

Part of the second goal is also to compare the speed at which data was returned data when said queries were executed against the databases developed for both models, while the data that was returned remained the same. That goal was accomplished because speed in milliseconds for each query (created to satisfy certain questions) was captured and recorded for query runtimes, with the same data being returned. Then those values were compared and analyzed between the models.

The third goal is to expand the data with new data sets (new buildings in this case) and it was successfully achieved by adding and creating four new data sets, while still not losing any data between databases that support the different models and still supports all the same kind of operations and return the same data. Queries in the expanded model were expanded to include location data, for instance.

The fourth goal is to compare the overall differences in database models and it was successfully achieved by comparing size on disk that both databases consumed, creation of hot nodes (nodes with the most number of relationships) in EFM, notation of differences in most used relationships, analysis of differences of data duplication due to the modeling approach used and analysis of query complexity for the two models. This goal was accomplished by measuring the above mentioned components and comparing the differences between the opposing databases.

In conclusion, both models have certain advantages as well as disadvantages. PFM is logically easier to conceive and to model, simpler and easier to maintain. EFM is faster in most cases, uses less disk space and allows for reduced data duplication. Both of them could be used for certain applications or parts of applications, allowing for both models to be used for different purposes in same application. It is highly unlikely than either of them would be used in their pure form as presented in this research.

## 7.2 Future Work

This research has just started the work on the formalization of modeling for graph databases and hopefully has opened the door for future development. There are several possible tracks for next steps and they include, but are not limited to:

- Formalization of a modeling process similar to the normalization process for relational databases which could include definitions of when to use nodes and relationships and when to use properties, for either of these.
- Similar to this, formalize the process to determine what is the best and most appropriate way to reuse nodes and to determine what effect the creation of hot nodes could have on database performance.
- Include indexes for both modeling approaches in both standardized and specialized ways, depending on the model, and discuss any differences index inclusion has had on performance, both within same model and between different models.
- Introduce PropertyFirst and EntityFirst model development for classical Online Transactional Processing systems (OLTP) vs. Online Analytical Processing (OLAP) applications and the effect these could have on them with graph databases.
- Possibly convert data or part a of data set from one model to another and study the effect that could have with troubleshooting performance issues.

LIST OF REFERENCES

[1] Angeles, R. Gutierrez, C. Survey of Graph Database Models, (2005), Technical Report TR/DCC-2005-10 - Computer Science Department - Universidad de Chile

[2] Robinson, I., Webber, J. and Eifrem, E. Graph Databases, (June 2013), O'Reilly Media, 1st Edition, ISBN: 9781449356262

[3] Codd E.F. A Relational Model of Data for Large Shared Data Banks, (1970), Communications of the ACM, 13, 6, June 1970, 377-387

[4] Cypher Query Language http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html March 2015

[5] Neo4J database http://www.neo4j.org/

[6] Neo Technology http://neo4j.com/company/

[7] NoSQL databases list http://nosql-database.org , March 2015

[8] Graph Database http://neo4j.com/developer/graph-database , March 2015

[9] Neo4J database http://neo4j.com/developer/get-started/ , March 2015

[10] Vukotic, A. Waytt, N. Partner, J. Neo4J in Action, Dec 2014, Manning Publications, 1st Edition, ISBN: 9781617290763

[11] Gray, J. "The Transaction Concept: Virtues and Limitations, (1981), 7th International Conference on Very Large Data Bases, Volume 7, 144 - 154

[12] Haerder, T. Reuter, A. Principles of transaction-oriented database recovery, (1983), ACM Computing Surveys, 15, 4, Dec 1983, 287-317

[13] Vogels, W. Eventually consistent, (2009), Communications of the ACM - Rural engineering development, 52, 1, January 2009, 40-44

[14] Trudeau, R.J. Introduction to Graph Theory (1993), Dover Publications, 2nd Edition, ISBN: 0486678709

[15] Chartrand, G. Introductory Graph Theory (1985), Dover Publications, 1st Edition, ISBN: 0486247759

[16] Oracle Corporation http://www.oracle.com

[17] Microsoft SQL Server www.microsoft.com/en-us/server-cloud/products/sql-server/

[18] IBM DB2 www.**ibm**.com/software/data/**db2**/

[19] MySQL http://www.mysql.com/

[20] PostgreSQL http://www.postgresql.org/

[21] Sybase http://www.sybase.com/

[22] Date, C.J. An Introduction to Database Systems (2000), Addison Wesley Longman Inc, 7th Edition, ISBN: 0201385902

[23] Ullman, J.D Principles of Database and Knowledge-Base Systems (1988, 1989), Computer Science Press

[24] Silberschatz, A., Korth, H. F., and Sudarshan, S. Data Models. (1996), ACM Computing Surveys 28, 1, 105–108.

[25] Gutierrez, A., Pucheral, P., Steffen, H., and Thevenin, J.-M. Database Graph Views: A Practical Model to Manage Persistent Graphs. (1994), 20th International Conference on Very Large Data Bases (VLDB). Morgan Kaufmann, 391–402.

[26] Hidders, J. Typing Graph-Manipulation Operations (2002), 9th International Conference on Database Theory (ICDT). Springer-Verlag, 394–409

[27] Graves, M., Bergeman, E. R., and Lawrence, C. B. A Graph-Theoretic Data Model for Genome Mapping Databases. (1995), 28th Hawaii International Conference on System Sciences (HICSS). IEEE Computer Society, 32

[28] Andries, M., Gemis, M., Paredaens, J., Thyssens, I., and den Bussche, J. V. Concepts for Graph-Oriented Object Manipulation. (1992), 3rd International Conference on Extending Database Technology (EDBT). LNCS, vol. 580. Springer, 21–38

[29] Hidders, J. and Paredaens, J. GOAL, A Graph-Based Object and Association Language (1993), Advances in Database Systems: Implementations and Applications, CISM, 247–265

[30] Gyssens, M., Paredaens, J., den Bussche, J. V., and Gucht, D. V. A Graph-Oriented Object Database Model.( 1990), 9th Symposium on Principles of Database Systems (PODS). ACM Press, 417–424

[31] Gyssens, M., Paredaens, J., and Gucht, D. V. 1990. A Graph-Oriented Object Model for Database End-User Interfaces, ACM SIGMOD International Conference on Management of data. ACM Press, 24–33

[32] Gyssens, M., Paredaens, J., den Bussche, J. V., and Gucht, D. V. A Graph-Oriented Object Database Model. (1991), 9th Symposium on Principles of Database Systems (PODS). ACM Press, 417–424

[33] Sheng, L., Ozsoyoglu, Z. M., and Ozsoyoglu, G. A Graph Query Language and Its Query Processing.(1999), 15th International Conference on Data Engineering (ICDE). IEEE Computer Society, 572–581

[34] Amann, B. and Scholl, M. Gram: A Graph Data Model and Query Language.(1992.), European Conference on Hypertext Technology (ECHT). ACM, 201–211

[35] Kiesel, N., Schurr, A., and Westfechtel, B. GRAS: A Graph-Oriented Software Engineering Database System.(1996), IPSEN Book. 397–425.

[36] Guting, R. H. GraphDB: Modeling and Querying Graphs in Databases. (1994), 20th International Conference on Very Large Data Bases (VLDB). Morgan Kaufmann, 297–308

[37] Levene, M. and Poulovassilis, A.. An Object-Oriented Data Model Formalised Through Hypergraphs.(1991), Data & Knowledge Engineering (DKE) 6, 3, 205–224

[38] Kunii, H. S. DBMS with Graph Data Model for Knowledge Handling. (1987), Fall Joint Computer Conference on Exploring technology: today and tomorrow. IEEE Computer Society Press, 138–142

[39] Paredaens, J., Peelman, P., and Tanca, L. G-Log: A Graph-Based Query Language.(1995), IEEE Transactions on Knowledge and Data Engineering (TKDE) 7, 3, 436–453

[40] Levene, M. and Poulovassilis, A. The Hypernode Model and its Associated Query Language.(1990), 5th Jerusalem Conference on Information technology. IEEE Computer Society Press, 520–530

[41] Poulovassilis, A. and Levene, M. A Nested-Graph Model for the Representation and Manipulation of Complex Objects. (1994), ACM Transactions on Information Systems (TOIS) 12, 1, 35–68

[42] Levene, M. and Loizou, G. A Graph-Based Data Model and its Ramifications.(1995), IEEE Transactions on Knowledge and Data Engineering (TKDE) 7, 5, 809–823

[43] Consens, M. and Mendelzon, A. Hy+: a Hygraph-based query and visualization system. (1993), SIGMOD Record 22, 2, 511–516

[44] Kuper, G. M. and Vardi, M. Y.. A New Approach to Database Logic.(1984), 3rd Symposium on Principles of Database Systems (PODS). ACM Press, 86–96

[45] Kuper, G. M. and Vardi, M. Y. The Logical Data Model.(1993), ACM Transactions on Database Systems (TODS) 18, 3, 379–413

[46] Lecluse, C., Richard, P., and Velez, F. O2, an Object-Oriented Data Model.(1988), ACM SIGMOD International Conference on Management of Data. ACM Press, 424–433

[47] Gemis, M. and Paredaens, J. An Object-Oriented Pattern Matching Language.(1993), 1st JSSST International Symposium on Object Technologies for Advanced Software. Springer-Verlag, 339–355

[48] Roussopoulos, N. and Mylopoulos, J. Using Semantic Networks for Database Management. (1975), International Conference on Very Large Data Bases (VLDB). ACM, 144–172

[49] Mainguenaud, M. Simatic XT: A Data Model to Deal with Multi-scaled Networks.(1992), Computer, Environment and Urban Systems 16, 281–288

[50] Tompa, F. W. A Data Model for Flexible Hypertext Database Systems. (1989), ACM Transactions on Information Systems (TOIS) 7, 1, 85–100

[51] Watters, C. and Shepherd, M. A. A Transient Hypergraph-Based Model for Data Access. (1990), ACM Transactions on Information Systems (TOIS) 8, 2, 77–102

[52] Codd, E.F. Extending the database relational model to capture more meaning (1979), ACM Transactions on Database Systems. 4, 4, 397–434

[53] Codd, E.F. 1971. Further Normalization of the Data Base relational Model (1971), IBM Research Report RJ909

[54] Codd, E.F. Normalized Data Base Structure: A Brief Tutorial. (1971), ACM SIGFIDET Workshop on Data Description, Access and Control, 1-17

[55] Codd, E.F. Recent Investigations in Relational Database Systems. (1974), IFIP Congress, 1017-1021

[56] Pritchett, D. BASE: An ACID Alternative (2008), ACM Queue - Object-Relational Mapping, 6, 3, May/June 2008, 48-55

APPENDIX
QUESTIONS AND QUERIES

## A.1 Questions

Question 1. Who lives on and above 15th floor and on which floor?

Question 2. How many people lives above 15th floor?

Question 3. Who lives on 11th floor?

Question 4. How many people live on 11th floor?

Question 5. Find Cylons that live above 9th floor and which condo they live in?

Question 6. Who attended AGM in 2013?

Question 7. How many people attended AGM in 2013?

Question 8. What is total rent building receives from commercial tenants?

Question 9. How many persons that are living in same building with parents?

Question 10. Which units don't have parking assigned?

Question 11. How many residential rental units?

Question 12. How many commercial rental units?

Question 13. How many total rental units?

Question 14. Who are owners?

Question 15. Who are renters?

Question 16. How many tenants?

Question 17. For buildings in Canada, what cities are they in and what are their names?

Question 18. How many buildings in Canada?

Question 19. For buildings in USA, what cities are they in and what are their names?

## A.2 Queries for Single Data Set in PFM

```
Q1. match (a)-[:LIVES_IN]->(b)
      where b.floor >= 15
      return  a.name as Name, b.floor as Floor;

Q2. match (a)-[:LIVES_IN]->(b)
      where b.floor > 15
      return count(a.name) ;

Q3. match (a)-[:LIVES_IN]->(b)
     where b.floor = 11
     return a.name as Name, b.floor as Floor;

Q4. match (a)-[:LIVES_IN]->(b)
      where b.floor = 11
      return count(a.name);

Q5. match (a)-[:LIVES_IN]->(b)
      where b.floor > 9 and a.Cylon=true
      return a.name as Name, b.name as Condo;

Q6. match (a)-[:LIVES_IN]->(b)
      where a.agm2013='attended'
      return a.name;

Q7. match (a)-[:LIVES_IN]->(b)
      where a.agm2013='attended'
      return count(a.name);

Q8. match (a)-[:PART_OF]->(b)
      where a.type='Commercial unit'
      return sum(a.rent);

Q9. match (a)-[:LIVES_IN]->(b)
     where a.sons is not null or a.daughters is not null
     return count(a.sons) + count(a.daughters);

Q10. match (a)-[:PART_OF]->(b)
       where a.parking='No'
       return distinct a.name;

Q11. match (a)-[:PART_OF]->(b)
       where a.type='Residential unit'
       return count(a.name);
```

Q12. match (a)-[:PART_OF]->(b)
where a.type='Commercial unit'
return count(a.name);

Q13. match (a)-[:PART_OF]->(b)
where a.type='Commercial unit' or a.type='Residential unit'
return count(a.name);

Q14. match (a)-[LIVES_IN]->(b)
where a.owner=true
return a.name;

Q15. match (a)-[LIVES_IN]->(b)
where a.rentsfrom is not null
return a.name;

Q16. match (a)-[LIVES_IN]->(b)
where a.type='Tenant'
return count(a.name);

### A.3 Queries for Single Data Set in EFM

```
Q1. match (a)-[:LIVES_IN]->(b)-[:ON_FLOOR]->(c)
      where c.name >=15
      return a.name as Name, c.name as Floor;

Q2. match (a)-[:LIVES_IN]->(b)-[:ON_FLOOR]->(c)
      where c.name >15
      return count(a.name) as Name;

Q3. match (a)-[LIVES_IN]->(b)-[:ON_FLOOR]->(c)
      where c.name = 11
      return a.name as Name, c.name as Floor;

Q4. match (a)-[LIVES_IN]->(b)-[:ON_FLOOR]->(c)
      where c.name = 11
      return count(a.name);

Q5. match (a)-[:LIVES_IN]->(b)-[:ON_FLOOR]->(c), (a)-[:IS_CYLON]->(d)
      where c.name >9 and d.name='Yes'
      return a.name as Name, b.name as Condo;

Q6. match (a)-[:ATTENDED]->(b)
      return a.name;

Q7. match (a)-[:ATTENDED]->(b)
      return count(a.name);

Q8. match (a)-[:PART_OF]->(b), (a)-[:IS_A]->(c), (a)-[:RENT_AMMOUNT]->(d)
      where c.name='Commercial unit'
      return sum(d.name);

Q9. match (a)-[:LIVES_IN]->(d), (b)-[:LIVES_IN]->(d), (a)-[:IS_SON_OF]->(f),
     (b)-[:IS_DAUGHTER_OF]->(f)
     return count(a.name) + count(b.name);

Q10. match (a)-[:PART_OF]->(b), (a)-[:WITH_PARKING]->(c)
       where c.name='No'
       return a.name;

Q11. match (a)-[:PART_OF]->(b), (a)-[:IS_A]->(c)
       where c.name='Residential unit'
       return count(a.name);
```

Q12. match (a)-[:PART_OF]->(b), (a)-[:IS_A]->(c)
where c.name='Commercial unit'
return count(a.name);

Q13. match (a)-[:PART_OF]->(b), (a)-[:IS_A]->(c)
where c.name='Commercial unit' or c.name='Residential unit'
return count(a.name);

Q14. match (a)-[:LIVES_IN]->(b), (a)-[:IS_OWNER]->(c)
where c.name='Yes'
return count(a.name);

Q15. match (a)-[:LIVES_IN]->(b), (a)-[:IS_OWNER]->(c)
where c.name='No'
return count(a.name);

Q16. match (a)-[:IS_A]->(b)
where b.name='Tenant'
return count(a.name);

## A.4 Queries for Multiple Data Sets in PFM

```
Q1. match (a)-[:LIVES_IN]->(b)-[:PART_OF]->(c)
      where b.floor >= 15 and c.name='View on the Fifth'
      return  a.name as Name, b.floor as Floor, c. name as Building;

Q3. match (a)-[:LIVES_IN]->(b)-[:PART_OF]->(c)
      where b.floor = 11 and c.name='View on the Fifth'
      return count (a.name);

Q5. match (a)-[:LIVES_IN]->(b)-[:PART_OF]->(c)
      where b.floor > 9 and a.Cylon=true and c.name='View on the Fifth'
      return a.name as Name, b.name as Condo;

Q8. match (a)-[:PART_OF]->(b)
     where a.type='Commercial unit' and b.name='View on the Fifth'
     return sum(a.rent);

Q16. match (a)-[LIVES_IN]->(b)-[:PART_OF]->(c)
       where a.type='Tenant' and c.name='View on the Fifth'
       return count(a.name);

Q17. match (a)-[:PART_OF]->(b)
       where b.country='Canada'
       return  b.name as Building , b.city as City, b.province as Province;

Q18. match (a)-[:PART_OF]->(b)
      where b.country='Canada'
      return  count(distinct b.name);

Q19. match (a)-[:PART_OF]->(b)
       where b.country='USA'
       return b.name as Building , b.city as City, b.state as State;
```

## A.5 Queries for Multiple Data Sets in EFM

```
Q1. match (a)-[:LIVES_IN]->(b)-[:ON_FLOOR]-(c), (b)-[:PART_OF]->(d)
     where c.name >=15  and d.name='View of the Fifth'
     return a.name as Name, c.name as Floor;

Q3. match (a)-[LIVES_IN]->(b)-[:ON_FLOOR]->(c), (b)-[:PART_OF]->(d)
      where c.name = 11 and d.name='View of the Fifth'
      return count (distinct a.name);

Q5. match (a)-[:LIVES_IN]->(b)-[:ON_FLOOR]->(c), (a)-[:IS_CYLON]->(d),
      (b)-[:PART_OF]->(e)
      where c.name >9 and d.name='Yes'  and e.name='View of the Fifth '
      return a.name as Name, b.name as Condo, e.name as Building;

Q8. match (a)-[:PART_OF]->(b), (a)-[:IS_A]->(c), (a)-[:RENT_AMMOUNT]->(d)
      where b.name='View of the Fifth' and c.name='Commercial unit'
      return sum(d.name);

Q16. match (a)-[:IS_A]->(d), (a)-[:LIVES_IN]->(b)-[:PART_OF]->(c)
       where d.name='Tenant' and c.name= 'View of the Fifth'
       return count(a.name);

Q17. match (a)-[:IN_CITY]->(b)-[:IN_PROVINCE]->(c)-[:IN_COUNTRY]->(d)
       where d.name='Canada'
       return  a.name as Building , b.name as City, c.name as Province;

Q18. match (a)-[:IN_CITY]->(b)-[:IN_PROVINCE]->(c)-[:IN_COUNTRY]->(d)
       where d.name='Canada'
       return  count(a.name);

Q19. match (a)-[:IN_CITY]->(b)-[:IN_STATE]->(c)-[:IN_COUNTRY]->(d)
       where d.name='USA'
       return  a.name as Building , b.name as City, c.name as State;
```