

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES
DEPARTMENT OF MATHEMATICS
AND INFORMATICS



Ramona Molnar

A COMPARATIVE ANALYSIS OF RELATIONAL AND GRAPH DATABASES IN SYSTEMS WITH COMPLEX RELATIONSHIPS BETWEEN ENTITIES

Master Thesis

Supervisor: dr. Lidija Fodor

Novi Sad, 2026

Acknowledgements

I would like to express my sincere gratitude to my supervisor, dr. Lidija Fodor, for her invaluable help and support throughout the writing of this master's thesis. I am especially grateful for her openness, guidance, advice, and patience throughout this process.

My deepest gratitude goes to my family, who have been by my side since the very beginning of my studies. I would like to thank my sister for her constant motivation, and my dear parents for their immeasurable trust, encouragement, and support. They have stood by me through every success and every setback, always believing in me and in my ability to achieve my goals. I dedicate this work to them.

Contents

1. Abstract 4
2. Introduction 5
2.1 Types of Databases 5
3. Entity-Relationship Model 7
3.1 Dataset description 10
3.2 The ER model of the Olist dataset 11
4. Relational Databases 13
4.1 Relational Model Concepts 13
4.1.1 Characteristics of Relations 13
4.1.2 Domain and Key Constraints 14
4.1.3 Relational Databases and Relational Database Schemas 14
4.1.4 Entity Integrity, Referential Integrity, and Foreign Keys 15
4.2 Relational Database Management Systems 15
4.3 Development of the Relational Database for the Olist Dataset 16
4.3.1 Relational Database Schema 16
4.3.2 Description of the Relations 17
5. Graph Databases 20
5.1 Graph Data Model 20
5.1.1 Labeled Property Graph Model 20
5.1.2 Basic Concepts of Graph Structure 21
5.2 Graph Database Management Systems 21
5.3 Development of the Graph Database for the Olist Dataset 22
5.3.1 Description of the Nodes and Relationships 22
6. Related Work 25
6.1 Positioning of the Present Study 26
7. Comparative Analysis 28
7.1 Database Construction and Data Model Implementation 28
7.1.1 Construction of Relational Database 28
7.1.2 Construction of the Graph Database 29
7.2 Basic Query Comparison 29
7.3 Advanced Query Comparison 33
7.3.1 Identification of Similar Customers 33
7.3.2 Recommendation-oriented Queries 41

7.3.3 Seller Network Based on Shared Customers 48
7.3.4 Multi-Step Traversal and the Effect of Path Length 58
8. Conclusion 64
Biography 66
Bibliography 67

1. Abstract

This thesis presents a comparative analysis of relational and graph database approaches using the Brazilian E-Commerce Public Dataset by Olist [15]. The same underlying data was implemented in PostgreSQL using a relational model and in Neo4j using a labeled property graph model. To support the practical comparison, the thesis first provides a concise theoretical overview of the main concepts of relational and graph databases, establishing the background necessary to understand how the two database models represent data, relationships, and database structures. The objective was to examine how the two approaches differ in terms of data modelling, database construction, query formulation, analytical capabilities, and execution performance, particularly as the analysed problems become increasingly relationship oriented.
The comparison includes both basic descriptive queries and more advanced analytical tasks. The latter focus on identifying similar customers based on shared sellers and product categories, generating category and product recommendations, constructing and analysing a seller network, and performing multi-step traversal at increasing path lengths.
The results show that both database systems are suitable for basic queries, where only limited differences in execution time and query complexity were observed. More considerable differences emerged in relationship oriented analyses. PostgreSQL [19] was able to perform the examined tasks through joins, aggregation, recursive common table expressions, and explicitly constructed graph-like representations, and in some cases careful query optimization considerably improved its performance. Neo4j [9], however, provided a more direct representation of connected data and a specialized environment for graph analysis through its native relationships and Graph Data Science algorithms. It also offers dedicated implementations of advanced graph algorithms such as PageRank and Louvain community detection, while reproducing comparable analyses in PostgreSQL would require substantially more complex relational constructions. A particularly clear performance difference was observed in multi-step traversal, where PostgreSQL execution time increased sharply with traversal depth, while Neo4j remained within the millisecond range in the conducted experiment.

Overall, the comparison demonstrates that neither database model offers a universal advantage in all types of analytical tasks. Their relative advantages depend on the structure of the data and the type of analysis being performed.

2. Introduction

Databases [4] are a fundamental component of modern information systems, providing structured mechanisms for storing, managing, retrieving, modifying, and analysing related data efficiently. They are used across a wide range of application domains, including e-commerce, financial services, healthcare, transportation, and social networking, where large amounts of data must be managed reliably and efficiently. A database is typically managed through a Database Management System (DBMS) [6], which provides an interface between users, applications, and the stored data. A DBMS supports operations for defining database structures, inserting, updating, deleting, and retrieving data, as well as executing more complex queries. One of the key responsibilities of a DBMS is to maintain the integrity and consistency of stored data. Integrity constraints can be used to enforce predefined rules on the data, while transaction management and concurrency control mechanisms help preserve consistency when multiple operations or users access the database simultaneously. DBMSs also provide mechanisms for controlling access to data through authentication, authorization, and user privileges. Furthermore, backup and recovery mechanisms can be used to protect data against system failures and support the restoration of the database to a consistent state [1], [6].

2.1 Types of Databases

As the amount and variety of digital data have grown, different database models have been developed to address different application requirements. Database models differ in how they organize data, represent relationships, and support data access. Some of the most important database types are summarized below.

Hierarchical Databases: Hierarchical databases [11] are among the earliest database models. They organize data in a tree-like structure consisting of parent and child records, where each child has only one parent. This structure is suitable for naturally hierarchical data but provides limited flexibility when more complex relationships need to be represented. IBM's Information Management System (IMS) is a well-known example of this model.

Network Databases: A network database [11] organizes data as records connected through explicit links or pointers. Records can have parent–child relationships, and navigation between them is performed by following these predefined connections. Different types of pointers can support forward, backward, or direct access between related records. This structure can provide fast access to connected data, although updates may be more complex because the links between records must also be maintained.

Relational Databases: The relational model [1], [12] was introduced by Edgar F. Codd in 1970 and subsequently became the dominant database model. Data is organized into relations, commonly represented as tables consisting of rows and columns, while relationships between data can be established through keys. Relational database management systems commonly use SQL for defining, manipulating, and querying data and provide transaction management mechanisms to maintain data consistency and integrity.

Object-Oriented Databases: Object-oriented databases [11] store data in the form of objects, similarly to object-oriented programming languages such as Java or C++. They can represent

complex data types and structures, including images, multimedia, hierarchies, and relationships between objects. Although object-oriented database systems became useful in specialized areas such as engineering and geographic applications, they never became as widely adopted as relational databases.

NoSQL Databases: NoSQL databases [11] include a range of non-relational database models designed to provide alternatives to the traditional relational structure. They commonly support flexible data representations and are often used in distributed systems and applications involving large or rapidly changing datasets. Major NoSQL categories include document, key-value, column-family, and graph databases.

Document Databases: Document databases [13] organize data into documents, commonly represented using formats such as JSON or BSON. Documents can have different structures, allowing the database schema to remain flexible. MongoDB is a widely used example of a document-oriented database management system.

Key-Value Databases: Key-value databases [1] store data as pairs consisting of a unique key and an associated value. Their simple structure enables efficient retrieval based on keys and makes them suitable for applications such as caching and session management. However, they generally provide more limited support for complex relationships and queries.

Column-Family Databases: Column-family databases [1] organize data into column families and are designed to support largescale and distributed data processing. Their structure can provide efficient access to selected groups of attributes and is particularly useful for applications involving large volumes of data.

Graph Databases: Graph databases [7], [8] focus on data in which relationships are an important part of the information being represented. Data is modeled using nodes, relationships, and properties, allowing connections between entities to be represented directly within the database structure. Typical application areas include recommendation systems, fraud detection, social networks, and knowledge graphs.

Time-Series Databases: Time-series databases [14] are specialized for storing and processing data associated with timestamps. They are designed for continuously generated observations and measurements and are commonly used for monitoring, sensor and IoT data, and financial or other time-dependent applications.

As the focus of this thesis is the comparison of relational and graph database approaches, these two database models are discussed in greater detail in *Chapters 4* and *5*.

3. Entity-Relationship Model

An Entity-Relationship model (ER model) [1] is a conceptual data model used to describe the structure of a database before it is implemented in a database management system. An ER model represents the structure of a database using a graphical representation called an Entity–Relationship Diagram (ERD). Its main purpose is to represent:

- what data needs to be stored
- how different pieces of data are related
- the rules and constraints that govern those relationships

An ER diagram has three main components: entity, attribute, relationship.

Entity

An entity [1] represents a physical or abstract object in the real world about which information is stored in a database. In an ER model, entities are usually represented by rectangles. Two main types of entities can be distinguished:

- **Strong Entity:** A strong entity can be uniquely identified by its own attributes and does not depend on another entity for identification. It therefore has its own primary key and is represented by a single rectangle in an ER diagram. For example, a Customer can be identified by its customer_id, while an Order can be identified by its order_id. *Figure 1* shows the Customer entity as an example of a strong entity. It can be uniquely identified by its customer_id attribute and also contains several additional attributes.

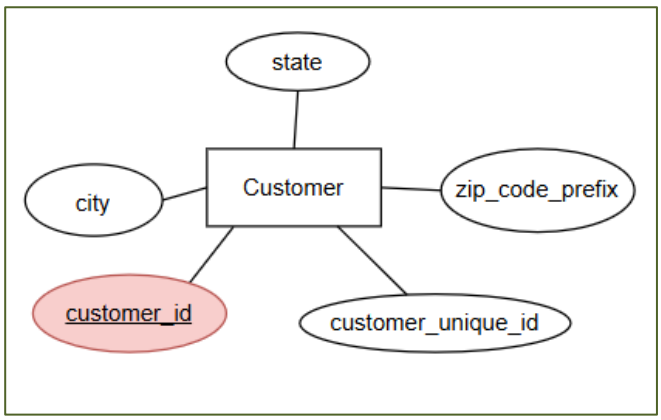


Figure 1: Example of a strong entity and its attributes in an ER diagram

- **Weak Entity:** A weak entity cannot be uniquely identified only by its own attributes. Instead, its identification depends on a related strong entity together with a partial key. A weak entity is represented by a double rectangle, while its identifying relationship is typically represented by a double diamond. For example, an Order Item may be identified by the combination of the related order_id and its order_item_id. Without knowing the corresponding order, the item cannot be uniquely identified. *Figure 2* shows the Order Item entity as an example of a weak entity. Its identification depends on the related Order entity, while order_item_id serves as a partial key.

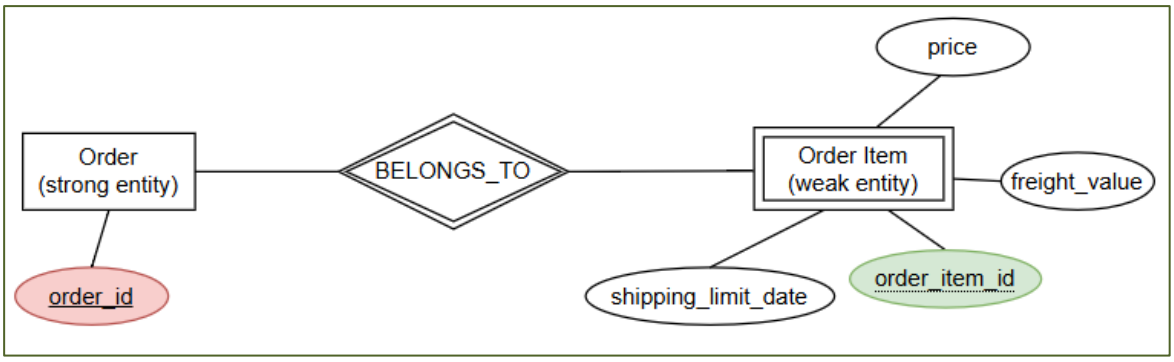


Figure 2: Example of a weak entity and its identifying relationship with a strong entity in an ER diagram

Attribute

Attributes [1] describe the properties or characteristics of an entity and provide additional information about its instances. Depending on their role and structure, several types of attributes can be distinguished in an ER model.

A **key attribute** uniquely identifies each instance of an entity and is represented by an underlined oval. For example, customer_id can serve as the key attribute of the Customer entity. In contrast, a **partial key attribute** belongs to a weak entity and can distinguish weak entity instances related to the same strong entity, but it cannot uniquely identify them independently. For example, order_item_id may act as a partial key for an Order Item, since it has to be considered together with the corresponding order. A partial key attribute is represented by an oval in which the attribute name is underlined with a dashed or dotted line.
Attributes can also differ according to their internal structure. A **simple attribute** cannot be meaningfully divided into smaller components, such as the price of an order item. A **composite attribute**, on the other hand, consists of several related components. For example, an address could be divided into city, street and ZIP code.
Some attributes may contain more than one value for a single entity instance. These are called **multivalued attributes** and are represented by a double oval. For example, a person may have several phone numbers or academic degrees. Finally, a **derived attribute** is calculated from other stored information rather than stored directly. A typical example is age, which can be calculated from a person's date of birth. In an ER diagram, derived attributes are usually represented by dashed ovals.
Figure 3 illustrates several common attribute types used in ER modeling. The Customer entity is connected to a simple attribute (state), a key attribute (customer_id), a composite attribute (address) consisting of city, street, and zip_code, and a derived attribute (age). The figure also shows the different graphical notations used to distinguish these attribute types.

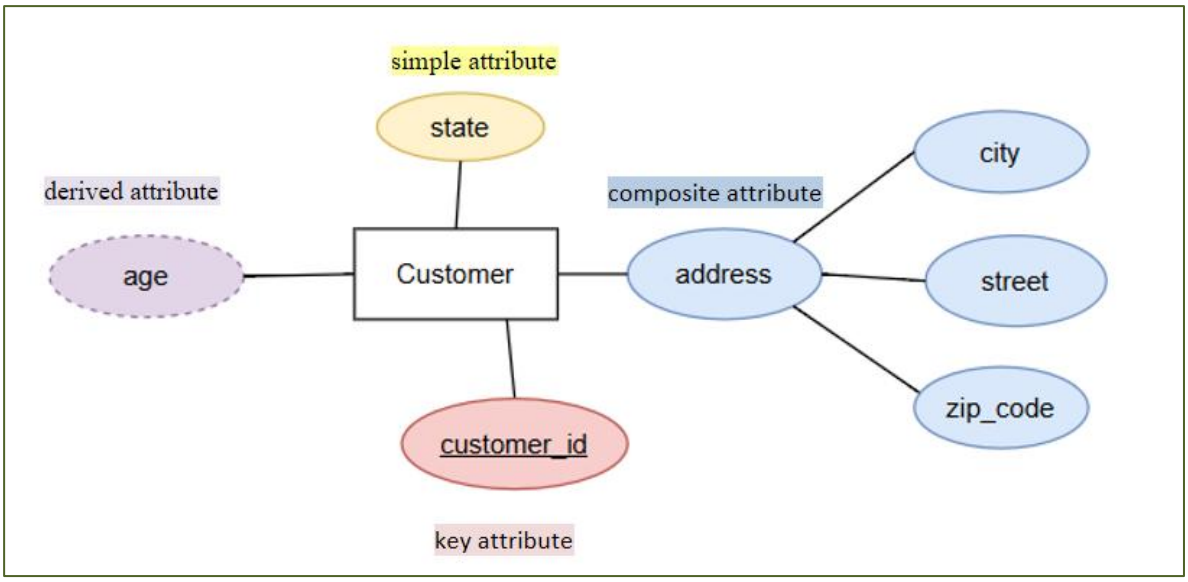


Figure 3: Examples and graphical representation of different attribute types in an ER diagram

Relationship

A relationship [1] represents an association between two or more entities and describes how their instances are connected. Relationships make it possible to model the dependencies and interactions between different parts of the data. In an ER diagram, a relationship is typically represented by a diamond shape connected to the participating entities. A relationship may also contain its own attributes when additional information about the association needs to be stored. An important property of a relationship is its cardinality, which defines how many instances of one entity may be associated with instances of another entity. There are four types of cardinal relationships:

- **One to One (1:1):** A one to one relationship means that one instance of an entity is associated with only one instance of another entity, and vice versa. For example, a customer could be associated with one loyalty card, while each loyalty card belongs to only one customer.
- **One to Many (1:N):** In a one to many relationship, one instance of the first entity can be associated with several instances of the second entity, while each instance of the second entity is associated with only one instance of the first. For example, one customer can place several orders, but each order is placed by one customer.
- **Many to One (N:1):** A many to one relationship describes the same type of association from the opposite direction, which means that multiple instances of one entity can be associated with a single instance of another entity. For example, multiple Customer entities may be associated with the same Geolocation, while each customer is linked to one geographical location.
- **Many to Many (N:M):** A many to many relationship allows multiple instances of both entities to be associated with each other. For example, one order may contain several products, while the same product may appear in several different orders.

Figure 4 illustrates the four main cardinality types used in ER modeling. A (1:1) relationship is shown between Customer and Loyalty Card, while a (1:N) relationship connects Customer and Order. The (N:1) example shows multiple Customer entities associated with one Geolocation, and the (N:M) relationship is illustrated between Order and Product.

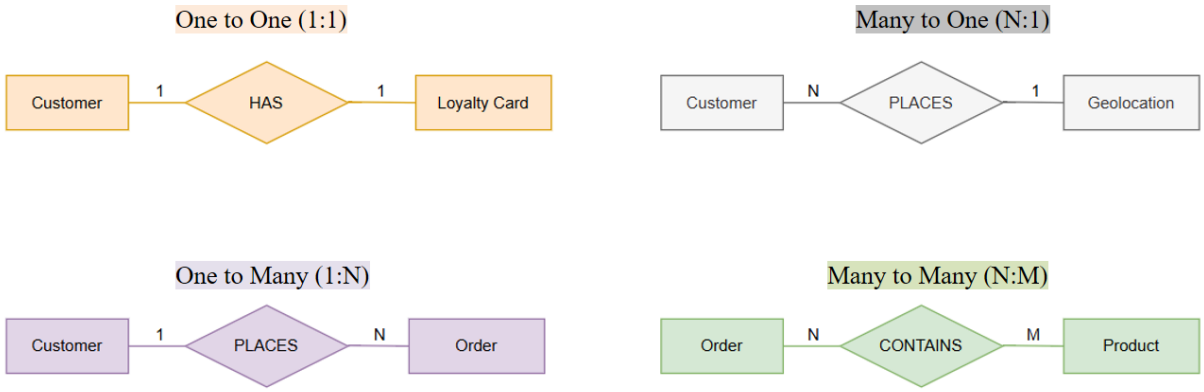


Figure 4: Examples of the main relationship cardinalities in an ER diagram

3.1 Dataset description

The dataset used throughout this thesis is the Brazilian E-Commerce Public Dataset by Olist [15], obtained from Kaggle. Olist is a Brazilian e-commerce platform, and the dataset contains information about approximately 100,000 orders placed between 2016 and 2018. It provides information from several perspectives of the e-commerce process, including customers, orders, purchased products, sellers, payments, customer reviews, and geographical information.

The dataset is divided into eight CSV files according to the type of information they contain. The customers file stores customer identifiers and location information, while the orders file describes individual orders, their status, and timestamps associated with different stages of order processing. The order_items file contains the individual products purchased within each order and identifies the corresponding seller. Product characteristics are stored in the products file, while seller information is provided in the sellers file. Additional files contain payment transactions, customer reviews, and geographical information.

The files are connected through common identifiers. The customer_id connects customers to orders, while order_id links orders to order items, payments, and reviews. Each order item contains a product_id and a seller_id, connecting it to the corresponding product and seller. Customer and seller ZIP code prefixes can also be linked to the geolocation data. These connections make it possible to reconstruct the main parts of the purchasing process across the different files.

The individual files also differ considerably in size. For example, the dataset contains 99 441 customer records and 99 441 order records, 112 650 order item records, 103 886 payment records, 32 951 product records, and 3 095 seller records. The geolocation file is considerably larger, containing more than one million rows because multiple coordinates may be associated with the same ZIP code prefix.

This structure makes the dataset particularly suitable for the objectives of this thesis. On the one hand, the separate but interconnected files naturally support a relational representation based on relations, primary keys, and foreign keys. On the other hand, the large number of connections between customers, orders, products, sellers, reviews, payments, and locations also provides a suitable basis for representing and analysing the same data as a graph. This allows the two database approaches to be compared using the same underlying e-commerce information.

3.2 The ER model of the Olist dataset

To provide a clearer representation of the structure of the Olist dataset, an ER model was developed based on the theoretical concepts discussed above. The model illustrates how the entities within the dataset are organized, the attributes that describe them, and the relationships that connect them.

As shown in *Figure 5*, the Order entity occupies a central position in the ER model, connecting the main components of the e-commerce process. Each order is associated wih a customer and can be linked to purchased products and sellers through order items, as well as to payment and review information. Each customer can place multiple orders, while every order is associated with exactly one customer, representing a one to many relationship. An order may contain multiple order items, each corresponding to a specific product sold by a seller. This structure also enables a single order to contain products offered by different sellers. Products and sellers are connected through the Order Item entity, which acts as an associative entity and stores transaction-specific information such as price, freight value, and shipping limit date. Additional entities provide complementary information about each transaction. The Order Payment entity stores payment details, including the payment sequential, payment type, number of installments, and payment value . The Order Review entity captures customer feedback through review scores, comments, and relevant timestamps. Customer and seller locations are represented through the Geolocation entity, which stores geographical information such as ZIP code prefix, city, state, latitude, and longitude. Both Customer and Seller are linked to this entity, enabling the spatial analysis of purchasing and selling activities.
The key attributes of the strong entities are highlighted in red in *Figure 5*. These attributes uniquely identify individual instances of entities such as Order, Customer, Seller, Product, and Geolocation. The weak entities Order Item, Order Payment, and Order Review depend on the Order entity for their identification. Their partial key attributes are highlighted in green and distinguish weak entity instances associated with the same order. Together with the key attribute of the identifying Order entity, they uniquely identify each weak entity instance.

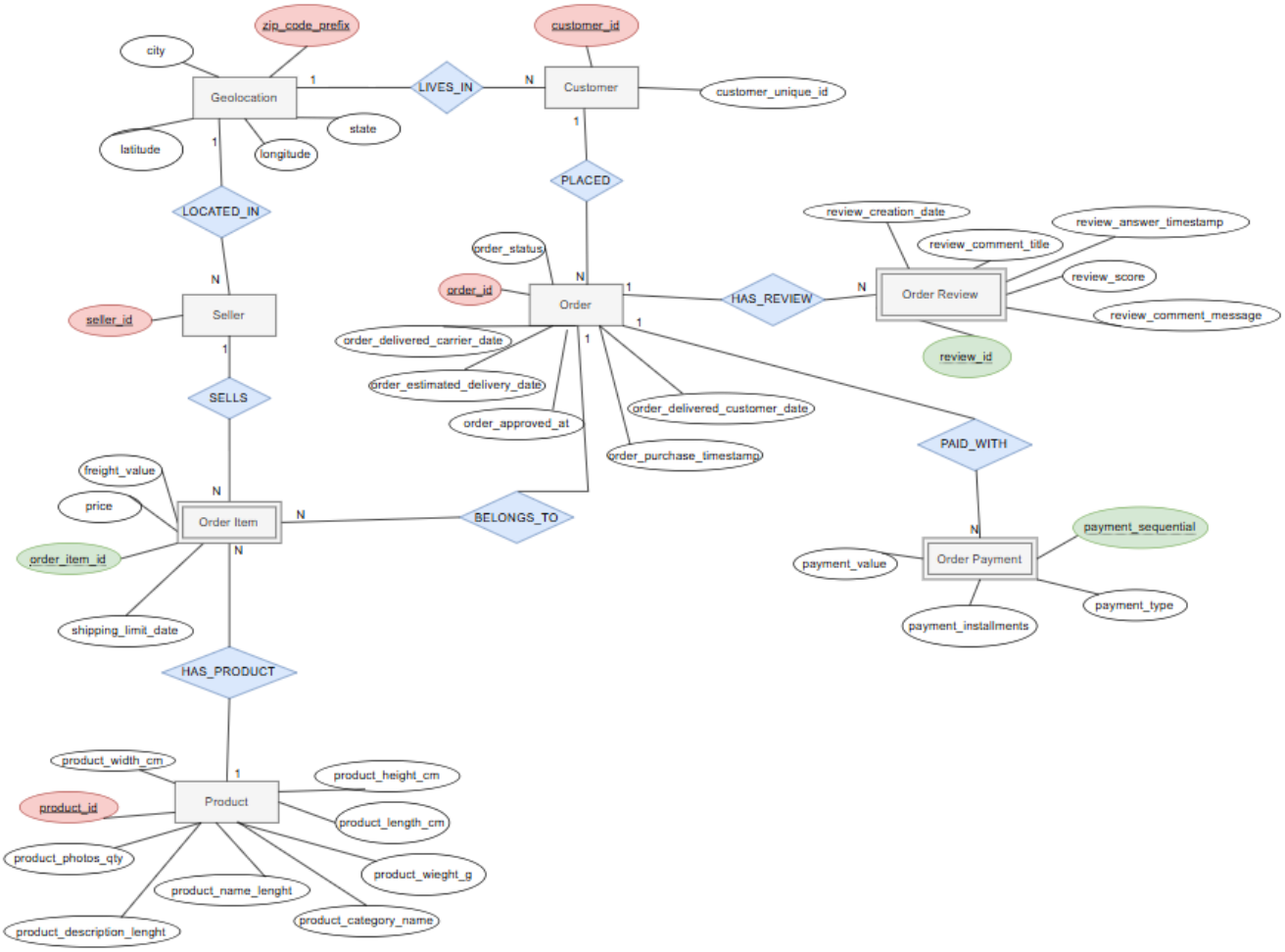


Figure 5: ER model of the Olist dataset

The ER model provides a clear overview of the structure of the Olist dataset by showing how its main entities are connected. It also serves as a basis for the subsequent database implementation and analysis by identifying the relationships and dependencies between the different data components.

4. Relational Databases

Relational databases represent one of the most common approaches to structured data management and form one of the central areas examined in this thesis. Since the main objective of this work is to compare relational and graph database approaches, understanding their underlying concepts is essential for interpreting the differences observed in their data models, query formulation, and analytical capabilities. Therefore, this chapter presents the most relevant theoretical foundations of relational databases, including the relational model, its fundamental concepts and integrity constraints, relational database management systems, and Structured Query Language (SQL).

4.1 Relational Model Concepts

The relational model [1] represents a database as a collection of relations. A relation can be viewed informally as a table consisting of rows and columns. In formal relational terminology, a row is referred to as a tuple, while a column is represented by an attribute. Each tuple describes a particular fact represented in the database, while the attributes determine the meaning and structure of the values stored within it.
Each attribute is associated with a domain, which defines the set of acceptable atomic values for that attribute. A domain therefore determines the type and possible values that can occur for a particular attribute. For example, in the Olist dataset, the order_status attribute of the orders relation contains values describing the current state of an order, while order_purchase_timestamp contains temporal values representing when orders were placed. Similarly, the price attribute of order_items represents numerical monetary values.
The structure of a relation is formally described by a **relation schema**, which can be expressed as $R(A_1, A_2, ..., A_n)$, where R is the name of the relation and $A_1, A_2, ..., A_n$ are its attributes. The number of attributes in the schema is called the degree, or arity of the relation [1]. For example, a simplified Olist product relation could be represented as

PRODUCTS(product_id, product_category_name, product_description_lenght),

where PRODUCTS is the relation name and product_id, product_category_name and product_description_lenght are its attributes.

While a relation schema describes the relatively stable structure of a relation, the **relation state** [1] represents the current set of tuples contained in the relation at a particular point in time. Consequently, the schema generally remains unchanged while the state can change when tuples are inserted, modified, or deleted. A relation state must conform to the structure and constraints specified by its relation schema.

4.1.1 Characteristics of Relations

Relations have several characteristics that distinguish the relational model from a simple collection of tabular records. First, a relation is mathematically defined as a set of tuples [1]. As a consequence, tuples have no inherent ordering. Although rows are necessarily stored and displayed in some physical sequence, this order has no semantic significance in the relational model. If a particular ordering is required when retrieving data, it must be explicitly specified by the query.

Similarly, the presentation order of attributes does not determine their meaning at the conceptual level. Values are associated with attributes, and changing the order in which columns are displayed does not change the meaning of a relation as long as the correspondence between attributes and their values is preserved [1].
Another fundamental characteristic is that attribute values are considered atomic [6] within the basic relational model. This means that a single attribute value is treated as an indivisible value in the context of the model. This principle corresponds to the first normal form (1NF) assumption and prevents multivalued attributes from being directly represented within a single attribute.
A tuple may also contain the special marker NULL [1] to represent the absence of a meaningful attribute value. NULL is different from zero or an empty string and may indicate that a value is unknown, unavailable, or not applicable to a particular tuple. Since NULL does not represent an ordinary domain value, it requires special treatment in operations such as comparisons and logical expressions.

4.1.2 Domain and Key Constraints

Relations are subject to constraints that restrict the values and combinations of values that may occur within them. Among the fundamental constraints of the relational model are domain constraints and key constraints [1].

For an attribute A_i of a relation, let D_i denote its domain. A **domain constraint** requires that the value of A_i in every tuple belongs to this domain:

$$t[A_i] \in D_i.$$

In practice, the domain may be restricted by the attribute's data type, format, or allowable range. For example, in the Olist dataset, the price attribute of the order_items relation contains numerical values, while the timestamp attributes of the orders relation contain temporal values.
Key constraints [1] provide a mechanism for uniquely identifying tuples within a relation. Let K denote a set of attributes. K is a **superkey** if no two distinct tuples have the same values for all attributes in K. Formally,

$$t_1[K] = t_2[K] \Longrightarrow t_1 = t_2$$

for any tuples t_1 and t_2 in the relation. A **candidate key** is a minimal superkey, meaning that no proper subset of K preserves this uniqueness property. If several candidate keys exist, one may be selected as the **primary key**, while the remaining candidate keys are considered alternate keys. For example, in the Olist dataset, order_id serves as the primary key of the orders relation, while customer_id and seller_id serve as the primary keys of the customers and sellers relations, respectively. Each of these attributes uniquely identifies a tuple within its corresponding relation.

4.1.3 Relational Databases and Relational Database Schemas

The concepts introduced so far describe individual relations and the constraints that apply to them. In practice, however, related information is commonly distributed across multiple relations. A relational database consists of a collection of relations whose data may be related to one another.

A **relational database schema** [1] defines the set of relation schemas together with the integrity constraints that apply to them. The corresponding **relational database state** [1] represents the collection of relation states at a particular point in time. A database state is considered valid when

the individual relation states conform to their schemas and satisfy the specified integrity constraints.

4.1.4 Entity Integrity, Referential Integrity, and Foreign Keys

In addition to constraints that apply to individual relations, a relational database must maintain consistency both within and between its relations. Two fundamental constraints serving this purpose are entity integrity and referential integrity.

The **entity integrity constraint** [1] requires that no attribute forming part of a primary key can contain a $NULL$ value. Formally, let PK denote the primary key of a relation R, then for every tuple $t \in R$,

$$t[A] \neq NULL \qquad \forall A \in PK.$$

This condition ensures that every tuple can be uniquely identified. Since the primary key is used for identification, allowing a $NULL$ value in any of its attributes would prevent the reliable identification of the corresponding tuple.
Referential integrity [1] concerns associations between different relations and is enforced through foreign keys. A foreign key is an attribute or a set of attributes in one relation that references a key in another relation. Suppose FK is a foreign key in relation R_1 that references a key K in relation R_2. Referential integrity requires that, for every tuple $t_1 \in R_1$,

$$t_1[FK] \in \{t_2[K] | t_2 \in R_2\}$$

or, when $NULL$ values are permitted,

$$t_1[FK] = NULL.$$

In other words, each non-NULL foreign key value must correspond to an existing key value in the referenced relation. This prevents references to entities that do not exist. For example, in the Olist database, customer_id in the orders relation references customer_id in the customers relation. Therefore, every non-NULL customer_id appearing in orders must also exist in customers. This ensures that an order cannot refer to a customer that is absent from the database.

4.2 Relational Database Management Systems

A **Relational Database Management System (RDBMS)** [1] is a software system used to define, store, manipulate, and retrieve data organized according to the relational model. In addition to providing mechanisms for creating and querying relations, an RDBMS is responsible for enforcing integrity constraints and managing the physical storage and processing of data.
Modern relational database systems also provide functionality such as transaction management, concurrency control, indexing, access control, backup, and recovery. These mechanisms enable multiple users and applications to interact with a database while preserving the consistency and reliability of stored information.
PostgreSQL [16] is an open-source object-relational database management system that implements the fundamental concepts of the relational model while providing additional functionality for complex data management and analysis. It supports primary and foreign keys, integrity constraints, transactions, indexes, views and materialized views, as well as advanced SQL functionality such as common table expressions, recursive queries, aggregate functions, and

window functions. PostgreSQL also includes a query planner and optimizer, which determines how SQL statements should be executed. Rather than requiring the user to specify the exact physical execution procedure, PostgreSQL evaluates possible execution strategies based on factors such as table statistics, available indexes, join methods, and estimated processing costs. This distinction between the declarative specification of a query and its physical execution is particularly relevant when analysing query performance.

Structured Query Language (SQL) [1] is the principal language used to interact with relational database management systems. SQL is declarative, meaning that a query specifies the required result rather than the exact sequence of internal operations needed to obtain it. The database management system is responsible for determining an appropriate execution strategy. SQL provides statements for several aspects of database management. Data Definition Language (DDL) statements define and modify database structures, including relations and constraints, while Data Manipulation Language (DML) statements are used to insert, update, and delete stored data. Data retrieval is primarily performed using the `SELECT` statement.

Queries can filter tuples, select specific attributes, group data, calculate aggregate values, and order query results. When the required information is distributed across multiple relations, join operations combine related tuples according to specified conditions. In normalized relational databases, joins play an important role because data related to the same business process is often distributed among several relations.

More complex operations can additionally be expressed through subqueries and Common Table Expressions (CTEs), while recursive CTEs allow SQL to process data structures or relationships that require repeated application of the same query logic [1].

4.3 Development of the Relational Database for the Olist Dataset

This chapter demonstrates the implementation of the Olist dataset as a relational database in PostgreSQL, building on the ER model and relational concepts introduced in the previous chapters. It describes the transformation of the conceptual model into a relational schema, including the definition of tables, primary and foreign keys, and relationships between them. The resulting database structure and the integrity constraints applied to maintain data consistency and referential integrity are also discussed.

4.3.1 Relational Database Schema

Based on the ER model presented in *Chapter 3*, the relational schema of the Olist database was developed by mapping the identified entities to tables and representing their relationships through foreign keys. *Figure 6* presents the resulting database schema and shows how the individual tables are connected.

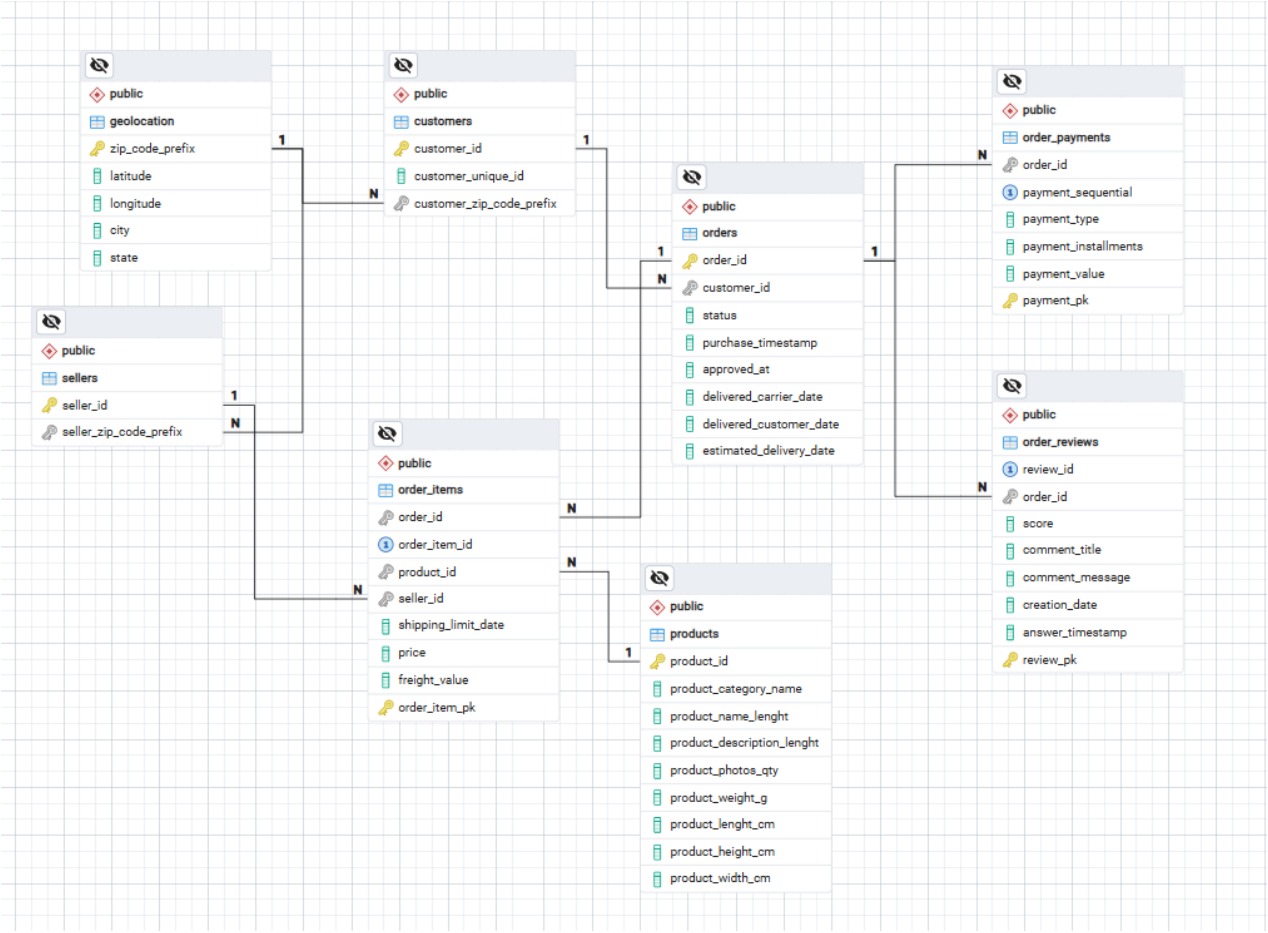


Figure 6: Relational database schema of the Olist dataset implemented in PostgreSQL.

4.3.2 Description of the Relations

This subsection provides a detailed description of the relations that constitute the relational implementation of the Olist dataset. It outlines the role of each relation within the schema, identifies the relevant primary and foreign keys, and explains how the relations are connected to one another. Where modifications to the original dataset were necessary, these are also described in order to clarify the design decisions made during the construction of the relational database.

The geolocation relation stores geographical information associated with ZIP code prefixes and serves as a reference table for the customers and sellers relations. The primary key is zip_code_prefix, which is referenced as a foreign key by both relations. Storing the geographical information separately also avoids repeating the corresponding location data for individual customers and sellers. In the original Olist dataset, multiple geolocation records may share the same geolocation_zip_code_prefix while containing different latitude and longitude values. Therefore, geolocation_zip_code_prefix cannot directly be used as a primary key, since it does not uniquely identify individual records. To obtain a single record for each ZIP code prefix, the geolocation data was grouped by geolocation_zip_code_prefix, and the arithmetic mean of the latitude and longitude values was calculated for each group. The resulting relation contains one representative coordinate pair for each ZIP code prefix, allowing zip_code_prefix to be used as its primary key and referenced by the customers and sellers relations. This transformation results in a loss of detail in the coordinates, since multiple original coordinate values are represented by their

average. However, this representation is sufficient for the purposes of this database, where geographical information is used to associate customers and sellers with ZIP code prefixes and to support analysis by region rather than to determine precise locations.

The customers relation stores information about customers who have placed orders through the Olist platform. Each tuple is uniquely identified by the customer_id attribute, which serves as the primary key of the relation. The relation also contains customer_unique_id, which allows the same customer to be identified across multiple customer records. To avoid storing geographical information redundantly, the customer city and state attributes from the original dataset were not retained in the customers relation, as this information is represented separately in the geolocation relation. Instead, customer_zip_code_prefix is retained as a foreign key referencing zip_code_prefix in geolocation. This establishes the relationship between each customer and the corresponding geographical area while keeping location related information in a separate relation.

The sellers relation stores information about sellers participating in the Olist marketplace. The primary key of the relation is seller_id, which uniquely identifies each seller. Similar to the customers relation, the seller city and state attributes from the original dataset were not retained, as geographical information is represented separately in the geolocation relation. Instead, seller_zip_code_prefix is used as a foreign key referencing zip_code_prefix in geolocation. This avoids redundant storage of geographical information and allows both customers and sellers to reference the same relation for their corresponding geographical areas.

The orders relation stores information about order placed through the Olist platform and represents a central relation within the relational schema. Each order is uniquely identified by the order_id attribute, which serves as the primary key. The relation also stores the status of each order and timestamps corresponding to different stages of the order lifecycle, including purchase, approval, delivery, and estimated delivery. The customer_id attribute is defined as a foreign key referencing the customers relation, establishing a one to many relationship in which a customer may place multiple orders, while each order belongs to exactly one customer. The orders relation is also referenced by the order_items, order_payments, and order_reviews relations, allowing each order to be associated with its purchased items, payment information, and review data.

The products relation stores descriptive information about the products available on the Olist platform. Each tuple corresponds to a single product and is uniquely identified by the product_id attribute. In addition to the product identifier, the relation contains the product category, the lengths of the product name and description, the number of product images, and the product's physical characteristics, including its weight and package dimensions. The products relation is referenced by the order_items relation through the product_id foreign key, allowing individual products to be associated with the orders in which they were purchased.

The order_items relation represents the individual items included in each order and connects the orders, products, and sellers relations. Each tuple represents a particular product item within an order, allowing an order to contain multiple items and the same product to occur in multiple orders. In the original Olist dataset, an order item can be uniquely identified by the combination of order_id and order_item_id. In the implemented relational database, a surrogate primary key, order_item_pk, was introduced to provide each tuple with a single-attribute identifier. The relation contains three foreign keys: order_id, referencing the orders relation; product_id, referencing the products relation; and seller_id, referencing the sellers relation. In addition, it contains information about each item, such as the shipping limit date, price, and freight value.

The order_payments relation contains information about payment transactions associated with orders. Individual payments are distinguished by using composite key (order_id, payment_sequential). However a surrogate primary key, payment_pk, was introduced in the relational database implementation to provide each tuple with a single-attribute identifier. The order_id attribute is therefore retained as a foreign key referencing the orders relation, while the order_payments also includes attributes describing the payment type, the number of payment installments, payment value and payment_sequential, which indicates the sequence of payments associated with an order.

The order_reviews relation stores customer reviews submitted for completed orders. Since review_id alone does not uniquely identify each tuple, as the same identifier may occur in association with different orders, a surrogate primary key, review_pk, was introduced to provide each tuple with a unique single-attribute identifier. The order_id attribute is retained as a foreign key referencing the orders relation, establishing the association between each review and its corresponding order. In addition to review_id, the relation stores the review score, title, message, creation date, and answer timestamp.

5. Graph Databases

Graph databases represent the second major database approach examined in this thesis. The following chapter introduces the most relevant concepts of the graph data model, with particular attention to nodes, relationships, properties, paths, and graph traversal. It also presents graph database management systems and the Cypher query language used to query and manipulate graph data. Together, these concepts provide the theoretical foundation for the implementation of the Olist dataset in Neo4j and its subsequent comparison with the relational approach.

5.1 Graph Data Model

The graph data model [8] is a logical data model designed to represent, store, and manage highly interconnected data. It is based on the principles of graph theory, in which information is represented as a collection of vertices connected by edges. Unlike traditional database models that primarily organize data according to predefined structures such as tables or hierarchical trees, the graph model focuses on the relationships between entities and considers these relationships to be an integral part of the stored data.

Mathematically, a graph can be represented as an ordered pair

$$G = (V, E)\,,$$

where V denotes a finite set of vertices (or nodes) and E denotes a finite set of edges connecting pairs of vertices [7].

5.1.1 Labeled Property Graph Model

Modern graph database systems commonly represent data using the labeled property graph model [8]. In this model, data is represented through nodes, relationships, labels, relationship types, and properties. Nodes represent entities and may be assigned one or more labels that describe their roles within the graph. Both nodes and relationships may contain properties, represented as key-value pairs, which store descriptive information about them. For example, in the graph representation of the Olist dataset, a customer can be represented by a node labeled `Customer`, while an order is represented by an `Order` node. Properties such as `customer_id` and `order_id` store identifying information directly on the corresponding nodes.

Vertices are connected through relationships, which represent associations between entities. Each relationship has a type and a direction and may also contain properties [8]. For example, the pattern `(Customer)-[:PLACED]->(Order)` expresses that a customer placed a particular order. The relationship type therefore provides semantic meaning to the connection itself.

The labeled property graph model also provides structural flexibility, as different node types may contain different properties and new node labels, relationship types, or properties can be introduced as the data model evolves. This makes it possible to extend the graph structure without necessarily redesigning existing data representations.

5.1.2 Basic Concepts of Graph Structure

Paths and graph traversal [8] are fundamental concepts in graph data processing. A path is a sequence of nodes connected through relationships, and its length is determined by the number of relationships contained in the path. Paths may have a fixed length or a variable length, allowing connections to be explored even when the number of intermediate nodes is not known in advance. Graph traversal refers to navigating through the graph by following relationships from one node to another. Traversal can be used to examine immediate neighbors, determine which nodes are reachable from a starting node, or discover paths satisfying particular structural conditions. Several additional concepts are useful for describing graph structure. The neighborhood [17] of a node consists of the nodes directly connected to it, while its degree [17] represents the number of incident relationships. In directed graphs, incoming and outgoing relationships can be distinguished through in-degree and out-degree [17]. Relationships may also contain numerical weights representing the strength, frequency, or other characteristics of a connection. A selected subset of nodes and relationships forms a subgraph, allowing particular portions of a larger graph to be analysed independently. These concepts provide a foundation for understanding the structure of graphs and for analysing relationships between connected entities.

5.2 Graph Database Management Systems

Graph Database Management Systems (GDBMSs) [8] provide the software infrastructure required for storing, querying, and managing data according to a graph data model. In addition to supporting graph oriented data representation and querying, they provide database management functionality such as transaction processing, indexing, security, and mechanisms for maintaining data consistency. Different graph database systems may implement these capabilities using different graph models and underlying architectures.

Neo4j [9], the graph database management system used in this work, is based on the labeled property graph model. Neo4j represents data through nodes, relationships, labels, and properties, with relationships explicitly connecting pairs of nodes.

An important feature of Neo4j is index-free adjacency, which supports efficient navigation through connected data. Instead of relying on global indexes each time a connection is followed, nodes maintain direct references to their neighboring nodes. Once a starting node has been found, the database can therefore continue the traversal by following these connections directly. As a result, traversal mainly depends on how much of the graph needs to be explored rather than on the total size of the database. Indexes still play an important role, particularly in locating the starting nodes of a query based on their labels and properties [8].

Data in Neo4j is queried and manipulated primarily through **Cypher** [9], a declarative graph query language. Rather than specifying the procedural steps required to retrieve data, Cypher allows users to describe the graph patterns they want to match.

Its syntax reflects the structure of a property graph: nodes are represented using parentheses, while relationships are represented using arrows and square brackets. Labels, relationship types, and properties can be integrated directly into these patterns [5]. For example, the pattern `(s:Seller)-[:LOCATED_IN]->(g:Geolocation)` represents a `Seller` node connected to a `Geolocation` node through a `LOCATED_IN` relationship. The database system is responsible for transforming such a declarative query into an appropriate execution plan.

Neo4j also provides constraints and indexes [8] to support data integrity and query performance. Constraints can enforce requirements such as the uniqueness, existence, or type of selected

properties, while indexes provide efficient access to nodes or relationships based on their labels, relationship types, and property values.
Beyond transactional graph querying, Neo4j provides analytical graph-processing functionality through the Graph Data Science (GDS) library [18]. To support efficient graph analysis, GDS loads graph data into an in-memory graph catalog, where graphs are created through graph projections from the stored Neo4j database. A projection can select the node labels, relationship types, properties, and relationship orientations required for a particular analysis. Graph algorithms are then executed on the projected graph. GDS includes algorithms for several classes of graph-analysis problems, including centrality, community detection, similarity, and path finding.

5.3 Development of the Graph Database for the Olist Dataset

Based on the structure of the Olist dataset and the concepts introduced in the previous sections, a graph database was developed in Neo4j to provide an alternative representation of the same data. The graph model represents the main entities as nodes and their associations as explicit relationships, while descriptive information is stored as properties of nodes or relationships. *Figure 7* demonstrates a schema visualization of the implemented graph database, showing the main node labels and the relationship types connecting them. The visualization provides an overview of the graph structure rather than displaying individual node instances and their data.

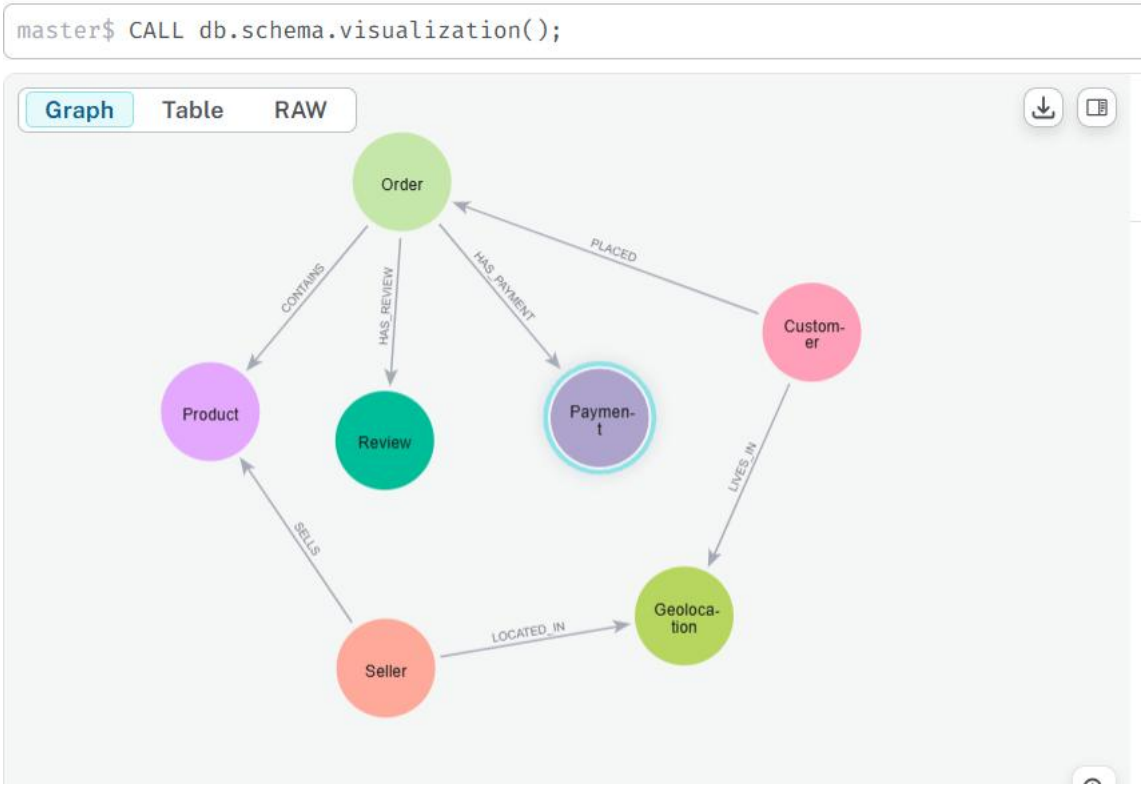


Figure 7: Graph schema in Neo4j

5.3.1 Description of the Nodes and Relationships

This subsection describes the main node types and relationships used in the Neo4j implementation of the Olist dataset. It explains the role of each node type, the properties used to identify and describe the represented entities, and the relationships through which the nodes are connected.

Particular attention is also given to structural differences from the relational implementation, especially where information represented as a separate relation in PostgreSQL is modeled directly through relationships and relationship properties in Neo4j.

Each Customer node represents one row from the customer CSV file in the Olist dataset. Each node is identified by customer_id and also contains customer_unique_id, which allows the same customer to be recognized across different Customer nodes. These nodes are connected to orders through PLACED relationships and to geographical locations through LIVES_IN relationships.

The Order node represents an individual order and is identified by order_id. In addition to its identifier, it stores the order status and timestamps associated with different stages of order processing, including purchase, approval, carrier delivery, customer delivery, and estimated delivery dates. Each order is connected to the customer who placed it through a PLACED relationship and to its purchased products through CONTAINS relationships. It is also connected to payment and review nodes through HAS_PAYMENT and HAS_REVIEW relationships, respectively.

The Product node represents a product available through the Olist marketplace and is identified by product_id. Along with the identifier, its properties contain descriptive product information, including the product category. Products are connected to orders through CONTAINS relationships and to sellers through SELLS relationships.

The Seller node represents a seller participating in the marketplace and is identified by seller_id. Sellers are connected to products through SELLS relationships and to their geographical locations through LOCATED_IN relationships.

The Geolocation node represents a geographical area identified by zip_code_prefix. Its properties include latitude, longitude, city, and state. Customer and Seller nodes are connected to geographical locations through LIVES_IN and LOCATED_IN relationships, respectively, allowing geographical information to be represented separately from the corresponding customer and seller nodes. To ensure consistency between the two database implementations, Neo4j uses the same processed geographical data and ZIP code prefix representation as PostgreSQL.

Within the graph model, customer feedback associated with an order is represented as a Review node. Its properties include the review identifier, score, title, message, creation date, and answer timestamp. Each Order is connected to its corresponding Review through a HAS_REVIEW relationship.

The Payment node represents a payment transaction associated with an order. Its properties describe the payment sequence, payment type, number of installments, and payment value. Each payment is associated with its corresponding order through a HAS_PAYMENT relationship from the Order node.

An important structural difference from the relational implementation relates to the representation of order items. In PostgreSQL, order_items is represented as a separate relation connecting orders, products, and sellers while storing item specific transaction information. In the Neo4j model, a separate OrderItem node was not created. Instead, each order item is represented by a CONTAINS relationship between an Order and a Product. The properties order_item_id, price, freight_value, shipping_limit_date, and seller_id are stored directly on the CONTAINS relationship because they describe a particular occurrence of a product within an order rather than the product itself. Multiple occurrences of the same product within an order can therefore be represented by separate

CONTAINS relationships, each retaining its own properties related to the individual item. The seller_id property identifies the seller associated with the particular order item.

The relationships forming the basic graph model are summarized in *Table 1*, including their direction and the properties stored on each relationship.

RELATIONSHIP	DIRECTION	RELATIONSHIP PROPERTIES
PLACED	Customer→ Order	None
CONTAINS	Order→ Product	order_item_id, price, freight_value, shipping_limit_date, seller_id
SELLS	Seller→ Product	None
HAS_REVIEW	Order→ Review	None
HAS_PAYMENT	Order→ Payment	None
LIVES_IN	Customer→ Geolocation	None
LOCATED_IN	Seller→ Geolocation	None

Table 1: Relationships of the basic Olist graph model

6. Related Work

The comparison of relational and graph databases has been addressed in several previous studies, particularly with regard to query performance, recursive operations, pattern matching, and the processing of highly connected data. Although these studies use different datasets, database systems, and experimental settings, their results generally indicate that neither database model is universally superior. Instead, the relative performance and suitability of each approach depend on the structure of the data, the type and complexity of the queries, and the optimization techniques applied.

One of the earlier direct comparisons was presented by Vicknair et al. [20], who compared Neo4j with MySQL in the context of data provenance. Their work was motivated by the observation that relational database systems are generally efficient for conventional data storage and retrieval, but queries involving many relationships may require a considerable number of joins. Since provenance information can naturally be represented as a directed graph, the authors investigated whether a graph database could provide a more suitable solution for storing and querying such data. They also pointed out that although graph structured information can be stored in a relational database, traversing the resulting structure may become inefficient because of the number of operations required to follow relationships. Their experiments showed that Neo4j performed particularly well for traversal operations, whereas traditional data queries could favour the relational implementation. This suggests that the characteristics of the query are an important factor when choosing between the two database models.
A more recent query based comparison is presented in [21], where Do et al. evaluated Neo4j and MySQL using the real world Career Village dataset. The evaluated queries were divided into four main categories: selection and search, recursion, aggregation, and pattern matching. This categorization is relevant because it separates conventional database operations from queries that depend more strongly on relationships and graph structure. In their experiments, Neo4j generally achieved lower execution times than MySQL, with the largest differences appearing in recursive and relationship intensive queries. They concluded that queries requiring complex joins across several relational tables were particularly favourable to the graph representation.
Similar results were reported by Jain, Khanchandani, and Rodrigues in [22]. Their study also compared Neo4j and MySQL using selection, recursion, aggregation, and pattern matching queries on a real world dataset. The authors reported that Neo4j performed better in most of their tests, particularly for recursive and pattern matching operations, while also acknowledging advantages of relational databases in areas such as consistency and transactional support. Their work further emphasizes that graph databases can provide a natural representation when the analysed data contains a large number of interconnected entities and the queries frequently follow relationships. However, the results of previous research do not consistently indicate that graph databases outperform relational systems. Hölsch, Schmidt, and Grossniklaus [23] compared Neo4j with a relational database using both analytical graph queries and different categories of pattern matching queries. Their analytical workload included operations such as node degree, degree centrality, connectedness, shortest paths, diameter, and closeness centrality. They found that the relational database performed better for the analytical queries used in their experiments, whereas Neo4j showed advantages for certain path queries, particularly when specific relationship types did not need to be filtered. An important contribution of [23] is the distinction between different forms of graph queries. The authors argue that graph database performance cannot be evaluated only by considering whether a query is graph oriented, because characteristics such as filtering by node labels, filtering by relationship types, the presence of cycles, and the proportion of the graph accessed can significantly influence execution behaviour. Their work also demonstrates that several graph measures can be implemented in both database models. For example, node degree,

degree centrality, and connectedness are expressed using Cypher in Neo4j and corresponding SQL statements in the relational database. This is particularly relevant to the present thesis, where similar network properties are also implemented and compared in relational and graph form.
The influence of query complexity and database optimization is examined in greater detail by Kotiranta, Junkkari, and Nummenmaa [24]. Their study compared Neo4j with MySQL and MariaDB and explicitly considered factors such as indexing, query optimization, aggregation, and recursive operations. The authors describe an important structural difference between the two approaches: relational queries typically combine related information through joins, whereas graph queries navigate through connections between nodes. Nevertheless, their results demonstrate that graph based navigation does not automatically lead to better performance for every type of query. In [24], some simple and relationship oriented queries favoured Neo4j, while modern relational systems performed better for several more complex queries. The experiments also showed that indexing could substantially influence the results, particularly in relational databases. The authors therefore concluded that query complexity, indexing, and optimization must be considered when comparing database systems and that no general conclusion can be made about which model always provides better query efficiency. This observation is especially important because it demonstrates that performance comparisons should not attribute differences exclusively to the underlying database model.
An earlier comparative study by Tyagi [26] specifically examined relational and graph databases and compared MySQL with Neo4j. The study motivated the use of graph databases by the difficulty of handling dynamic, recursive, and complex relationships through conventional relational structures. Its results favoured Neo4j for the selected relationship oriented queries and emphasized the flexibility of the graph model. At the same time, the study also considered other aspects of database systems, including maturity, security, and schema flexibility, illustrating that database selection involves more than execution time alone.

6.1 Positioning of the Present Study

The present thesis builds on the previous work described above but differs from it in several aspects. First, PostgreSQL and Neo4j are evaluated using the same real world Olist e commerce dataset. Particular attention is given to ensuring that both database implementations operate on the same cleaned subset of the original data. This allows equivalent analytical problems to be evaluated without differences in the underlying records affecting the comparison. Second, the experimental evaluation covers a wider progression of workloads than many of the previous studies. While [21] and [22] organize their comparisons primarily around selection, recursion, aggregation, and pattern matching, the present study begins with conventional descriptive queries and gradually moves toward increasingly relationship dependent analyses. These include customer similarity, recommendation scenarios, the construction and analysis of a seller network, connected components, clustering coefficients, PageRank, community detection, and multi-step traversal. In this way, the study examines not only isolated query categories but also how the characteristics of the two database models become more significant as the analytical task becomes increasingly graph oriented.
Another difference concerns the treatment of network analysis. In [23], graph analytical properties such as degree, connectedness, and centrality are expressed in both Cypher and SQL. The present study follows a similar comparative principle but extends it by constructing an explicit seller network from shared customer relationships and analysing the same network structure using both PostgreSQL and Neo4j. PostgreSQL implementations use relational techniques such as joins, views, aggregation, and recursive CTEs, whereas Neo4j can additionally make use of algorithms provided by the Graph Data Science library. This allows the comparison to include not only execution performance but also differences in implementation complexity and the availability of dedicated graph analytical methods.

The traversal experiment also provides a direct connection to earlier work. Traversal and recursive operations were already identified as important comparison areas in [20], [21], [22], and [24]. However, the present thesis evaluates traversal by fixing the same starting entity and the same permitted relationship types in both databases and systematically increasing the exact path length from one to seven. Both implementations produce the same number of reachable entities at each depth, allowing the resulting execution times to be compared directly. The experiment therefore provides insight into how the internal processing mechanisms of PostgreSQL and Neo4j respond as the traversal search space increases.

The results of the existing literature also show why such an experimental comparison remains relevant. Some studies, including [20], [21], and [22], report considerable advantages for Neo4j in relationship intensive and recursive queries, whereas [23] and [24] demonstrate that relational databases can perform equally well or better for particular analytical or complex workloads. These apparently different conclusions suggest that the question should not be reduced to which database model is generally faster. Instead, the suitability of relational and graph databases depends on the structure of the problem, the type of query, the available optimization mechanisms, and the extent to which the analysis relies on relationships and traversal.

Overall, the present study complements previous research by demonstrating that differences between relational and graph databases become more apparent in certain types of analytical tasks, particularly those involving complex relationships and network structures. The findings also highlight the importance of query design, indexing, and specialized algorithms in determining execution performance. Therefore, the results reinforce the conclusions of earlier studies that database suitability cannot be determined solely by the underlying data model, but must also be evaluated in relation to the specific analytical requirements and implementation strategies.

7. Comparative Analysis

This chapter presents a comparative analysis of the PostgreSQL and Neo4j implementations of the Olist e-commerce dataset. The aim is to examine how the relational and graph database models differ in terms of database construction, query formulation, execution, and their suitability for different types of analytical tasks.

7.1 Database Construction and Data Model Implementation

The first aspect of the comparative analysis concerns the construction of the two database implementations. As described in the previous chapters, the same Olist domain was represented in PostgreSQL using a relational schema and in Neo4j using a property graph model. The structures and contents of the individual relations, nodes, and relationships have already been presented in *Sections 4.3.2* and *5.3.1* ; therefore, they are not repeated here. Instead, the purpose of this section is to compare the construction processes themselves, with particular attention to the required data preparation, the order in which database elements had to be created, the mechanisms used to establish connections between entities, and the additional structures required to support efficient analysis.

7.1.1 Construction of Relational Database

The construction of the relational database was strongly influenced by the dependencies defined in the relational schema. As discussed in the *Section 4.3.2*, primary and foreign keys were used to identify records and establish relationships between tables. Consequently, the order in which the relations were loaded had to respect the referential dependencies defined by the schema.
The geolocation relation represented an important starting point in this process. As explained previously, the Olist geolocation dataset required preprocessing before the ZIP code prefix could serve as a primary key and be referenced by other relations. Once the geographical reference data had been prepared, the remaining relations were loaded according to their dependencies. The customers and sellers relations could be populated after geolocation, while products could be loaded independently. Orders were subsequently loaded using the existing customer records, followed by order_items, which depended on orders, products, and sellers. Payment and review records were also associated with existing orders. Thus, rather than following a completely linear sequence, the loading process was determined by the dependency structure defined through the foreign key relationships.
These dependencies also influenced the preparation of the original Olist data. Some records in the source datasets contained geographical identifiers for which no corresponding record existed in the processed geolocation data. Because foreign key constraints were used to enforce these connections, such references could not remain unresolved. Records that could not satisfy the required referential constraints therefore had to be identified and excluded from the relational database. Where these records were referenced by other parts of the dataset, their removal also had to be taken into account when preparing dependent data. Thus, data cleaning in the relational implementation involved not only processing individual CSV files but also preserving consistency across related tables.
This illustrates an important characteristic of relational database construction. The schema determines not only how entities are connected but also which records can be inserted while maintaining referential integrity. Primary keys, foreign keys, and other integrity constraints

therefore played an active role in the import process by preventing inconsistent or unresolved references from being included in the final database.

7.1.2 Construction of the Graph Database

The graph implementation followed a different construction process. As described in the *Section 5.3.1*, the Olist entities were first represented as nodes and their associations were subsequently created as explicit relationships. Unlike in PostgreSQL, the order of construction was governed less by referential integrity constraints and more by the practical requirement that the endpoint nodes exist before relationships could be established between them. Therefore, the principal entity nodes, including Customer, Order, Product, Seller, Geolocation, Payment, and Review nodes, were first imported from their respective source datasets. Once the required nodes were available, the basic relationships, such as PLACED, CONTAINS, LIVES_IN, LOCATED_IN, HAS_PAYMENT, and HAS_REVIEW, could be created by matching the corresponding identifiers or properties of their endpoint nodes.

A particularly important difference between the two implementations emerged when the source data contained unresolved geographical references. In PostgreSQL, records containing foreign key values for which no corresponding geolocation record existed could not be retained while preserving the defined referential constraints. Neo4j did not impose the same requirement. A Customer or Seller node could exist even when no corresponding Geolocation node was available; in such cases, the node could remain in the database without a LIVES_IN or LOCATED_IN relationship. This demonstrates greater structural flexibility during the initial graph import, but it also means that the existence of all expected connections is not automatically guaranteed and may therefore require additional validation.

Although Neo4j would technically have allowed these unmatched nodes to remain in the database, they were not retained in the final graph used for the comparative experiments. The records excluded from PostgreSQL during relational data preparation were identified, and the corresponding nodes were also removed from Neo4j. This was not required by the graph data model itself but was a methodological decision to ensure that both database systems worked with the same underlying dataset. Maintaining this consistency was essential for the validity of the comparison. If Neo4j had retained entities that were not present in PostgreSQL, differences in query results, graph topology, result cardinality, or execution time could have resulted from differences in the input data rather than from the database systems themselves. By applying the same exclusions to both implementations, the differences observed later can be more clearly attributed to the different data models and query processing approaches of the two database systems.

7.2 Basic Query Comparison

Before examining more complex analytical tasks, a set of basic queries was implemented to establish an initial comparison between the two database systems. *Tables 2–7* provide selected examples of the basic queries implemented in PostgreSQL and Neo4j. The selected queries cover common data retrieval and descriptive analytical operations, including filtering, aggregation, grouping, and combining information from multiple entities. Both systems can express these operations in a clear and straightforward way.

The purpose of these queries is to examine how the same analytical tasks can be formulated using SQL and Cypher. Although both implementations produce the same results, the way these results are obtained reflects the differences between the relational and graph data models.

The first example in *Table 2* counts the number of orders associated with each order status. In PostgreSQL, the records are grouped according to the status attribute of the orders relation and then counted. In Neo4j, the same operation is performed directly on Order nodes by grouping the returned values of the status property. This example mainly demonstrates that simple aggregations can be expressed in a very similar way in both systems.

How many orders are there for each status?	
PostgreSQL	**Neo4j**
SELECT status, COUNT(*) AS number_of_orders FROM orders GROUP BY status ORDER BY number_of_orders DESC;	MATCH (o:Order) RETURN o.status AS Status, COUNT(*) AS NumberOfOrders ORDER BY NumberOfOrders DESC;

Table 2: PostgreSQL and Neo4j queries for counting orders by status

The second example in *Table 3* determines how customers are distributed across states. In PostgreSQL, this requires joining the customers and geolocation relations through the ZIP code prefix before grouping the result by state. In Neo4j, the corresponding information is obtained by following the LIVES_IN relationship from Customer nodes to Geolocation nodes. This example already illustrates an important structural difference between the two approaches: PostgreSQL reconstructs the connection through a join condition, while Neo4j follows an explicitly stored relationship.

Which states have the most customers?	
PostgreSQL	**Neo4j**
SELECT state, COUNT(*) AS customers FROM geolocation g JOIN customers c ON g.zip_code_prefix = c.customer_zip_code_prefix GROUP BY state ORDER BY customers DESC;	MATCH (c:Customer)-[:LIVES_IN]->(g:Geolocation) RETURN g.state AS State, COUNT(c) AS customers ORDER BY customers DESC;

Table 3: PostgreSQL and Neo4j queries for identifying the states with the most customers

The third example in *Table 4* calculates the average shipping cost per order. Because an order may contain several order items, the freight values must first be summed for each order before the average is calculated. PostgreSQL performs this in a subquery that groups order_items by order_id, while Neo4j aggregates the freight_value property stored on CONTAINS relationships for each Order node. This example therefore shows how intermediate aggregation can be expressed in both models.

What is the average shipping cost per order?	
PostgreSQL	**Neo4j**

SELECT ROUND(AVG(shipping_cost), 2) AS avg_shipping_cost
FROM (
  SELECT order_id, SUM(freight_value) AS shipping_cost
  FROM order_items
  GROUP BY order_id) AS order_shipping;
	MATCH (o:Order)-[c:CONTAINS]->(:Product)
	WITH o, SUM(c.freight_value) AS ShippingCost
	RETURN ROUND(AVG(ShippingCost), 2) AS avgShippingCost;

Table 4: PostgreSQL and Neo4j queries for calculating the average shipping cost per order

The queries in *Table 5* determines how many products were sold by each seller. In PostgreSQL, sellers are joined with order_items using seller_id, after which the purchased products are counted for each seller. In Neo4j, the seller identifier stored on the CONTAINS relationship is used to associate each purchased item with the corresponding Seller node. The result is then grouped and ordered by the number of products sold.

Number of products sold by each seller	
PostgreSQL	**Neo4j**
SELECT s.seller_id, COUNT(oi.product_id) AS products_sold FROM sellers s JOIN order_items oi ON s.seller_id = oi.seller_id GROUP BY s.seller_id ORDER BY products_sold DESC;	MATCH (:Order)-[r:CONTAINS]->(:Product) MATCH (s:Seller {seller_id: r.seller_id}) RETURN s.seller_id AS seller_id, COUNT(r) AS productsSold ORDER BY productsSold DESC;

Table 5: PostgreSQL and Neo4j queries for determining the number of products sold by each seller

The average review score by product category requires information from several connected entities. As shown in *Table 6*, PostgreSQL follows the path from reviews to orders, order items, and products through a sequence of joins before calculating the average review score for each category. Neo4j expresses the same logical path as a graph pattern connecting Product, Order, and Review nodes. This example makes the structural difference between join based navigation and relationship based traversal particularly visible.

Average review score by category	
PostgreSQL	**Neo4j**
SELECT p.product_category_name, ROUND(AVG(r.score),2) AS avg_score FROM order_reviews r JOIN orders o ON r.order_id=o.order_id JOIN order_items oi ON o.order_id=oi.order_id JOIN products p ON oi.product_id=p.product_id GROUP BY p.product_category_name ORDER BY avg_score DESC;	MATCH (p:Product)<-[:CONTAINS]-(o:Order)-[:HAS_REVIEW]->(r:Review) RETURN p.category AS Category, ROUND(AVG(r.score), 2) AS avgScore ORDER BY avgScore DESC;

Table 6: PostgreSQL and Neo4j queries for calculating the average review score by product category

The delivery accuracy query compares the actual customer delivery date with the estimated delivery date. As shown in *Table 7*, PostgreSQL calculates the difference between the two timestamps, converts it from seconds into days, and then calculates the average. Neo4j performs the same calculation using the epochSeconds values of the corresponding datetime properties. A negative result indicates that deliveries arrived earlier than estimated on average, while a positive result indicates that they arrived later.

Delivery Accuracy	
PostgreSQL	**Neo4j**
SELECT ROUND(AVG(EXTRACT(EPOCH FROM (delivered_customer_date - estimated_delivery_date)) / 86400),2) AS avg_days_difference FROM orders o WHERE o.delivered_customer_date IS NOT NULL AND o.estimated_delivery_date IS NOT NULL;	MATCH (o:Order) WHERE o.delivered_customer_date IS NOT NULL AND o.estimated_delivery_date IS NOT NULL WITH datetime(o.delivered_customer_date).epochSeconds - datetime(o.estimated_delivery_date).epochSeconds AS difference_seconds RETURN ROUND(AVG(difference_seconds / 86400.0),2) AS avg_days_difference;

Table 7: PostgreSQL and Neo4j queries for calculating delivery accuracy based on the difference between actual and estimated delivery dates

In summary the basic queries show that both relational and graph databases are suitable for simple descriptive analysis, with relatively small differences for straightforward aggregations. The structural differences between the two approaches become more apparent when information from multiple entities must be combined. In PostgreSQL, related data is accessed by explicitly joining relations through their corresponding key values, whereas in Neo4j the same associations are expressed by matching patterns and following relationships between nodes. This difference is particularly visible in queries involving several connected entities, such as the calculation of average review scores by product category, where PostgreSQL requires a sequence of joins between the relevant relations, while Neo4j represents the connection as a graph pattern between Product, Order, and Review nodes. Nevertheless, the underlying analytical logic remains largely the same: the required records or nodes are identified, the relevant values are aggregated, and the results are grouped or ordered as necessary.

In terms of execution performance, both database systems handled basic analytical queries efficiently, as shown in *Table 8*. PostgreSQL generally achieved slightly lower execution times, with the largest difference observed in the number of products sold by each seller. However, all queries completed in less than one second in both systems, indicating no substantial performance differences for basic operations.

Basic Queries	PostgreSQL execution time	Neo4j execution time
How many orders are there for each status?	~91 ms	~87 ms
Which states have the most customers?	~142 ms	~126 ms
What is the average shipping cost per order?	~140 ms	~310 ms
Number of products sold by each seller	~174 ms	~975 ms
Average review score by category	~264 ms	~422 ms
Delivery Accuracy	~104 ms	~178 ms

Table 8: Execution time comparison of basic analytical queries in PostgreSQL and Neo4j

Overall, PostgreSQL is particularly natural for simple tabular aggregations, whereas Neo4j becomes more expressive as queries involve more interconnected data. This difference becomes more apparent in the more complex analyses presented in the following sections.

7.3 Advanced Query Comparison

Following the analysis of basic query operations, this section examines more advanced analytical problems in order to provide a more comprehensive evaluation of the relational and graph database approaches. The analysis is organized around several thematic areas, each of which is investigated through a set of queries implemented in both PostgreSQL and Neo4j. This approach enables the two database models to be evaluated not only in terms of query formulation and performance, but also with respect to their suitability for representing and analysing different types of relationships within the dataset. The selected problems are designed to highlight situations in which the structural characteristics of either the relational or graph model may provide particular advantages. The analysis covers the identification of similar customers based on common sellers and common categories, recommendation-oriented queries, the construction and analysis of seller networks, and multi-step database traversal.

7.3.1 Identification of Similar Customers

The first advanced analytical problem focuses on identifying customers with similar purchasing behaviour. In this analysis, similarity is examined from two perspectives: customers who purchased from the same sellers and customers who purchased products belonging to the same categories. This problem is particularly relevant for the comparison because customer similarity requires the analysis of relationships across multiple entities, while the two database models represent and access these relationships in fundamentally different ways.

Since multiple `customer_id` values may belong to the same physical customer in the Olist dataset, the comparison is performed using `customer_unique_id`. This ensures that purchasing behaviour is aggregated at the level of individual customers rather than separate customer records.

7.3.1.1 Similar Customers Based on Shared Sellers

In PostgreSQL, the first approach identifies pairs of customers who purchased from at least two common sellers. The query joins the `customers`, `orders`, and `order_items` tables for both customers and matches the two sides according to `seller_id`. Customer pairs are then grouped by their `customer_unique_id` values, while `COUNT(DISTINCT seller_id)` determines the number of sellers shared by each pair. The condition requiring at least two common sellers removes pairs for which the similarity is based on only a single seller. *Listing 1* shows the PostgreSQL query used to identify pairs of customers who purchased from at least two common sellers.

```
SELECT
   c1.customer_unique_id AS customer1,
   c2.customer_unique_id AS customer2,
   COUNT(DISTINCT oi1.seller_id) AS common_sellers

FROM customers c1
JOIN orders o1
   ON c1.customer_id = o1.customer_id
JOIN order_items oi1
   ON o1.order_id = oi1.order_id

JOIN customers c2
   ON c1.customer_unique_id < c2.customer_unique_id
JOIN orders o2
   ON c2.customer_id = o2.customer_id
JOIN order_items oi2
   ON o2.order_id = oi2.order_id
   AND oi1.seller_id = oi2.seller_id

GROUP BY c1.customer_unique_id, c2.customer_unique_id
HAVING COUNT(DISTINCT oi1.seller_id) >= 2
ORDER BY common_sellers DESC;
```

Listing 1: PostgreSQL query for identifying customer pairs with at least two common sellers

Although the number of common sellers provides a straightforward measure of similarity, it does not consider the total number of sellers associated with each customer. For this reason, a second query applies the *Jaccard similarity coefficient* [27]. The query was implemented using several Common Table Expressions (CTEs), which divide the calculation into separate logical stages. This structure makes it possible to construct the intermediate datasets required for the similarity calculation step by step and avoids repeating the same joins throughout the query. The first CTE, customer_sellers, creates a distinct set of customer and seller combinations. The second CTE, customer_degrees, calculates the total number of distinct sellers associated with each customer. The third CTE, common_sellers, compares the customer seller sets and determines the number of sellers shared by each customer pair. The condition cs1.customer_unique_id < cs2.customer_unique_id ensures that each pair is generated only once and prevents customers from being compared with themselves. These intermediate results are then combined in the final part of the query to calculate the Jaccard similarity coefficient. The similarity between two customers is defined as

$$J(A,B) = \frac{|A \cap B|}{|AUB|} = \frac{|A \cap B|}{|A|+|B|-|A \cap B|},$$

where A and B represent the sets of sellers from which the two customers purchased. Unlike the absolute number of common sellers, this measure takes the size of both customers' seller sets into account. Consequently, two customers who share two out of two sellers receive a higher similarity score than two customers who share two sellers but have purchased from many additional sellers. Only customer pairs with at least two common sellers are retained. In PostgreSQL, the individual components of the similarity calculation must therefore be explicitly constructed from the

underlying relations. *Listing 2* presents the PostgreSQL implementation of Jaccard similarity based on shared sellers.

```
WITH customer_sellers AS (
   SELECT DISTINCT c.customer_unique_id, oi.seller_id
   FROM customers c
   JOIN orders o ON c.customer_id = o.customer_id
   JOIN order_items oi ON o.order_id = oi.order_id),
customer_degrees AS (
   SELECT customer_unique_id, COUNT(*) AS seller_count
   FROM customer_sellers
   GROUP BY customer_unique_id),
common_sellers AS (
   SELECT
      cs1.customer_unique_id AS customer1,
      cs2.customer_unique_id AS customer2,
      COUNT(*) AS common_sellers
   FROM customer_sellers cs1
   JOIN customer_sellers cs2 ON cs1.seller_id = cs2.seller_id
     AND cs1.customer_unique_id < cs2.customer_unique_id
   GROUP BY cs1.customer_unique_id, cs2.customer_unique_id
)
SELECT
   cs.customer1,
   cs.customer2,
   cs.common_sellers,
   cd1.seller_count AS customer1_sellers,
   cd2.seller_count AS customer2_sellers,
   cs.common_sellers::numeric
   /
   ( cd1.seller_count + cd2.seller_count - cs.common_sellers ) AS similarity
FROM common_sellers cs
JOIN customer_degrees cd1 ON cs.customer1 = cd1.customer_unique_id
JOIN customer_degrees cd2 ON cs.customer2 = cd2.customer_unique_id
WHERE cs.common_sellers >= 2
ORDER BY similarity DESC, common_sellers DESC LIMIT 100;
```

Listing 2: PostgreSQL implementation of Jaccard similarity based on shared sellers

The Neo4j implementation represents the same problem differently. Because the original Customer nodes correspond to customer_id, separate CustomerUnique nodes were introduced to represent physical customers. These nodes were connected to Seller nodes through BOUGHT_FROM relationships. The use of MERGE ensures that a customer and seller are connected by only one such relationship even when the customer purchased from the same seller multiple times. This results in a graph structure in which the neighbourhood of a CustomerUnique node directly represents the set of sellers associated with that customer.

An important result of the experiment was that a direct Cypher query for determining customer pairs with common sellers exceeded the available memory. The pattern CustomerUnique–Seller–CustomerUnique can generate a very large number of intermediate customer pairs, particularly for sellers connected to many customers. Furthermore, the same customer pair may be encountered through several sellers. These intermediate results must be maintained until aggregation

determines the number of distinct common sellers, while the filtering condition can only be applied after this aggregation. Consequently, the otherwise intuitive graph pattern does not necessarily result in an efficient execution strategy. This observation is significant for the comparison because it demonstrates that the graph model should not automatically be considered more efficient for every relationship oriented query. The corresponding PostgreSQL operation is also computationally demanding, requiring approximately 1.5 minutes to complete. However, the relational database can rely on alternative join strategies and, where necessary, temporary disk operations during aggregation and sorting. Therefore, PostgreSQL was able to complete this particular type of operation in situations where the direct Cypher implementation exceeded the available transaction memory.

The advantage of the Neo4j approach becomes more apparent when the Graph Data Science library is used instead of general Cypher pattern matching. A projected `customerUniqueSellerGraph` was created, consisting of `CustomerUnique` and `Seller` nodes connected through `BOUGHT_FROM` relationships. Within this bipartite graph, the neighbourhood of each `CustomerUnique` node corresponds to the set of sellers from which the customer has made purchases. The Node Similarity algorithm was then applied using Jaccard similarity. Unlike the direct Cypher query, the algorithm operates on an in-memory graph representation and is specifically optimized for comparing node neighbourhoods. It can therefore calculate customer similarity without generating the same large intermediate result as the direct Cypher query.

The obtained similarity results were consistent with the relational analysis, demonstrating that the same conceptual similarity measure can be implemented in both systems, although through substantially different mechanisms.

Listing 3 presents the Neo4j implementation of customer similarity based on shared sellers using the Graph Data Science Node Similarity algorithm.

```
CALL gds.nodeSimilarity.stream('customerUniqueSellerGraph', {
similarityMetric: 'JACCARD'})
YIELD node1, node2, similarity

WITH
gds.util.asNode(node1) AS cu1,
gds.util.asNode(node2) AS cu2,
similarity

WHERE cu1:CustomerUnique
AND cu2:CustomerUnique
AND cu1.customer_unique_id < cu2.customer_unique_id

MATCH (cu1)-[:BOUGHT_FROM]->(s:Seller)<-[:BOUGHT_FROM]-(cu2)
WITH cu1, cu2, similarity, count(DISTINCT s) AS common_sellers
WHERE common_sellers >= 2

RETURN
cu1.customer_unique_id AS customer1,
cu2.customer_unique_id AS customer2,
common_sellers,
similarity
ORDER BY similarity DESC, common_sellers DESC LIMIT 100;
```

Listing 3: Neo4j Node Similarity query for customer similarity based on shared sellers

The execution times of the customer similarity queries based on shared sellers are summarized in *Table 9*. The results show that PostgreSQL completed the initial query in approximately 1.5 minutes, while the direct Cypher implementation in Neo4j exceeded the available memory. For the Jaccard similarity calculation, both systems successfully completed the analysis, with Neo4j achieving a slightly shorter execution time using the Graph Data Science library.

Experiment	PostgreSQL execution time	Neo4j execution time
Similar Customers Based on Shared Sellers	~1.5 min	not completed (out of memory)
Similar Customers Based on Shared Sellers with Jaccard similarity	~3 min 20 s	~2 min 24 s

Table 9: Execution time comparison of customer similarity queries based on shared sellers in PostgreSQL and Neo4j

7.3.1.2 Similar Customers Based on Common Categories

The second part of the analysis considers common product categories instead of common sellers. In PostgreSQL, the initial query followed a direct approach by joining the purchasing histories of two customers and matching the product categories they had purchased. However, this query ran for more than 1.5 hours without producing a final result. The main problem was the generation of a very large number of potential customer pairs before it could be determined whether the customers actually shared any product categories. The subsequent joins with orders, order items, and products further increased the size of the intermediate result, making the operation computationally expensive. *Listing 4* presents the initial PostgreSQL query for identifying customer pairs based on common product categories.

```
SELECT c1.customer_unique_id AS customer1,
       c2.customer_unique_id AS customer2,
       COUNT(DISTINCT p.product_category_name) AS common_categories
FROM customers c1
JOIN orders o1
  ON c1.customer_id = o1.customer_id
JOIN order_items oi1
  ON o1.order_id = oi1.order_id
JOIN products p
  ON oi1.product_id = p.product_id

JOIN customers c2
  ON c1.customer_unique_id < c2.customer_unique_id
JOIN orders o2
  ON c2.customer_id = o2.customer_id
JOIN order_items oi2
  ON o2.order_id = oi2.order_id
JOIN products p2
  ON oi2.product_id = p2.product_id
  AND p.product_category_name = p2.product_category_name
GROUP BY c1.customer_unique_id, c2.customer_unique_id
HAVING COUNT(DISTINCT p.product_category_name) >= 2
ORDER BY common_categories DESC;
```

Listing 4: Initial unoptimized PostgreSQL query for customer similarity detection based on common product categories

To address this limitation, the PostgreSQL query was redesigned to reduce the number of unnecessary customer comparisons. First, the distinct product categories purchased by each unique customer were extracted, transforming the transactional data into a customer–category representation. This preprocessing step ensured that repeated purchases within the same category did not influence the similarity calculation. Rather than directly comparing every customer with every other customer, the optimized approach generated all unique category pairs associated with each customer. These category pairs were then used to identify candidate customer pairs. Two customers became candidates only if they shared the same category pair, implying that they had already purchased products from at least two common categories. As a result, customer pairs without sufficient category overlap were excluded before the expensive comparison stage, significantly reducing the search space. Once the candidate customer pairs had been identified, the number of common categories was derived from the number of shared category pairs. If two customers share k categories, those categories form $k \cdot (k - 1) / 2$ distinct category pairs. Therefore, after the matching category pairs have been counted, the number of common categories can be recovered using the inverse expression $(1 + \sqrt{1 + 8p}) / 2$, where p is the number of shared category pairs. The total number of distinct categories purchased by each customer was then used together with the number of common categories to calculate the Jaccard similarity coefficient.
Listing 5 presents the optimized PostgreSQL query for identifying similar customers based on shared product categories.

```
WITH customer_categories AS (
   SELECT DISTINCT
      c.customer_unique_id,
      p.product_category_name AS category
   FROM customers c
   JOIN orders o ON c.customer_id = o.customer_id
   JOIN order_items oi ON o.order_id = oi.order_id
   JOIN products p ON oi.product_id = p.product_id
   WHERE p.product_category_name IS NOT NULL ),
customer_category_counts AS (
   SELECT  customer_unique_id,  COUNT(*) AS total_categories
   FROM customer_categories
   GROUP BY customer_unique_id ),
customer_category_pairs AS (
   SELECT cc1.customer_unique_id, cc1.category AS category1, cc2.category AS category2
   FROM customer_categories cc1
   JOIN customer_categories cc2 ON cc1.customer_unique_id = cc2.customer_unique_id
     AND cc1.category < cc2.category ),
similar_pairs AS (
   SELECT
      cp1.customer_unique_id AS customer1,
      cp2.customer_unique_id AS customer2,
      ( (1 + SQRT(1 + 8 * COUNT(*)::numeric)) / 2)::integer AS common_categories
   FROM customer_category_pairs cp1
   JOIN customer_category_pairs cp2 ON cp1.category1 = cp2.category1
     AND cp1.category2 = cp2.category2
     AND cp1.customer_unique_id < cp2.customer_unique_id
   GROUP BY cp1.customer_unique_id, cp2.customer_unique_id )
SELECT
   sp.customer1, sp.customer2, sp.common_categories,
   c1.total_categories AS customer1_categories,
   c2.total_categories AS customer2_categories,
   ROUND( sp.common_categories::numeric /
      ( c1.total_categories+ c2.total_categories - sp.common_categories),4) AS jaccard_similarity
FROM similar_pairs sp
JOIN customer_category_counts c1 ON sp.customer1 = c1.customer_unique_id
JOIN customer_category_counts c2 ON sp.customer2 = c2.customer_unique_id
ORDER BY jaccard_similarity DESC, common_categories DESC;
```

Listing 5: Optimized PostgreSQL query for identifying similar customers based on shared product categories and Jaccard similarity

This reformulation produced a significant performance improvement. While the initial direct customer-pair comparison failed to complete after more than 1.5 hours, the optimized query executed in approximately 2 seconds. The improvement resulted primarily from generating candidate pairs only after identifying shared category combinations, thereby avoiding the creation of a large number of irrelevant customer pairs. This example demonstrates how query formulation and the reduction of intermediate result sets can substantially influence the computational cost of similarity analysis in relational databases.

In Neo4j, the category based problem was represented explicitly in the graph. Separate Category nodes were created from the category property of Product nodes and connected to products through IN_CATEGORY relationships. CustomerUnique nodes were then connected to the categories from which they had purchased through BOUGHT_CATEGORY relationships. As with the seller representation, repeated purchases from the same category result in only one relationship between a particular customer and category.

A projected graph named customerCategoryGraph was subsequently created, consisting of CustomerUnique and Category nodes linked by BOUGHT_CATEGORY relationships. Within this bipartite graph, the neighbourhood of each CustomerUnique node corresponds to the set of product categories associated with that customer. Jaccard Node Similarity was applied to the customer–category network. This allows customer similarity based on categories to be analysed using the same general graph algorithm as similarity based on sellers; only the type of neighbourhood being compared changes. Additional parameters such as *topK* and *similarityCutoff* can also be used to restrict the result to the most relevant similar customers and to eliminate weak similarities.

Listing 6 presents the Neo4j implementation of customer similarity based on shared product categories using the Graph Data Science Node Similarity algorithm.

```
CALL gds.nodeSimilarity.stream( 'customerCategoryGraph',
 {    similarityMetric: 'JACCARD',
      topK: 10,
      similarityCutoff: 0.2})
YIELD node1, node2, similarity
WITH
   gds.util.asNode(node1) AS cu1, gds.util.asNode(node2) AS cu2, similarity
WHERE cu1:CustomerUnique
  AND cu2:CustomerUnique
  AND cu1.customer_unique_id < cu2.customer_unique_id

MATCH (cu1)-[:BOUGHT_CATEGORY]->(cat:Category) <-[:BOUGHT_CATEGORY]-(cu2)
WITH cu1, cu2, similarity,
   collect(DISTINCT cat.name) AS common_category_names
WHERE size(common_category_names) >= 2
RETURN
   cu1.customer_unique_id AS customer1,
   cu2.customer_unique_id AS customer2,
   size(common_category_names) AS common_categories,
   common_category_names,
   similarity AS jaccard_similarity
ORDER BY jaccard_similarity DESC, common_categories DESC LIMIT 100;
```

Listing 6: Neo4j Node Similarity query for customer similarity based on shared product categories

Table 10 summarizes the execution times of the customer similarity queries based on shared product categories. The results highlight the significant improvement achieved through query optimization in PostgreSQL, reducing execution time from over 1.5 hours without completion to approximately two seconds. The optimized PostgreSQL query also performed considerably faster than Neo4j, which completed the Jaccard similarity analysis in 2 minutes and 14 seconds.

Experiment	PostgrSQL execution time	Neo4j execution time
Similar Customers Based on Common Categories	initial query did not complete	-
Similar Customers Based on Common Categories with Jaccard similarity	optimized version: ~1 s 9 ms	~2 min 14 s

Table 10: Execution time comparison of customer similarity queries based on shared product categories in PostgreSQL and Neo4j

In summary, the similarity analysis demonstrated how relational and graph databases approach relationship oriented analytical tasks in different ways. In PostgreSQL, customer similarity was calculated through joins, aggregations, and explicitly constructed intermediate results. Although the initial category based query was inefficient because it generated a very large number of customer pairs, an optimized strategy based on shared category pairs significantly reduced the search space and made the analysis practical. In Neo4j, similarity was analysed through the graph structure by comparing the neighbourhoods of CustomerUnique nodes. Direct Cypher implementations were not always practical because of their memory requirements, whereas the Graph Data Science Node Similarity algorithm provided a more suitable method for this type of analysis. Overall, the results indicate that performance depends not only on the database model itself, but also on the design of the query and the availability of specialized processing methods.

7.3.2 Recommendation-oriented Queries

Following the identification of similar customers, the next analysis examines how customer similarity can be used to generate category and product recommendations. Two recommendation scenarios were implemented in both databases. The first generates category recommendations for all customers by identifying customers with at least two product categories in common and recommending categories purchased by similar customers but not by the target customer. The second is a more targeted recommendation task: for a specific customer, the twenty customers with the highest category overlap are selected, after which products purchased by these similar customers but not by the target customer are ranked.

7.3.2.1 Category recommendation for all customers

The first experiment reuses the approach introduced in the previous analysis of similarity based on common categories. In PostgreSQL, the query begins in the same way: the distinct product categories purchased by each customer are extracted through joins between customers, orders, order_items, and products. This removes repeated purchases from the same category and produces a compact customer–category representation. Unique category pairs are then generated for each customer, and only customers sharing the same category pair are considered for further comparison. This step reduces the search space by excluding customer pairs that do not have sufficient category overlap before the recommendation process begins. Because each unordered customer pair is initially generated only once, the similar_customers step represents it in both directions so that either customer can act as the recommendation target. The final recommendation stage then proposes categories purchased by similar customers but not by the target customer. The recommendations are ranked according to the number of supporting similar customers and the strongest category overlap observed among those customers. *Listing 7* presents the PostgreSQL query used to generate category recommendations based on similar customers.

```
WITH customer_categories AS (
   SELECT DISTINCT c.customer_unique_id, p.product_category_name AS category
   FROM customers c
   JOIN orders o
      ON c.customer_id = o.customer_id
   JOIN order_items oi
      ON o.order_id = oi.order_id
   JOIN products p
      ON oi.product_id = p.product_id
   WHERE p.product_category_name IS NOT NULL),
customer_category_pairs AS (
   SELECT cc1.customer_unique_id, cc1.category AS category1, cc2.category AS category2
   FROM customer_categories cc1
   JOIN customer_categories cc2
      ON cc1.customer_unique_id = cc2.customer_unique_id
     AND cc1.category < cc2.category),
similar_pairs AS (
   SELECT
      cp1.customer_unique_id AS customer1,
      cp2.customer_unique_id AS customer2,
      ( (1 + SQRT(1 + 8 * COUNT(*)::numeric)) / 2)::integer AS common_categories
   FROM customer_category_pairs cp1
   JOIN customer_category_pairs cp2
      ON cp1.category1 = cp2.category1
     AND cp1.category2 = cp2.category2
     AND cp1.customer_unique_id < cp2.customer_unique_id
   GROUP BY cp1.customer_unique_id, cp2.customer_unique_id),
similar_customers AS (
   SELECT customer1 AS target_customer, customer2 AS similar_customer, common_categories
   FROM similar_pairs
   UNION ALL
   SELECT customer2 AS target_customer, customer1 AS similar_customer, common_categories
   FROM similar_pairs),
recommendations AS (
   SELECT sc.target_customer, cc.category AS recommended_category,
      COUNT(DISTINCT sc.similar_customer) AS supporting_customers,
      MAX(sc.common_categories) AS max_common_categories
   FROM similar_customers sc
   JOIN customer_categories cc
      ON sc.similar_customer = cc.customer_unique_id
   WHERE NOT EXISTS (
      SELECT 1 FROM customer_categories own
      WHERE own.customer_unique_id = sc.target_customer AND own.category = cc.category)
   GROUP BY sc.target_customer, cc.category)
SELECT
   target_customer, recommended_category, supporting_customers, max_common_categories
FROM recommendations
ORDER BY target_customer, supporting_customers DESC, max_common_categories DESC;
```

Listing 7: PostgreSQL query for generating category recommendations based on similar customers

Neo4j follows a similar strategy to PostgreSQL when identifying customers with common product categories. First, two categories purchased by the same CustomerUnique are selected. Another customer is considered similar only if they are connected to both of these categories. In this way, customers with fewer than two common categories are excluded at an early stage, which reduces the number of customer pairs that need to be processed. Unlike PostgreSQL, Neo4j can identify these connections directly through the BOUGHT_CATEGORY relationships stored in the graph, without reconstructing them through joins. After the possible customer pairs have been identified, the query counts their total number of common categories. Each pair is then considered in both directions using UNWIND, so either customer can act as the target customer. Finally, categories purchased by the similar customer but not by the target customer are selected and ranked according to the number of supporting customers and the maximum number of common categories.
Listing 8 presents the corresponding Cypher query. Both implementations produced the same recommendation results, although they used different ways to access and process the relationships between customers and categories.

```
MATCH (c1:CustomerUnique)-[:BOUGHT_CATEGORY]->(cat1:Category)
MATCH (c1)-[:BOUGHT_CATEGORY]->(cat2:Category)
WHERE cat1.name < cat2.name

MATCH (c2:CustomerUnique)-[:BOUGHT_CATEGORY]->(cat1)
MATCH (c2)-[:BOUGHT_CATEGORY]->(cat2)
WHERE c1.customer_unique_id < c2.customer_unique_id
WITH DISTINCT c1, c2

MATCH (c1)-[:BOUGHT_CATEGORY]->(common:Category)
    <-[:BOUGHT_CATEGORY]-(c2)

WITH c1, c2, count(DISTINCT common) AS common_categories
WHERE common_categories >= 2

UNWIND [[c1, c2], [c2, c1]] AS pair

WITH pair[0] AS target, pair[1] AS similar, common_categories

MATCH (similar)-[:BOUGHT_CATEGORY]->(recommended:Category)
WHERE NOT EXISTS {
  MATCH (target)-[:BOUGHT_CATEGORY]->(recommended)}

RETURN
  target.customer_unique_id AS target_customer,
  recommended.name AS recommended_category,
  count(DISTINCT similar) AS supporting_customers,
  max(common_categories) AS max_common_categories

ORDER BY target_customer, supporting_customers DESC,  max_common_categories DESC;
```

Listing 8: Neo4j Cypher query for generating category recommendations based on similar customers

7.3.2.2 Product recommendation for a specific costumer

The second experiment follows the same idea of finding similar customers based on common product categories, but it focuses on one specific customer. Since the target customer is already known, PostgreSQL does not need to compare all customers with each other. In *Listing 9* the query first identifies the categories and products previously purchased by the target customer. It then finds other customers who purchased products from the same categories and counts how many categories they have in common with the target customer. Customers with fewer than two common categories are excluded, and the 20 customers with the highest overlap are selected. The products purchased by these customers are then used to generate recommendations, while products already purchased by the target customer are removed from the result. This approach reduces the amount of data that needs to be processed compared with an analysis involving all possible customer pairs.

```
WITH target_categories AS (
   SELECT DISTINCT p.product_category_name AS category
   FROM customers c
   JOIN orders o ON c.customer_id = o.customer_id
   JOIN order_items oi ON o.order_id = oi.order_id
   JOIN products p ON oi.product_id = p.product_id
   WHERE c.customer_unique_id = '1b6c7548a2a1f9037c1fd3ddfed95f33'
   AND p.product_category_name IS NOT NULL),
target_products AS (
   SELECT DISTINCT oi.product_id
   FROM customers c
   JOIN orders o ON c.customer_id = o.customer_id
   JOIN order_items oi ON o.order_id = oi.order_id
   WHERE c.customer_unique_id ='1b6c7548a2a1f9037c1fd3ddfed95f33'),
similar_customers AS (
   SELECT c.customer_unique_id AS similar_customer,
          COUNT(DISTINCT p.product_category_name) AS common_categories
   FROM customers c
   JOIN orders o ON c.customer_id = o.customer_id
   JOIN order_items oi ON o.order_id = oi.order_id
   JOIN products p ON oi.product_id = p.product_id
   JOIN target_categories tc ON p.product_category_name = tc.category
   WHERE c.customer_unique_id <> '1b6c7548a2a1f9037c1fd3ddfed95f33'
   GROUP BY  c.customer_unique_id
   HAVING COUNT(DISTINCT p.product_category_name) >= 2
   ORDER BY common_categories DESC, c.customer_unique_id ASC LIMIT 20),
recommendations AS (
   SELECT oi.product_id AS recommended_product,
          p.product_category_name AS category,
          COUNT(DISTINCT sc.similar_customer) AS supporting_customers,
          MAX(sc.common_categories) AS max_common_categories
   FROM similar_customers sc
   JOIN customers c ON c.customer_unique_id = sc.similar_customer
   JOIN orders o ON c.customer_id = o.customer_id
   JOIN order_items oi ON o.order_id = oi.order_id
   JOIN products p ON oi.product_id = p.product_id
   WHERE NOT EXISTS ( SELECT 1 FROM target_products tp WHERE tp.product_id = oi.product_id )
   GROUP BY oi.product_id, p.product_category_name)
SELECT recommended_product, category, supporting_customers, max_common_categories
FROM recommendations
ORDER BY supporting_customers DESC, max_common_categories DESC,
          recommended_product ASC LIMIT 10;
```

Listing 9: PostgreSQL product recommendations for a specific customer

Neo4j performs the same task directly through the graph structure. In *Listing 10* the query starts from the selected CustomerUnique node and follows BOUGHT_CATEGORY relationships to the categories purchased by that customer. It then finds other customers connected to the same categories and counts how many categories they have in common with the target customer. Only customers with at least two common categories are kept, and the 20 most similar customers are selected. The query then follows the BELONGS_TO, PLACED, and CONTAINS relationships to identify the products purchased by these customers. Products already purchased by the target customer are

excluded. The remaining products are ranked based on how many similar customers purchased them and on the highest number of common categories among those customers. Finally, the ten highest ranked products are returned as recommendations. *Listing 10* presents the corresponding Cypher query, while *Figure 8* shows the first ten rows of the query results.

```
MATCH (target:CustomerUnique {
   customer_unique_id: "1b6c7548a2a1f9037c1fd3ddfed95f33"})

MATCH (target)-[:BOUGHT_CATEGORY]->(cat:Category)<-[:BOUGHT_CATEGORY]-
(other:CustomerUnique)
WHERE other <> target
WITH target, other, COUNT(DISTINCT cat) AS common_categories
WHERE common_categories >= 2
ORDER BY common_categories DESC, other.customer_unique_id ASC LIMIT 20

MATCH (other)<-[:BELONGS_TO]-(c:Customer) -[:PLACED]->(:Order) -
[:CONTAINS]->(recommended:Product)
WHERE NOT EXISTS {
   MATCH (target)<-[:BELONGS_TO]-(:Customer) -[:PLACED]->(:Order)
      -[:CONTAINS]->(recommended)}

RETURN
   recommended.product_id AS recommended_product,
   recommended.category AS category,
   COUNT(DISTINCT other) AS supporting_customers,
   MAX(common_categories) AS max_common_categories

ORDER BY
   supporting_customers DESC,
   max_common_categories DESC,
   recommended_product ASC LIMIT 10;
```

Listing 10: Neo4j product recommendations for a specific customer

	recommended_product character (32)	category character varying (100)	supporting_customers bigint	max_common_categories bigint
1	9e0e152552a1323f7e5dcf63d50cda...	perfumaria	1	3
2	f0bc0aed23d451f89495f7650d81e2...	beleza_saude	1	3
3	f71050e65776861c8c3ec7a2d2b019...	utilidades_domesticas	1	3
4	0b61b74bd2917877a6fc5d31192d3...	moveis_decoracao	1	2
5	0d9374142ba86a7c22357ea192bcfd...	utilidades_domesticas	1	2
6	145140d23f166a1166dc4437107df8...	papelaria	1	2
7	166b38c4ecd6765dad14586d0a8f70...	papelaria	1	2
8	2835bf7dc6f51f7dcd51d2ca3ab60ce9	moveis_decoracao	1	2
9	288dbc06591742c5d7fe90fbfb0894...	moveis_decoracao	1	2
10	2948658cb6abc82847412be7201bfc...	beleza_saude	1	2

Figure 8: Top 10 product recommendations for the selected customer in PostgreSQL

7.3.2.2.1 The Execution Plan

To better understand the execution times observed for the product recommendation query for a specific customer, the execution plans of the PostgreSQL and Neo4j implementations were also examined. This analysis helps identify which operations account for most of the processing time and how the two database systems access and combine the data required to generate the recommendations.

In Neo4j, an index on CustomerUnique.customer_unique_id was already available before this experiment because a uniqueness constraint had been created as part of the graph model. This constraint ensures that two CustomerUnique nodes cannot have the same identifier and also provides efficient lookup when a specific customer is used as the starting point of a query. The recommendation query therefore ran under already indexed conditions and completed in approximately 100 ms.
In PostgreSQL, however, there was initially no separate index on customers.customer_unique_id. The original query still completed relatively quickly, in about 1.87 seconds, but this meant that the two implementations were not being compared under equally optimized conditions. Since Neo4j already used an index for locating the target customer, a corresponding index was also created in PostgreSQL to provide a more balanced performance comparison. After the index was added, PostgreSQL used Index Scan operations for these lookups instead of relying on sequential and parallel sequential scans in the same parts of the query. As a result, the execution time decreased from approximately 1.87 seconds to 627 ms, representing an improvement of almost three times.
The indexed PostgreSQL execution plan also provides a clearer view of where the remaining execution time is spent. The most time consuming part is the identification of similar customers. In the optimized execution plan, approximately 550 ms had already been used by the time the top 20 customers were selected, while the complete query finished in about 627 ms. This shows that most of the execution time is spent on generating, comparing, and aggregating the similarity candidates. Once the candidate set has been reduced to 20 customers, the remaining recommendation steps operate on a much smaller amount of data. This part of the query therefore accounts for only a relatively small portion of the total execution time.
The execution plan also shows a large number of repeated indexed lookups during the similarity search. In particular, 30,804 index searches were performed using the primary key indexes of both the orders and customers tables. Although these individual lookups are efficient, their large number contributes to the overall execution cost. In Neo4j, the same logical connections are represented directly through relationships such as BOUGHT_CATEGORY, BELONGS_TO, PLACED, and CONTAINS, allowing the query to follow the existing graph structure rather than reconstructing these connections through relational joins.

Table 11 summarizes the execution times of the two recommendation experiments in PostgreSQL and Neo4j. Both systems completed the queries efficiently, with Neo4j achieving slightly faster execution for category recommendations across all customers and a more noticeable advantage for product recommendations targeting a specific customer.

Experiment	PostgrSQL execution time	Neo4j execution time
Category recommendation for all customers	~2 s 145 ms	~1 s 425 ms
Product recommendation for a specific costumer	~627 ms	~100 ms

Table 11: Execution time comparison of category and product recommendation queries in PostgreSQL and Neo4j

Overall, the recommendation experiments show that relational and graph databases can implement the same recommendation logic while using different methods to access and process related data. In both systems, customer similarity is based on shared product categories, after which categories or products not previously purchased by the target customer are recommended. PostgreSQL performs these tasks through joins, aggregations, and CTEs, while Neo4j follows relationships already represented in the graph. Both implementations produced equivalent recommendation results, but the graph representation provides a more direct way to express relationship based recommendation tasks. The performance analysis of the targeted product recommendation also showed that indexing has an important effect on execution time and should be considered when comparing the two systems.

7.3.3 Seller Network Based on Shared Customers

The next area of comparison focuses on seller network analysis, where relationships between sellers are identified through their shared customers. This part of the analysis first examines the construction of the seller network and then focuses on analysing the resulting network structure. Rather than examining sellers only as individual entities, the analysis considers the structure formed by these connections. In both implementations, two sellers are considered connected when they share at least two physical customers, identified by customer_unique_id. In PostgreSQL, this network is represented by the seller_network table, whereas in Neo4j the same concept is represented directly as a SHARED_CUSTOMERS relationship between Seller nodes.

7.3.3.1 Construction of the Seller Network

In PostgreSQL, the seller network first has to be derived from the original relational schema. The customers, orders, and order_items tables are joined to compose distinct customer_unique_id–seller_id combinations. The result is then self-joined by customer so that seller pairs with the same customers can be identified. Each seller pair is stored only once, and the common_customers column represents the number of customers shared by the two sellers. A minimum threshold of two common customers is used to determine whether an edge should exist. After the seller pairs have been generated, the resulting seller_network table is conceptually an edge list. In other words, although the information describes a graph, PostgreSQL still stores it as rows consisting of seller_1, seller_2, and common_customers. The network interpretation exists primarily in the query logic rather than in the data model itself. *Listing 11* presents the PostgreSQL construction of the seller network.

```
CREATE TABLE seller_network AS
WITH customer_sellers AS (
   SELECT DISTINCT
      c.customer_unique_id,
      oi.seller_id
   FROM customers c
   JOIN orders o
      ON c.customer_id = o.customer_id
   JOIN order_items oi
      ON o.order_id = oi.order_id
)
SELECT
   cs1.seller_id AS seller_1,
   cs2.seller_id AS seller_2,
   COUNT(*) AS common_customers
FROM customer_sellers cs1
JOIN customer_sellers cs2
   ON cs1.customer_unique_id = cs2.customer_unique_id
  AND cs1.seller_id < cs2.seller_id
GROUP BY  cs1.seller_id, cs2.seller_id
HAVING COUNT(*) >= 2;
```

Listing 11: Construction of the seller_network table in PostgreSQL

The Neo4j solution is conceptually more direct. Sellers already exist as nodes, and the result of the shared customer calculation is represented directly as SHARED_CUSTOMERS relationship. The number of shared customers becomes a property of that relationship. Thus, the network representation corresponds directly to the analytical concept: an entity is a node, an association is a relationship, and the strength of that association is a relationship property. As in PostgreSQL, a SHARED_CUSTOMERS relationship is created only when two sellers share at least two customers. *Listing 12* shows the Neo4j construction of the seller network.

```
MATCH (cu:CustomerUnique)-[:BOUGHT_FROM]->(s1:Seller)
MATCH (cu)-[:BOUGHT_FROM]->(s2:Seller)

WHERE s1.seller_id < s2.seller_id

WITH   s1, s2, COUNT(DISTINCT cu) AS common_customers
WHERE common_customers >= 2

MERGE (s1)-[r:SHARED_CUSTOMERS]->(s2)
SET r.shared_customers = common_customers;
```

Listing 12: Construction of the Seller Network in Neo4j

7.3.3.2 Analysis of the Seller Network

Once the seller network had been constructed in PostgreSQL, the seller_network table was used as the basis for further network analysis. Since each seller pair was stored only once, an additional seller_edges view was created in which every connection appeared in both directions. This was necessary because the shared customer relationship is naturally undirected: if Seller A and Seller B have common customers, the same relationship also exists from the perspective of Seller B. The

relational representation therefore required an explicit transformation before the network could conveniently be traversed in both directions. The corresponding SQL query used to create the seller_edges view is shown in *Listing 13*.

```
CREATE OR REPLACE VIEW seller_edges AS
SELECT
   seller_1 AS seller_from,
   seller_2 AS seller_to,
   common_customers AS weight
FROM seller_network
UNION ALL
SELECT
   seller_2 AS seller_from,
   seller_1 AS seller_to,
   common_customers AS weight
FROM seller_network;
```

Listing 13: Creation of the seller_edges view

Based on this representation, one of the main structural analyses performed in PostgreSQL was connected component analysis. A connected component contains sellers that can reach one another either directly or indirectly through other sellers. Since connected components are not native relational structures, they were identified using a recursive CTE. Each seller was first considered as a starting point, after which the recursive part repeatedly joined the currently reachable sellers to the seller_edges view. The resulting reachability information was then used to assign sellers to components and calculate the size of each component. The corresponding PostgreSQL query for identifying the connected components is shown in *Listing 14*.

```
WITH RECURSIVE
all_sellers AS (
   SELECT DISTINCT seller_from AS seller_id
   FROM seller_edges),
reachability AS (
   SELECT seller_id AS root, seller_id AS reachable_seller
   FROM all_sellers
   UNION
   SELECT r.root, e.seller_to
   FROM reachability r
   JOIN seller_edges e ON r.reachable_seller = e.seller_from),
seller_components AS (
   SELECT root AS seller_id, MIN(reachable_seller) AS component_key
   FROM reachability
   GROUP BY root),
component_numbers AS (
   SELECT component_key, DENSE_RANK() OVER (ORDER BY component_key ) AS component_id
   FROM (
      SELECT DISTINCT component_key
      FROM seller_components) x),
component_sizes AS (
   SELECT component_key, COUNT(*) AS component_size
   FROM seller_components
   GROUP BY component_key)
SELECT sc.seller_id, cn.component_id, cs.component_size
FROM seller_components sc
JOIN component_numbers cn ON sc.component_key = cn.component_key
JOIN component_sizes cs ON sc.component_key = cs.component_key
ORDER BY  cs.component_size DESC, cn.component_id, sc.seller_id;
```

Listing 14: Identification of connected components in the PostgreSQL seller network

To extend the connected component analysis, a second version of the seller network was created in which only stronger seller relationships were retained. While the seller_edges view included seller pairs sharing at least two customers, the strong_seller_edges view increased this threshold to five common customers. The purpose of this additional analysis was to examine how the structure of the network changes when weaker relationships are removed and only connections supported by a larger customer overlap remain. The connected component analysis was then repeated using these stronger relationships. To ensure a valid comparison with the original network, the same set of sellers was retained in both analyses. Therefore, sellers that no longer had any connection after applying the higher threshold were not removed from the analysis, but remained as isolated components of size one. Only the set of relationships was changed. This makes it possible to observe whether larger connected groups remain connected through strong relationships or divide into smaller components when weaker links are excluded. This analysis provides an additional perspective on the structure of the seller network. The original network describes connectivity based on relatively weak customer overlap, whereas the stronger network focuses on relationships supported by at least five shared customers. Comparing the resulting component structures therefore shows how dependent the connectivity of the network is on weaker seller relationships. However, this remains a threshold based connected component analysis, since the groups are determined by reachability after applying a manually selected relationship threshold rather than by a dedicated community detection algorithm. *Table 12* summarizes the differences in the component structure between the original seller network and the stronger network.

Aspect	seller_edges	strong_seller_edges
number of components	27	191
min component size	2	1
max component size	153	15

Table 12: Result comparison based on component analysis

PostgreSQL was also capable of examining the local structure of the seller network. The **clustering coefficient**, for example, was calculated by identifying the neighbours of each seller, determining how many pairs of these neighbours were themselves connected, and comparing this number with the maximum possible number of connections between the neighbours. Unlike the previous connected component analysis, which describes the network at a broader structural level, the clustering coefficient focuses on the immediate neighbourhood of individual sellers and indicates how densely interconnected their neighbouring sellers are.

In the relational implementation, this calculation required several explicit processing steps. First, the neighbors CTE extracted the distinct neighbours of each seller from the bidirectional seller_edges view. The degrees CTE then counted these neighbours to obtain the degree of each seller. Subsequently, neighbor_connections compared the neighbours of each seller with one another and used another join with seller_edges to determine which neighbour pairs were themselves connected. Since the bidirectional edge representation causes each connected neighbour pair to be encountered twice, the resulting count was divided by two. Finally, the number of connected neighbour pairs was divided by the maximum possible number of such pairs, calculated as $degree \cdot (degree - 1)/2$, to obtain the local clustering coefficient. Only sellers with a degree of at least five were retained in the final result. The corresponding query for calculating the local clustering coefficients is shown in *Listing 15*.

```
WITH neighbors AS (
   SELECT DISTINCT seller_from AS seller, seller_to AS neighbor
   FROM seller_edges),
degrees AS (
   SELECT seller, COUNT(*) AS degree
   FROM neighbors
   GROUP BY seller),
neighbor_connections AS (
   SELECT n1.seller, COUNT(*) / 2.0 AS connected_neighbor_pairs
   FROM neighbors n1
   JOIN neighbors n2 ON n1.seller = n2.seller
     AND n1.neighbor <> n2.neighbor
   JOIN seller_edges e ON e.seller_from = n1.neighbor
     AND e.seller_to = n2.neighbor
   GROUP BY n1.seller)
SELECT d.seller, d.degree,
   COALESCE(nc.connected_neighbor_pairs, 0) AS connected_neighbor_pairs,
   CASE
      WHEN d.degree >= 2 THEN
         COALESCE(nc.connected_neighbor_pairs, 0) /(d.degree * (d.degree - 1) / 2.0)
      ELSE 0
   END AS clustering_coefficient
FROM degrees d
LEFT JOIN neighbor_connections nc ON d.seller = nc.seller
WHERE d.degree >= 5
ORDER BY clustering_coefficient DESC, degree DESC;
```

Listing 15: Calculation of Local Clustering Coefficients in PostgreSQL

This calculation illustrates an important characteristic: the clustering coefficient can be calculated successfully, but the graph concept has to be reconstructed through relational operations. Neighbourhoods, degrees, and connections between neighbours are not directly available as graph structures and therefore have to be derived through CTEs, self-joins, aggregation, and explicit mathematical calculation.

The same seller network can be analysed more directly in Neo4j. Since sellers are represented as nodes and their shared customer associations are already stored as SHARED_CUSTOMERS relationships, the existing data model corresponds directly to the network required for the analysis. The **local clustering coefficient** can therefore be calculated in a more natural graph oriented manner by using the GDS library. Instead of manually reconstructing neighbourhoods and connections between neighbours, as required in PostgreSQL, GDS provides a dedicated local clustering coefficient algorithm that operates directly on a projected graph. For this purpose, a graph named sellerGraph was created in the GDS graph catalog using Seller nodes and SHARED_CUSTOMERS relationships. Since the shared customer relationship is natively symmetric, the relationships were projected with an undirected orientation. The gds.localClusteringCoefficient algorithm was then applied to the projected graph to calculate the coefficient for each seller. The algorithm directly returns the coefficient for each seller node, after which the degree was added and the same minimum degree of five was applied as in PostgreSQL. The corresponding Neo4j query is shown in *Listing 16*.

```
CALL gds.localClusteringCoefficient.stream('sellerGraph')
YIELD nodeId, localClusteringCoefficient
WITH
   gds.util.asNode(nodeId) AS seller, localClusteringCoefficient
MATCH (seller)-[:SHARED_CUSTOMERS]-(neighbor:Seller)
WITH seller, localClusteringCoefficient, count(DISTINCT neighbor) AS degree

WHERE degree >= 5

RETURN seller.seller_id AS seller, degree, localClusteringCoefficient AS clustering_coefficient
ORDER BY clustering_coefficient DESC, degree DESC;
```

Listing 16: Local clustering coefficient calculation using the corresponding algorithm in Neo4j

This creates a particularly clear comparison between the two systems. Both implementations calculate the same type of network property, but PostgreSQL requires the underlying neighbourhood logic and coefficient formula to be explicitly constructed, whereas Neo4j provides a dedicated graph algorithm operating directly on the projected seller network. Despite these implementation differences, the queries produced the same clustering coefficient results in both database systems. In terms of execution time, both approaches were also efficient in this experiment, with the results obtained within a few milliseconds. *Figure 9* shows several example results of the local clustering coefficient calculation in Neo4j.

	seller	degree	clustering_coefficient
1	"41b39e28db005d9731d9d485a83b4c38"	5	0.9
2	"cfb1a033743668a192316f3c6d1d2671"	5	0.7
3	"6973a06f484aacf400ece213dbf3d946"	5	0.6
4	"0b90b6df587eb83608a64ea8b390cf07"	5	0.6
5	"d1c281d3ae149232351cd8c8cc885f0d"	7	0.5714285714285714
6	"a3e9a2c700480d9bb01fba070ba80a0e"	5	0.5

Figure 9: Some Local Clustering Coefficent examples in Neo4j

The seller network was analysed further in Neo4j using **PageRank.** While degree measures only how many other sellers a seller is connected to, PageRank provides a more structural interpretation of importance by considering not only the existence of connections but also the importance of the nodes reached through those connections. In the seller network, a seller can therefore obtain a high PageRank score even without having the highest degree if it occupies a strategically important position through its connections to other influential sellers. To perform the PageRank analysis, a GDS graph projection named sellerSharedCustomersGraph was created from the seller network.

The projection included the Seller nodes and the SHARED_CUSTOMERS relationships between them, with the relationships treated as undirected. The shared_customers property was included as the relationship weight, representing the strength of the connection between each pair of sellers. PageRank was then executed on this projected graph to identify the most important sellers within the shared customer network. The GDS graph projection and the corresponding PageRank query are presented in *Listing 17*. The highest ranked sellers obtained from the PageRank analysis are shown in *Figure 10*.

GDS graph projection	PageRank Algorithm

```
CALL gds.graph.project(
  'sellerSharedCustomersGraph',
  'Seller',
  {
    SHARED_CUSTOMERS: {
      orientation: 'UNDIRECTED',
      properties: 'shared_customers'
    }
  }
);
```

```
CALL gds.pageRank.stream(
  'sellerSharedCustomersGraph',
{relationshipWeightProperty:'shared_customers'
  })
YIELD nodeId, score
RETURN
  gds.util.asNode(nodeId).seller_id AS seller,
  score AS pagerank
ORDER BY pagerank DESC
LIMIT 20;
```

Listing 17: GDS Graph Projection and PageRank Calculation

	seller	pagerank
1	"4a3ca9315b744ce9f8e9374361493884"	8.842422229227752
2	"6560211a19b47992c3666cc44a7e94c0"	5.62222764347986
3	"1025f0e2d44d7041d6cf58b6550e0bfa"	4.301397404876692
4	"1900267e848ceeba8fa32d80c1a5f5a8"	4.162908954477706
5	"1f50f920176fa81dab994f9023523100"	3.859893018990568
6	"cca3071e3e9bb7d12640c9fbe2301306"	3.816314928816713
7	"cc419e0650a3c5ba77189a1882b7556a"	3.4272507982884948
8	"e9779976487b77c6d4ac45f75ec7afe9"	2.867899439467119
9	"1835b56ce799e6a4dc4eddc053f04066"	2.818170382548653

Figure 10: Top Sellers Ranked by PageRank Score in Neo4j

An analogous PageRank calculation could theoretically be implemented in PostgreSQL, as the algorithm can be expressed through iterative numerical operations on an edge list representation of the network. However, PostgreSQL does not provide PageRank as a built-in relational operation or graph analysis function. Implementing the algorithm directly would therefore require more complex iterative or recursive queries, or the use of an additional graph extension or external analytical tool. As a result, manually reproducing PageRank in PostgreSQL would add considerable complexity without providing much value for the present comparison.

The same general limitation also applies to community detection. In PostgreSQL, the earlier connected component analysis identified groups of sellers that were reachable from one another, and the additional threshold based analysis demonstrated how these groups changed when only stronger seller relationships were retained. However, connected components should not be interpreted as equivalent to communities. A connected component only requires that a path exists between its members, whereas community detection aims to identify densely connected groups within a network. The PostgreSQL implementation therefore provides useful structural grouping, but it does not perform community detection in the algorithmic sense.
Neo4j was able to extend the analysis using the **Louvain community detection algorithm**. In this context, a community represents a group of sellers that are relatively strongly interconnected within the shared customer network. This makes Louvain analytically different from the threshold-based connected components calculated in PostgreSQL: instead of first manually removing edges below a selected threshold and then determining reachability, Louvain analyses the network structure to identify communities according to its optimization criterion. As with PageRank, implementing the Louvain algorithm directly in standard PostgreSQL would be theoretically possible but highly complex and impractial, since it requires iterative graph specific calculations for which the relational system does not provide a native abstraction. Neo4j GDS, in contrast, provides Louvain as an algorithm that is already available for direct use.
The corresponding Neo4j query for applying the Louvain community detection algorithm is shown in *Listing 18*.

```
CALL gds.louvain.stream(
  'sellerSharedCustomersGraph',
  { relationshipWeightProperty: 'shared_customers'})
YIELD nodeId, communityId

RETURN
  communityId,
  COUNT(*) AS sellers_in_community
ORDER BY sellers_in_community DESC;
```

Listing 18: Lovain algorithm on the sellerSharedCustomersGraph projection

The Louvain result also highlights the importance of graph projection choices. The projected graph contained all Seller nodes, whereas SHARED_CUSTOMERS relationships existed only between sellers sharing at least two customers. As a result, many sellers were isolated in the projected graph. This explains both the 2,901 returned records and the large number of single seller communities. Therefore, the number of detected communities should be interpreted with caution, as many of them simply represent isolated sellers rather than larger, structurally meaningful groups. *Figure 11* presents the ten largest communities identified by the Louvain algorithm.

	communityId	sellers_in_community
1	2996	43
2	2617	33
3	2207	18
4	2720	17
5	1302	15
6	2962	15
7	606	9
8	332	9
9	471	7
10	2215	5

Figure 11: The 10 largest communities identified by Louvain community detection in Neo4j

Table 13 summarizes the execution times of the seller network experiments in PostgreSQL and Neo4j. Both systems performed efficiently in network construction and local clustering coefficient calculations, with relatively small differences in execution time. More advanced analyses, such as PageRank and Louvain community detection, were performed only in Neo4j using the Graph Data Science library, highlighting its specialized capabilities for network analysis.

Experiment	PostgreSQL execution time	Neo4j execution time
Network Construction	~1s 241ms	~2 s 144 ms
Connected Component Analysis with seller_edges	~95 ms	-
Local Clustering Coefficient	~105 ms	~88 ms
PageRank	-	~210 ms
Lovain Community Detection	-	~245 ms

Table 13: Execution time comparison of seller network construction and analysis in PostgreSQL and Neo4j

Overall, the seller network analysis shows that the differences between relational and graph databases become more noticeable as the analysis becomes more graph oriented. A relational database can represent the seller network as an edge list and calculate several graph measures, such as degree, connected components, and clustering coefficients, using joins, recursive CTEs, aggregation, and explicit calculations. However, these implementations become more complex when the analysis focuses on the structure and topology of the network. In contrast, a graph database represents seller relationships directly as graph structures and, in the case of Neo4j, the Graph Data Science library provides dedicated algorithms such as local clustering coefficient, PageRank, and Louvain community detection. Therefore, the main advantage of the graph database approach is not that these analyses are impossible in relational databases, but that it provides a more direct and specialized environment for network analysis.

7.3.4 Multi-Step Traversal and the Effect of Path Length

The following comparison focuses on multi step traversal and investigates how relational and graph databases behave as the required path length increases. Unlike the previous analyses, where the objective was to identify a particular relationship or calculate a network measure, this experiment examines the process of navigating through an interconnected data structure. A fixed unique customer was selected as the starting point, and the number of distinct entities reachable at an exact distance from this starting point was measured. The path length was gradually increased while the starting entity and the allowed types of connections remained unchanged. This makes it possible to observe both how the reachable part of the network expands and how the two database models express the same traversal problem.
For comparability, the same connections were considered in both database systems: connections between unique customers and customer records, customers and orders, orders and products, and products and sellers. In Neo4j, these connections are represented directly by the BELONGS_TO, PLACED, CONTAINS, and SELLS relationships. PostgreSQL represents the same information through the attributes and relationships of the relational schema. One structural difference is that CustomerUnique exists as a separate node in Neo4j, whereas in PostgreSQL the customer_unique_id is stored directly in the customers relation.

To support the traversal in PostgreSQL, these relational connections were combined into a graph_edges view. The view provides a common edge list representation in which each row contains an edge_id, two connected elements (node_from and node_to), and a relationship_type. Prefixes such as CU:, C:, O:, P:, and S: are used to distinguish unique customers, customer records, orders, products, and sellers. The BELONGS_TO connections are constructed from customer_id and customer_unique_id, PLACED from the relationship between customers and orders, CONTAINS from order_items, and SELLS from distinct seller–product combinations. In this way, the relational data can be traversed through a unified edge structure without changing the underlying PostgreSQL data model. The traversal does not follow a fixed sequence of entity types. At each step, any permitted relationship connected to the current entity can be followed in either direction. As a result, the search can branch into different parts of the interconnected structure and may reach the same types of entities again through different paths as the traversal depth increases. The graph_edges view used to represent the permitted relationships for the traversal analysis is defined in *Listing 19*. An example of the resulting view is shown in *Figure 12*.

```sql
CREATE VIEW graph_edges AS
SELECT
  'BELONGS_TO:' || c.customer_id AS edge_id,
  'C:'  || c.customer_id AS node_from,
  'CU:' || c.customer_unique_id AS node_to,
  'BELONGS_TO'::text AS relationship_type
FROM customers c
WHERE c.customer_unique_id IS NOT NULL
UNION ALL
SELECT
  'PLACED:' || o.order_id AS edge_id,
  'C:' || o.customer_id AS node_from,
  'O:' || o.order_id AS node_to,
  'PLACED'::text AS relationship_type
FROM orders o
UNION ALL
SELECT
  'CONTAINS:' || oi.order_item_pk::text AS edge_id,
  'O:' || oi.order_id AS node_from,
  'P:' || oi.product_id AS node_to,
  'CONTAINS'::text AS relationship_type
FROM order_items oi
UNION ALL
SELECT DISTINCT
  'SELLS:' || oi.seller_id || ':' || oi.product_id AS edge_id,
  'S:' || oi.seller_id AS node_from,
  'P:' || oi.product_id AS node_to,
  'SELLS'::text AS relationship_type
FROM order_items oi;
```

Listing 19: Definition of the graph_edges view

	edge_id text	node_from text	node_to text	relationship_type text
1	BELONGS_TO:06b8999e2fba1a1fbc88172c00ba8...	C:06b8999e2fba1a1fbc88172c00ba8...	CU:861eff4711a542e4b93843c6dd7febb0	BELONGS_TO
2	BELONGS_TO:18955e83d337fd6b2def6b18a428a...	C:18955e83d337fd6b2def6b18a428a...	CU:290c77bc529b7ac935b93aa66c333d...	BELONGS_TO
3	BELONGS_TO:4e7b3e00288586ebd08712fdd037...	C:4e7b3e00288586ebd08712fdd037...	CU:060e732b5b29e8181a18229c7b0b2b...	BELONGS_TO
4	BELONGS_TO:b2b6027bc5c5109e529d4dc6358b...	C:b2b6027bc5c5109e529d4dc6358b...	CU:259dac757896d24d7702b9acbbff3f3c	BELONGS_TO
5	BELONGS_TO:4f2d8ab171c80ec8364f7c12e35b2...	C:4f2d8ab171c80ec8364f7c12e35b2...	CU:345ecd01c38d18a9036ed96c73b8d0...	BELONGS_TO

Figure 12: First 5 rows of the graph_edges view

The PostgreSQL traversal itself is implemented with a recursive CTE. The selected unique customer is inserted as the initial node at depth zero. During each recursive iteration, graph_edges is joined to the nodes reached during the previous iteration. Since an edge can match either node_from or node_to, the traversal treats the selected relationships as undirected for the purpose of this experiment. The CASE expression determines which endpoint represents the next node. In addition, an array named used_edges records the relationships already used within a particular path, preventing the same edge from being traversed repeatedly within that path. The recursion continues until the required depth is reached, after which COUNT(DISTINCT node) determines how many distinct nodes can occur as endpoints at exactly that depth. The corresponding recursive SQL query for depth 7 is shown in *Listing 20*.

```
WITH RECURSIVE traversal AS (
  SELECT
    'CU:0000366f3b9a7992bf8c76cfdf3221e2'::text AS node,
    ARRAY[]::text[] AS used_edges,
    0 AS depth
  UNION ALL
  SELECT
    CASE
      WHEN e.node_from = t.node THEN e.node_to
      ELSE e.node_from
    END AS node,
    t.used_edges || e.edge_id,
    t.depth + 1
  FROM traversal t
  JOIN graph_edges e ON e.node_from = t.node OR e.node_to = t.node
  WHERE t.depth < 7
   AND NOT e.edge_id = ANY(t.used_edges))

SELECT
  COUNT(DISTINCT node) AS reachable_nodes
FROM traversal
WHERE depth = 7;
```

Listing 20: Recursive SQL query for counting nodes reachable at depth 7

Neo4j approaches exactly the same problem from a different starting point. The entities already exist as nodes and their associations as relationships, so no equivalent of the PostgreSQL graph_edges view is required. The starting CustomerUnique is located through its customer_unique_id, after which the traversal pattern defines the allowed relationship types and the exact number of steps to be followed. For example, the expression [:BELONGS_TO|PLACED|CONTAINS|SELLS*7] searches for paths of exactly seven relationships using only the listed relationship types. The absence of an arrow makes the traversal independent of stored relationship direction, corresponding to the bidirectional treatment of edges in the PostgreSQL recursive query. Finally, count(DISTINCT target) counts the distinct endpoints reached at the requested distance. The corresponding query for the seven step traversal is shown in *Listing 21*.

```
MATCH (start:CustomerUnique {
  customer_unique_id:
  '0000366f3b9a7992bf8c76cfdf3221e2'})
MATCH path = (start)-[:BELONGS_TO|PLACED|CONTAINS|SELLS*7]-(target)

RETURN count(DISTINCT target) AS reachable_nodes;
```

Listing 21: Neo4j query for seven-step traversal

The results show how the network around the selected customer grows as the traversal depth increases. The queries were run for depths from 1 to 7 steps in both PostgreSQL and Neo4j. Both databases produced the same results at every depth, with the same number of distinct reachable nodes. The corresponding reachable node counts and execution times are summarized in *Table 14*.

Path length	Reachable nodes	PostgreSQL execution time	Neo4j execution time
1	1	~0.90 s	~1 ms
2	1	~1.28 s	~5 ms
3	1	~1.69 s	~4 ms
4	23	~2.77 s	~3 ms
5	245	~10.01 s	~5 ms
6	1,341	~1 min 0.25 s	~6 ms
7	1,618	~4 min 25.71 s	~11 ms

Table 14: Reachable nodes and execution time at different traversal depths

The pattern of reachable entities also reflects the structure of the underlying Olist network. At the first few traversal depths, the search remains close to the selected customer and therefore reaches only a limited number of entities. As the traversal progresses and reaches products and sellers, many additional connections to other orders and customers become accessible, causing the number of reachable entities to increase rapidly. This expansion is particularly visible at depths 5 and 6. By depth 7, however, the growth is smaller. This suggests that the traversal paths increasingly reach entities that were already found instead of discovering many new parts of the network.

The execution times show a clear difference in how the two systems handle increasing traversal depth. In PostgreSQL, the runtime increases moderately at the lower depths but rises sharply as the traversal becomes deeper. This is mainly caused by the recursive CTE repeatedly processing the edge representation and generating an increasing number of intermediate path combinations that must be joined, filtered, and checked for previously used edges. As the search space expands, the relational engine therefore has to perform substantially more work before the final distinct reachable nodes can be determined. Neo4j shows a very different pattern. Its execution time remains in the millisecond range across all tested depths, varying only slightly between approximately 3 and 11 ms. This behaviour is supported by Neo4j's graph storage and traversal mechanisms, where relationships are stored as direct connections between nodes and can be followed without repeatedly reconstructing them through joins. The traversal operator can therefore expand through the graph structure directly, which keeps the execution time relatively stable even as the number of reachable entities increases at greater depths. The difference in execution behaviour between the two systems is illustrated in *Figure 13*, where the logarithmic scale highlights the rapidly increasing PostgreSQL runtime compared with the relatively stable Neo4j execution time.

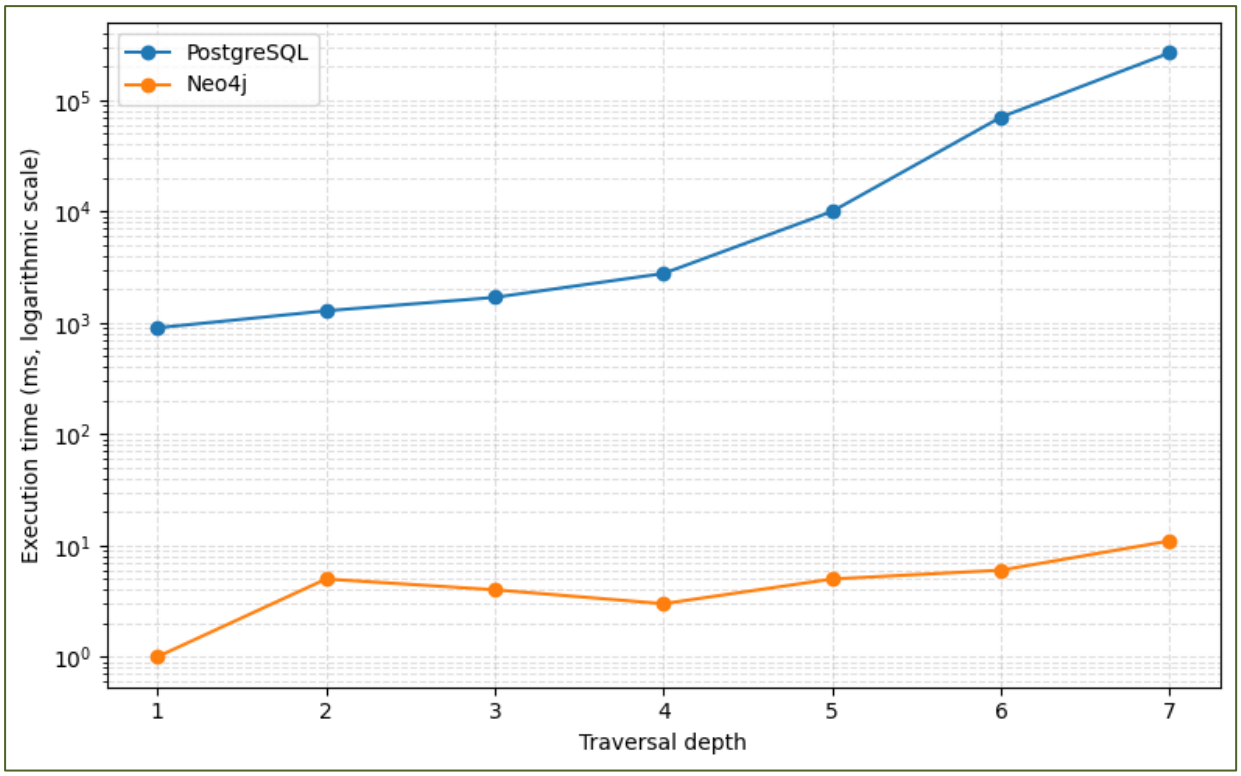


Figure 13: Execution time by traversal depth in both systems

7.3.4.1 The Execution Plan

The PostgreSQL execution plan at depth 7 explains the sharp increase in execution time observed at greater traversal depths. Most of the execution time is spent within the recursive part of the query rather than in the final aggregation. The graph_edges view, containing 344,753 edges, is repeatedly processed during the recursive execution, while the growing number of intermediate traversal states substantially increases the work required by the nested loop join. At depth 7, approximately 95.9 million candidate combinations are rejected by the join filter. Furthermore, 6,803 traversal results are produced at the final depth before the COUNT(DISTINCT node) operation reduces them to 1,618 distinct reachable entities. This demonstrates that the increasing execution time is caused not only by the growth in the number of final reachable entities, but more importantly by the rapid expansion of the intermediate search space that must be evaluated during recursive traversal.
In contrast, the Neo4j execution profile retains the same basic execution strategy between depths 4 and 7. Although the number of database accesses increases from 41 to 6,579, the traversal continues to be handled by the VarLengthExpand(Pruning) operator and remains within the millisecond range. This indicates that the increased traversal depth also results in substantially more work in Neo4j, but the native relationship-based traversal handles this expansion considerably more efficiently in this experiment.

Overall, the difference in query construction is therefore significant even though the analytical question is the same. The implementation in PostgreSQL demonstrates an important characteristic of relational databases when dealing with graph traversal. It is capable of performing the operation, but the graph semantics have to be constructed explicitly. The relational schema first has to be converted into a common edge representation, the direction-independent traversal has to be expressed through join conditions, the current depth must be maintained explicitly, and already-

used edges must be stored as part of the recursive state. The SQL query therefore describes not only what should be found, but also much of how the graph should be traversed. At the same time in Neo4j, traversal is expressed directly as a path pattern. This represents one of the clearest conceptual advantages of the graph model in this experiment: path length is a native part of the query expression rather than a recursive procedure that must be manually constructed. This advantage was also clearly reflected in the execution time results, where Neo4j handled increasing traversal depths noticeably more efficiently, while the execution time of the PostgreSQL recursive query increased sharply as the search depth grew.

8. Conclusion

The aim of this thesis was to compare relational and graph database approaches using the same e-commerce domain and underlying dataset. The Brazilian E-Commerce Public Dataset by Olist was implemented in PostgreSQL using a relational model and in Neo4j using a labeled property graph model. The comparison focused on database construction, query formulation, analytical capabilities, and execution behaviour across tasks of increasing structural complexity.

The construction of the two databases already demonstrated important differences between the models. In PostgreSQL, the database structure was strongly influenced by primary keys, foreign keys, and referential integrity constraints. The order of data loading therefore had to follow the dependencies between the relations, and records containing unresolved references had to be excluded in order to maintain consistency. Neo4j provided greater structural flexibility during the import process because nodes could exist even when some expected relationships were missing. However, for the comparative experiments, the same exclusions were applied to both databases so that they operated on the same cleaned subset of the Olist data. This ensured that differences observed in later analyses resulted from the database models and processing approaches rather than from differences in the underlying data.

The basic query comparison showed that both PostgreSQL and Neo4j are well suited to conventional descriptive analytical tasks. Queries involving filtering, grouping, aggregation, and relatively simple combinations of entities could be expressed clearly in both SQL and Cypher, and no meaningful performance difference was observed in these operations. PostgreSQL was particularly natural for operations that correspond directly to tabular aggregation, while Neo4j became increasingly expressive as more connected entities had to be included in a query. The main difference at this level was therefore not the analytical result itself, but the way relationships were accessed: PostgreSQL reconstructs them through joins based on keys, whereas Neo4j follows relationships that are explicitly represented in the graph.

More substantial differences appeared in the advanced analyses. Customer similarity demonstrated that performance depends not only on the selected database model but also strongly on query design. In PostgreSQL, the initial category-based similarity query generated a very large number of customer pairs and failed to complete within a practical amount of time. By redesigning the query around shared category pairs, the search space could be substantially reduced and the same analysis completed efficiently. Neo4j provided a conceptually direct representation of customer similarity through shared neighbourhoods, but a direct Cypher implementation based on common sellers also encountered memory limitations. The Graph Data Science Node Similarity algorithm provided a more appropriate solution by operating on an in-memory graph projection specifically designed for this type of analysis. These results demonstrate that neither relational nor graph-based representations automatically guarantee efficient execution; the formulation of the problem and the availability of suitable processing methods remain important factors.

The recommendation experiments further illustrated this point. Both systems were capable of implementing category and product recommendation scenarios based on customer similarity. PostgreSQL expressed the required logic through joins, aggregation, and candidate filtering, while Neo4j navigated through the relationships between customers, categories, orders, and products. The PostgreSQL experiments also showed the importance of indexing, as adding an index to `customer_unique_id` remarkably reduced the execution time of the targeted product recommendation query. Therefore, the comparison indicates that database performance should not be interpreted only as a consequence of the underlying data model, since query formulation, indexing, and optimization can substantially influence the final result.

The seller network analysis made the distinction between the two approaches more visible. PostgreSQL successfully represented the seller network as an edge structure and supported

analyses such as degree calculation, connected components, and local clustering coefficients. However, these analyses required the graph concepts to be reconstructed explicitly through joins, recursive common table expressions, aggregation, and mathematical calculations. Neo4j represented the same seller relationships directly as graph relationships, allowing network-oriented analyses to operate on the graph structure itself. Through the Graph Data Science library, dedicated algorithms could be applied for local clustering coefficient, PageRank, and Louvain community detection. The significance of this difference is not that advanced network analysis is fundamentally impossible in a relational database, but that increasingly graph oriented problems require increasingly complex relational constructions, while a graph database provides a more specialized environment for these tasks.

The clearest difference between the two systems was observed in the multi-step traversal experiment. Both databases returned the same number of reachable entities at every tested traversal depth, confirming that the two implementations represented the same logical connections. However, the way the traversal was expressed and executed differed considerably. PostgreSQL required the relational connections to be transformed into a common edge representation and traversed through a recursive CTE that explicitly maintained the current depth and the edges already used within each path. As the traversal depth increased, the number of intermediate traversal states expanded rapidly, causing a sharp increase in execution time. Neo4j, in contrast, expressed the same operation directly as a path pattern and handled all tested traversal depths within the millisecond range. Although the amount of work performed by Neo4j also increased with depth, its native relationship-based traversal handled this expansion considerably more efficiently in this experiment.

Taken together, the experiments show that the distinction between relational and graph databases becomes more important as the analytical focus shifts from individual records and tabular aggregation toward relationships, network structure, and multi-step navigation. PostgreSQL proved capable of supporting all major categories of analysis examined in this thesis and, with appropriate query design and optimization, could also perform several tasks that are naturally associated with graph processing. Its relational structure remains particularly suitable when the data is naturally tabular, integrity constraints are central, and the required analyses can be expressed efficiently through joins and aggregations. Neo4j, on the other hand, provides a more direct representation when relationships themselves form a central part of the problem and offers specialized mechanisms for graph algorithms and traversal.

Overall, the comparison demonstrates that neither database model offers a universal advantage across all analytical tasks. The appropriate choice depends on the structure of the data, the importance and complexity of its relationships, and the types of queries and analyses that need to be performed. For the Olist dataset, both approaches proved effective for conventional queries and several relationship-based analyses, but the advantages of the graph model became increasingly apparent as the tasks became more dependent on network topology and deeper traversal. Rather than identifying one model as generally superior, the results show that relational and graph databases are better suited to different types of analytical problems, and their relative advantages depend on the requirements of the specific task.

Biography



Ramona Molnar was born on September 30, 2000, in Senta, Serbia. She attended “Ferenc Kiš” Elementary School in Trešnjevac and later the mathematics program at “Bolyai Secondary Grammar School and Dormitory for Gifted Students“ in Senta.
In 2019, she began her bachelor’s studies in Mathematics at the Faculty of Sciences, University of Novi Sad, which she completed in July 2023.
In the same year, she continued her education at the same university by enrolling in the master’s degree program in Applied Mathematics – Data Science. In 2026, she completed all required examinations with an average grade of 9.29.

Bibliography

[1] Ramez Elmasri, Shamkant B.Navathe, *Fundamentals of Database Systems-7th edition*, 2016.

[2] Cheng Chen, *Multi-Perspective Evaluation of Relational and Graph Databases*, Master's thesis, University of Helsinki, 2022.

[3] Marko A. Rodriguez, Peter Neubauer, *Constructions from Dots and Lines*, Bulletin of the American Society for Information Science and Technology, 2009.

[4] Watt, A. and N. Eng, *Database Design – 2nd Edition*, Victoria, B.C.: BCcampus. Retrieved from https://opentextbc.ca/dbdesign01/, 2014.

[5] Michael Hunger, Ryan Boyd & William Lyon, *The Definitive Guide to Graph Databases for the RDBMS Developer*

[6] LibreTexts, *Introduction to Database Systems*, LibreTexts, 2026.

[7] Oshini Goonetilleke, *Graph Database Management Systems*, RMIT University, College of Science, Engineering, and Health, 2017

[8] Ian Robinson, Jim Webber & Emil Eifrem, *Graph Databases*, Second Edition, O'Reilly Media, 2015.

[9] Rik Van Bruggen, *Learning Neo4j*, Packt Publishing, 2014.

[10] Supriya Ramireddy, *Query Processing in Graph Databases*, Master's Thesis, University of Georgia, Athens, Georgia, 2017.

[11] Peter Lake, Paul Crowther, *Concise Guide to Databases*, A practical Introduction, Springer-Verlag London, 2013

[12] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, IBM Research Laboratory, San Jose, California, June 1970.

[13] Josh Mintz, "A Brief Overview of the Database Landscape," IBM, available online: https://www.ibm.com/think/topics/database-types, accessed: August 2026.

[14] Søren Kejser Jensen, Torben Bach Pedersen, Christian Thomsen, *Time Series Management Systems: A Survey*, *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2581–2600, 2017

[15] Olist, *Brazilian E-Commerce Public Dataset by Olist*, Kaggle, available online: https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce, accessed July 2026.

[16] The PostgreSQL Global Development Group, *PostgreSQL 17.11 Documentation*. Available online: https://www.postgresql.org/docs/17/index.html. Accessed: August, 2026.

[17] M. Newman, *Networks*, 2nd ed. Oxford, U.K.: Oxford University Press, 2018.

[18] Neo4j, *The Neo4j Graph Data Science Library Manual*, Version 2026.08, 2026. Available: https://neo4j.com/docs/graph-data-science/current/ , accessed: September, 2026

[19] PostgreSQL Global Development Group, "PostgreSQL," Available online: https://www.postgresql.org/, accessed: August 2026.

[20] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, *A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective*, 2010.

[21] T. T. T. Do, T. B. Mai Hoang, V. Q. Nguyen, and Q. T. Huynh, *Query based Performance Comparison of Graph Database and Relational Database*, 2022.

[22] M. Jain, A. Khanchandani, and C. Rodrigues, *Performance Comparison of Graph Database and Relational Database.*

[23] J. Hölsch, T. Schmidt, and M. Grossniklaus, *On the Performance of Analytical and Pattern Matching Graph Queries in Neo4j and a Relational Database*, 2017.

[24] P. Kotiranta, M. Junkkari, and J. Nummenmaa, *Performance of Graph and Relational Databases in Complex Queries*, Applied Sciences, 2022.

[25] C. Győrödi, R. Győrödi, and R. Sotoc, *A Comparative Study of Relational and Non Relational Database Models in a Web Based Application*, 2015.

[26] C. Tyagi, *Comparative Analysis of Relational Databases and Graph Databases*, 2012.

[27] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*, 3rd ed. Cambridge: Cambridge University Press, 2020.

[28] R. Angles and C. Gutierrez, *Survey of Graph Database Models*, ACM Computing Surveys, vol. 40, no. 1, pp. 1–39, 2008. Available: https://doi.org/10.1145/1322432.1322433

[29] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, Cypher: *An Evolving Query Language for Property Graphs*, Proceedings of the 2018 International Conference on Management of Data (SIGMOD), pp. 1433–1445, 2018. Available: https://doi.org/10.1145/3183713.3190657

[30] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi, Extending In-Memory *Relational Database Engines with Native Graph Support*, 2017. Available: https://arxiv.org/abs/1709.06715

UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj: **RBR**	
Identifikacioni broj: **IBR**	
Tip dokumentacije: **TD**	Monografska dokumentacija
Tip zapisa: **TZ**	Tekstualni štampani materijal
Vrsta rada: **VR**	Master rad
Autor: **AU**	Ramona Molnar
Mentor: **MN**	dr. Lidija Fodor
Naslov rada: **NR**	Poređenje relacionih i grafovskih baza podataka u sistemima sa kompleksnim relacijama između entiteta
Jezik publikacije: **JP**	engleski
Jezik izvoda: **JI**	srpski
Zemlja publikovanja: **ZP**	Srbija
Uže geografsko područje: **UGP**	Vojvodina
Godina: **GO**	2026
Izdavač: **IZ**	Autorski reprint
Mesto i adresa: **MA**	Prirodno-matematički fakultet, Trg Dositeja Obradovića 4, Novi Sad
Fizički opis rada: **FO**	8/72/30/14/13/0/0
Naučna oblast: **NO**	Računarske nauke
Naučna disciplina: **ND**	Baza podataka
Predmetna odrednica/ ključne reči: **PO UDK**	Relacione baze podataka, grafovske baze podataka, PostgreSQL, Neo4j, SQL,Cypher, analiza performansi upita
Čuva se: **ČU**	Biblioteka Departmana za matematiku i informatiku, PMF-a u Novom Sadu
Važna napomena: **VN**	nema
Izvod: **IZ**	U ovom master radu sprovedena je uporedna analiza relacionih i grafovskih baza podataka na primeru javno dostupnog skupa podataka o elektronskoj trgovini kompanije Olist. Isti skup podataka implementiran je u sistemu PostgreSQL primenom relacionog modela i u sistemu Neo4j primenom grafovskog modela podataka. Cilj istraživanja bio je da se ispitaju razlike između ova dva pristupa u pogledu modelovanja podataka, izgradnje baze, formulisanja upita, analitičkih mogućnosti i performansi izvršavanja.

Analiza je obuhvatila osnovne upite, kao i složenije zadatke koji uključuju identifikaciju sličnih kupaca na osnovu zajedničkih prodavaca i kategorija proizvoda, generisanje preporuka, analizu mreže prodavaca i pretragu grafa na različitim dubinama. Rezultati su pokazali da su oba sistema pogodna za izvršavanje osnovnih upita, dok su značajnije razlike uočene kod složenijih analiza zasnovanih na međusobnim vezama između entiteta. PostgreSQL je omogućio realizaciju ovih zadataka korišćenjem spajanja tabela, agregacija i rekurzivnih upita, dok je Neo4j pružio direktniji pristup povezanim podacima i specijalizovane algoritme za analizu grafova. Posebno izražena razlika u performansama zabeležena je kod pretrage grafa na većim dubinama, gde je Neo4j ostvario znatno kraće vreme izvršavanja.Na osnovu dobijenih rezultata zaključeno je da nijedan od analiziranih modela nema univerzalnu prednost. Izbor odgovarajućeg pristupa zavisi od strukture podataka, složenosti njihovih međusobnih odnosa i vrste upita i analiza koje je potrebno izvršavati.

Datum prihvatanja teme od NN veća: 14.07.2026.
DP

Datum odbrane:
DO

Članovi komisije:
KO

Predsednik: dr. Danijela Boberić Krstićev, red.prof. Prirodno-matematičkog fakulteta u Novom Sadu

Član: dr. Samir Zahirović, docent Prirodno-matematičkog fakulteta u Novom Sadu

Član, mentor: dr. Lidija Fodor, docent Prirodno-matematičkog fakulteta u Novom Sadu

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCE AND MATHEMATICS

KEY WORDS DOCUMENTATION

Accession number: **ANO**	
Identification number: **INO**	
Document type: **DT**	Monograph publication
Type of record: **TR**	Textual printed material
Content code: **CC**	Final paper
Author: **AU**	Ramona Molnar
Mentor/comentor: **MN**	dr. Lidija Fodor
Title: **TI**	A Comparative Analysis of Relational and Graph Databases in Systems with Complex Realationships Between Entities
Language of text: **LT**	Englsih
Language of abstract: **LA**	English
Country of publication: **CP**	Serbia
Locality of publication: **LP**	Vojvodina
Publication year: **PY**	2026
Publisher: **PU**	Author's reprint
Publication place: **PP**	Faculty of Science, Trg Dositeja Obradovića 4, Novi Sad
Physical description: ***PD***	8/72/30/14/13/0/0
Scientific field: **SF**	Computer Science
Scientific discipline: **SD**	Databases
Subject/ Key words: **SKW UC**	Relational databases, graph databases, PostgreSQL, Neo4j, SQL, Cypher, query performance analysis
Holding data: **HD**	Library of Department of Matematics and and Informatics, Trg Dositeja Obradovića 4
Note: **N**	none
Abstract: **AB**	This master's thesis presents a comparative analysis of relational and graph databases using the publicly available Olist e-commerce dataset. The same dataset was implemented in PostgreSQL using a relational model and in Neo4j using a graph data model. The aim of the research was to examine the differences between these two approaches in terms of data modeling, database construction, query formulation, analytical capabilities, and execution performance. The analysis included basic queries as well as more complex tasks involving the identification of similar customers based on common sellers and product categories, recommendation generation, seller network analysis, and graph traversal at different depths. The results showed that

both systems are suitable for executing basic queries, while more significant differences were observed in complex analyses based on relationships between entities. PostgreSQL supported these tasks through table joins, aggregations, and recursive queries, while Neo4j provided more direct access to connected data and specialized algorithms for graph analysis. A particularly significant performance difference was observed in graph traversal at greater depths, where Neo4j achieved considerably shorter execution times.

Based on the obtained results, it was concluded that neither of the analyzed models offers a universal advantage. The choice of an appropriate approach depends on the structure of the data, the complexity of the relationships between entities, and the types of queries and analyses that need to be performed.

Accepted by the Scientific Board:	14.07.2026
ASB	
Defended on:	
DE	
Thesis defend board:	
DB	
President:	dr. Danijela Boberić Krstićev, , Associate Professor, Faculty of Sciences, University of Novi Sad
Member:	dr. Samir Zahirović, Assistant Professor, Faculty of Sciences, University of Novi Sad
Member, mentor:	dr. Lidija Fodor, Assistant Professor, Faculty of Sciences, University of Novi Sad