Journal of Engineering Science and Technology
Special Issue on Development of Sustainability Systems - II, 176 - 192

ALGORITHMIC APPROACH FOR CONVERTING DENORMALIZED DATABASES INTO GRAPH MODELS

SAMA S. SAMAAN*, SAJA D. KHUDHUR, OMAR N. M. TAHER

Computer Engineering Department, University of Technology- Iraq, Baghdad, Iraq
*Corresponding Author: sama.s.samaan@uotechnology.edu.iq

Abstract

In a unified information system, there is often a need to handle data presented in various data models within the same domain. One solution to this challenge is a multimodal database, which can accommodate multiple data models concurrently. This database management system necessitates explicit mapping of data schemas. In this paper, the viability of using a graph-based database, specifically the Neo4j database, is demonstrated to handle complex relationships. Two novel algorithms have been developed to transform a flattened, denormalized database into a populated property graph model within the realm of information engineering. To assess the performance of the proposed algorithms, we apply them to Synthea, a synthetic healthcare database consisting of several Comma-Separated Values files with 1,172 patients. As a result, a graph database is constructed containing approximately 72,000 nodes and 880,000 relationships. The resulting property graph offers an efficient approach for handling and visualizing healthcare data from a graph perspective, providing insights into relationships that are often obscured in traditional relational databases.

Keywords: Denormalized database, Graph model, Healthcare dataset, Information engineering, Mapping.

1. Introduction

In today's era, a multitude of diverse data expands rapidly, each possessing unique characteristics that mandate specialized approaches for processing and storage [1]. Information engineering is centred around the systematic design, advancement, and management of information systems, aiming to comprehend the fundamental data structures, models, and architectures inherent in contemporary information systems [2]. Nevertheless, even within a unified information system, there often arises the necessity to manage data expressed in various data models within the same domain of knowledge. Moreover, the integration of diverse data models and technologies remains a significant challenge [3].

Relational data models are considered popular models in which the essential entities are relations [4]. Nevertheless, adhering to such a traditional method is not universally effective. It presents significant drawbacks, particularly in scenarios where frequent querying and processing of relationships are involved. This stems from the relational model's emphasis on normalization to ensure Atomicity, Consistency, Isolation, and Durability (ACID) transactions for maintaining data consistency. However, when intricate relationships are queried, numerous expensive joint operations ensue, resulting in substantial overhead [5]. Moreover, as big data expands, so does the complexity of relationships between interconnected entities, intensifying the overhead even further.

Compared to relational databases, graph databases are based on the graph model. They require fewer schema regulations and enable effortless management of link relationships without expensive join operations [6]. Furthermore, the performance edge of graph databases over relational ones remains consistent, meaning their performance remains steady as data volume increases, unlike relational databases, which experience exponential overhead growth [7].

Graph databases offer numerous advantages over relational databases in certain domains. However, much of this information is presented in a theoretical context, with a lack of practical evaluations supporting these claims. Beyond the general performance contrasts between graph and relational databases, it is imperative to concentrate on particular scenarios where graph databases excel and prove to be ideally suited in those contexts.

In certain instances, collected data may exhibit numerous relationships and repetitions, complicating the normalization process and necessitating thorough analysis to prevent data loss [8]. This situation is particularly prevalent in fields such as healthcare and business analysis systems. Moreover, within these information systems, the need to visualize data in ways that support human interpretation becomes crucial [9]. However, with the rapid advancement of data digitization and its essential role in information analysis and processing, the shift towards denormalized databases (DDB)has become increasingly noticeable.

Healthcare is witnessing a surge in rapidly generated digitized data from diverse sources like mobile devices, wearable sensors, Electronic Health Records [HER], social media, and remote health monitoring devices. This voluminous and diverse data, exhibiting the characteristics of big data, underscores the need for effective management solutions. Deghmani et al. [10] gave a survey about the usage of big data in the healthcare sector from a graph database perspective. The paper introduced the capabilities of big data analytics in the healthcare system, including

traceability, the capability of analysing unstructured data, decision support capability, and predictive capability. The paper considered the graph database as a big data tool since it can effectively deal with complex, densely connected, and semi-structured data. The authors also surveyed a number of related works that exploited graph databases in handling huge amounts of healthcare data.

Currently, quantum chemical calculations are commonly employed to produce large datasets for machine learning. However, these datasets typically focus on equilibrium structures and a few nearby conformers. While exploring potential energy surfaces offers valuable insights into ground and transition states, the analysis becomes complex due to the multitude of potential reaction pathways. Gimadiev et al. [11] introduced a database architecture designed for the storage and analysis of 3D chemical structures, as computed by quantum chemistry. While conventional relational databases can manage data related to individual molecules, they prove to be less suitable for handling chemical reaction data effectively. As a solution, a hybrid database system was proposed that combines the strengths of both graph and relational architectures.

Unal and Oguztuzun [12] studied the migration process from the relational database to the graph database. The reason behind this migration is, according to the paper, due to the increasing sizes of data and the challenge of extracting information from such massive amounts of data. The performance of relational databases declines when the number of connections between entities increases since they use expensive and complicated join statements in querying and accessing data. The authors introduced rules for transforming relational databases into graphs. However, they did not apply these rules to a database nor study the deployment of any graph algorithms after the migration process.

Rapid advancements in experimental and analytical techniques provide diverse information about biological components. Although individual details are important, the fundamental aim of biology is to understand the complex interactions among heterogeneous data contributing to cellular functions. Identifying these intricate relationships is challenging due to their complexity. Yoon et al. [13] clarified the feasibility of using graph databases for representing and storing complex biological data and relationships. They collected diverse biological data, such as gene-disease associations, protein-protein interaction, etc. and compared the performance of My Structured Query Language (MySQL) and Neo4j as a relational database and graph database, respectively. They found that Neo4j outperformed MySQL in all perspectives, particularly query response time and search speed.

Schäfer et al. [14] developed a web application that utilizes Neo4j as a graph database gained by transforming a Structured Query Language (SQL) database. The developed application provided analysis and visualization of patients. In addition, they used NeoDash, a tool used for creating a dashboard in Neo4j, to create a dashboard for visualizing and querying a specific patient or group of patients and to display information such as gender distribution and find specific patterns shared among patients. Some limitations are observed, such as the difficulty of exporting report charts and tables from the dashboard directly. In addition, the paper didn't exploit graph algorithms for observing patterns such as Patients with identical or comparable medical histories. These observations are very useful in pinpointing a medical treatment roadmap for each patient.

Palagashvilia and Stupnikovb [15] propose algorithms for mapping between relational and graph databases, focusing on reversible mappings where the composition of the mappings is identity mapping. These algorithms form a basis for creating multimodal graph-relational database management systems. Specifically, algorithms for relational-to-graph and graph-to-relational mapping of databases have been developed, and the identity of the compositions of these mappings under certain conditions imposed on graph databases has been proved. The algorithms have been implemented in Python for Post Ingres Structured Query Language (PostgreSQL), relational Database Management System (DBMS) and Neo4j graph DBMS. The relational-to-graph mapping has been tested on the Northwind database. However, the authors did not give any evaluation metrics, such as the speed of the mapping process.

The literature presented above studied the transformation of a normalized database into a graph model. Handling denormalized data can lead to inefficiencies, as it often involves issues like redundancy, anomalies, and problems with data dependencies, including transitive and partial dependencies. Such challenges may hinder the maintenance of data integrity. Consequently, the advent of graph database model, with their flexible schemas that permit dynamic changes in data structure and relationships, has gained prominence. This versatility makes it an ideal choice for scenarios involving denormalized data, where schemas might evolve, or entities display a variety of relationships.

However, the conversion of a denormalized database populated with data into a graph model remains a relatively unexplored area in existing research. The process of effectively handling and transforming denormalized databases into graph structures has yet to receive substantial attention within the current literature. Accordingly, the main contributions of this paper are threefold. First, we develop two novel algorithms: the first transforms a denormalized database into a property graph model, and the second populates the property graph model with data.

The performance of the proposed algorithms is evaluated by applying them to a denormalized healthcare database [16], resulting in a fully populated property graph model. The rest of this paper is structured as follows. A theoretical background about relational and graph database models is provided in Section 2. Section 3 presents the proposed algorithms that handle the research problem. Section 4 describes and deliberates the empirical experiment design and results of implementing the proposed algorithms. Finally, the conclusion of this research is discussed in Section 5.

2. Theoretical Foundations on Data Models

This research explores the transformative process of mapping denormalized databases to graph models, a realm less explored in the current literature. This section emphasizes the distinctions between graph and relational data models. Additionally, it briefly clarifies the concepts of normalization and functional dependencies.

2.1. Relational and graph data models

When comparing graph and relational models, several key distinctions emerge. Firstly, in terms of scalability, the graph model demonstrates an advantage with horizontal scaling and partitioning, allowing multiple servers to process graph

queries in parallel. Conversely, the relational model often resorts to vertical scaling, upgrading hardware to handle increased workloads [17]. Performance-wise, graph databases shine with index-free adjacency, enabling swift traversal between related entities without the need for indexes.

In contrast, the relational model relies on index lookups, leading to potential performance bottlenecks. Additionally, graph databases prove more user-friendly in managing connected data, excelling in multi-hop queries with visually expressed relationships using the Cypher query language [18]. Relational databases employing SQL may face challenges in handling complex queries involving multiple joins and nested subqueries, resulting in less intuitive and readable code [19].

As an example, a simple database consisting of three entities is presented in Fig. 1. Part a of the figure represents the relational model of this database, in which the "Actor" and "Movie" entities are connected in a many-to-many relationship, while the "Director" and "Movie" entities are connected in a one-to-many relationship. Mapping this ER diagram to tables in SQL will result in four tables, one for each entity, in addition to the joining table that represents the many-to-many relationship between the "Actor" and "Movie" entities. Conversely, the same database is represented as three labelled nodes: "Movie", "Actor", and "Director", and two relationships, "Actor_Movie" and "Director_Movie". Table 1 gives a brief comparison between the two models in several criteria.

Table 1. Relational model vs. graph model.

Criteria	Relational DB	Graph DB
Format	Tables with rows and columns	Nodes and edges
Relationship	Established using foreign keys between tables	Represented as edges between nodes
Complicated Queries	Require complex joins between tables	Do not require joins
Scalability	Vertical	horizontal
Schema	Rigid	Flexible
Deep analytics	Poor performance	High performance

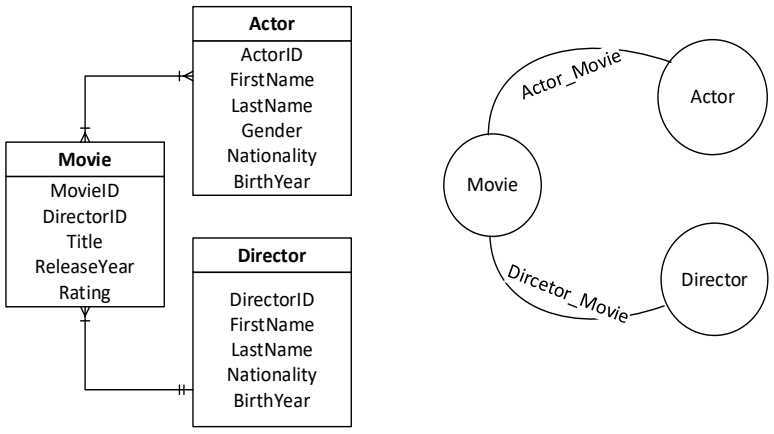

Fig. 1. Movie Database as represented by relational and graph models.

2.2. Database normalization

Normalization is a process that ensures the elimination of redundancy and undesirable characteristics like insertion, update, and deletion anomalies. It involves a systematic process to organize data into small entities and link them together using relationship properties. The normalization process goes through several normal forms, each with specific rules to address a specific type of data dependency. These forms include the First Normal Form (1NF), which is the basic level of database normalization that ensures prohibiting multivalued attributes by representing each attribute in a table as an atomic value.

The second level of normalization is known as the Second Normal Form (2NF), ensuring that non-key attributes are functionally dependent on the entire primary key by cancelling any partial dependencies. However, the Third Normal Form (3NF) is achieved by ensuring that all non-key attributes are not functionally dependent on other non-key attributes, which is performed by removing the transitive dependencies between the non-key attributes [20].

The concept of functional dependency is specific to the relational database model. It illustrates the particular relationships between attributes. This dependency is defined as the value of one attribute uniquely determining the value of another attribute. In other words, if attribute B is functionally dependent on attribute A (denoted as $A \rightarrow B$), that means for a known value of A, there is a unique corresponding value of B in all records [21]. Functional dependency plays a crucial role in the normalization process, through which the candidate key can be identified and the table structure determined [22].

3. Method

This section presents the proposed algorithms that are used to map the denormalized database into a graph model and populate it with data. The algorithms are evaluated using the Synthea dataset, which is described in this section.

3.1. The developed mapping algorithms

Two mapping algorithms, the Property Graph Creation (PGC) and the Property Graph Population (PGP), are developed for transforming a denormalized database into a graph model.

3.2. Property graph creation algorithm

The PGC algorithm transforms a denormalized database into a graph model in three main steps. Initially, it creates a distinct labelled node for each entity found in the original database. Following this, it enforces 1NF on each file, provided it is not already in this form, ensuring data is atomic and without multivalued attributes or repeating groups. The algorithm then examines the functional dependencies, deciding on the mapping of attributes to either node properties or relationship properties based on their relationship with the primary key.

Attributes linked to a composite primary key are carefully distinguished between those dependent on the part of the key versus the entire key, influencing their mapping. Technical primary keys, primarily designed for database management, are excluded from this transformation. This exclusion is intentional,

aiming to declutter the graph model from implementation-specific artefacts, thus enhancing its readability and relevance to domain-specific queries. Foreign keys become the basis for relationships between nodes, facilitating the representation of data interconnectedness within the graph model. This structured approach ensures a comprehensive and accurate transformation from a denormalized database to a graph database model. Before describing the proposed algorithms, formal definitions of the relational and graph database models are provided in the nomenclature in *Appendix I*. The PGC algorithm is provided in *Appendix II*.

3.3. Property graph population algorithm

The result of executing the PGC algorithm is a property graph model comprising nodes, relationships, and properties associated with either nodes or relationships. The decision to maintain transitive dependencies between attributes aligns with the principles of the graph database paradigm, optimizing query performance, simplifying the schema, and enhancing data traversal efficiency. While the property graph model is structured, it lacks actual data. Therefore, the PGP algorithm is introduced to import data from the Comma-Separated Values (CSV) files into the property graph, enabling it to be visualized and utilized with all the features of the graph database. The PGP algorithm is provided in Appendix III. Figure 2 highlights the fundamental steps constituting the PGC and PGP algorithms.

It is worth mentioning that the proposed algorithms can be applied to a normalized database, i.e., a relational database. A normalized database may include joining tables to facilitate the many-to-many relationships between entities. A joining table consists of foreign key columns and optionally additional columns to store metadata or other relevant information. These tables can be mapped according to the PGC algorithm.

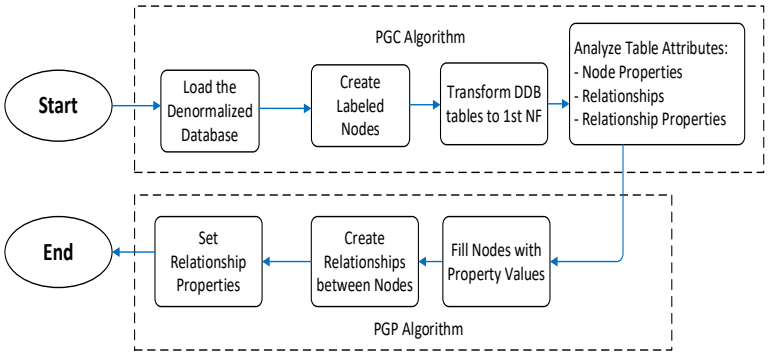

Fig. 2. A block diagram showing the PGC and PGP algorithms.

3.4. Dataset description

As a case study, a sample denormalized database that is generated using Synthea [16] is taken. Synthea is a synthetic patient data generator that provides superior, artificial, genuine-seeming patient data and covers every element of healthcare. The data generated is unbounded by privacy and security limitations. The deployed version, provided in [16], consists of a number of CSV files, each representing different aspects of the simulated healthcare data. The CSV files and their attributes are listed in Table 3, located at the end of this paper.

3.5. Experiment setup

Neo4j version 4.4 is used as a graph database since it is a highly scalable, high-performance database with native graph storage and processing capabilities [23]. Cypher is employed for executing the PGC and PGP algorithms because it is the most widely adopted, fully specified, and open query language for property graph databases. It offers an intuitive and efficient way of working with property graphs, making it ideal for this experiment. All experiments are executed on a laptop with CPU Intel Core i7, four cores, and installed memory (RAM) of 32 GB.

4. Results and Discussion

Several files from Synthea are selected to assess the performance of the proposed algorithms. The first file, "Patients," is already in 2NF. The second and third files, "Allergies" and "Immunizations," are in 1NF due to the presence of partial dependencies. Finally, the "Careplans" file, which contains a technical primary key, would be considered in 1NF if the technical primary key is omitted, as it also exhibits partial dependency.

4.1. Patients

The first file to consider is the "Patients" file since multiple files in the experimented database are connected to this file, i.e., they have foreign keys named patient referencing the primary key of this file [ID]. After analysing this file, it is evident that it is already in 2NF since there are no partial dependencies between its attributes. The file description is as follows:

$A_{Patients}$ = {ID, Birthdate, SSN, First, Last, Race, Ethnicity, Gender, Birthplace, Address, City, State, Healthcare_Expenses, Healthcare_Coverage}
$PK_{Patients}$ = {ID}
$NK_{Patients}$ ={Birthdate, SSN, First, Last, Race, Ethnicity, Gender, Birthplace, Address, City, State, Healthcare_Expenses, Healthcare_Coverage }
$FK_{Patients}$ = ∅

Applying the PGC algorithm results in the following:

1. A node is created and labelled as a *patient*.
2. The functional dependency between attributes is analysed as follows:
   - **a.** $PK_{Patients} \rightarrow NK_{Patients}$
     Therefore, all attributes in the $NK_{Patients}$, in addition to the primary key, are mapped as node properties for the *Patient* nodes.
   - **b.** Since $FK_{Patients}$ = ∅, there are no attributes mapped as relationships.

After applying the PGC algorithm, the PGP algorithm is applied to import the patients' data from the CSV file to the property graph.

4.2. Allergies

The second file to consider is the "Allergies" file. After analysing this file, it is found that the file is in 1NF since there is a partial dependency. The file description is as follows:

$A_{Allergies}$ = {Patient, Encounter, Code, Description, Start, Stop}
$PK_{Allergies}$ = {Patient, Encounter, Code}

$NK_{Allergies}$ = {Description, Start, Stop}
$FK_{Allergies}$ = {Patient, Encounter}

Applying the PGC algorithm results in the following:

1. A node is created and labelled as *Allergy*.
2. Identify the potential dependency between attributes as follows:
   - **a.** *Code* → *Description*
     i.e., partial dependency. Therefore, *Code* and *Description* are mapped as node properties for the *Allergy* nodes.
   - **b.** The *patient* attribute is mapped as a relationship between *Allergy* nodes and *Patient* nodes, whereas the *Encounter* attribute is mapped as a relationship between *Allergy* nodes and *Encounter* nodes.
   - **c.** $PK_{Allergies}$ → *Start*, *Stop*
     Therefore, the *Start* and *Stop* attributes are mapped as relationship properties for the relationships between *Allergy* and *Patient* and between *Allergy* and *Encounter*.

After applying the PGC algorithm, the PGP algorithm is applied to import the allergy data from the CSV file to the property graph. Since the "Allergies" file contains a foreign key referencing the primary key of the "Patients" file, a loading query is executed to match two nodes, *Allergy* and *Patient* and create a relationship labelled as *Patient_Has_Allergy* between them. The corresponding relationship properties are set according to the attribute values in each row within the "Allergies" file.

A sample of the "Allergies" file stored in the denormalized database is presented in Fig. 3. The patient with the ID of '0288abb6-633c-40c3-ba0c-66c7d957727e' had been diagnosed with five types of allergies. This information is mapped so that the node corresponding to this patient has a relationship with five different allergies, as indicated in Fig. 4. The Date on which this patient was diagnosed with a specific type of allergy is indicated in the *Start* attribute, which is mapped as a relationship property, along with the *Stop* attribute, which is null in all five cases.

CODE	DESCRIPTION	Patient	Start	Stop
419474003	Allergy to mould	0288abb6-633c-40c3-ba0c-66c7d957727e	1952-03-10	NULL
232350006	House dust mite allergy	0288abb6-633c-40c3-ba0c-66c7d957727e	1952-03-10	NULL
232347008	Dander (animal) allergy	0288abb6-633c-40c3-ba0c-66c7d957727e	1952-03-10	NULL
418689008	Allergy to grass pollen	0288abb6-633c-40c3-ba0c-66c7d957727e	1952-03-10	NULL
91935009	Allergy to peanuts	0288abb6-633c-40c3-ba0c-66c7d957727e	1952-03-10	NULL

Fig. 3. A sample of the "Allergies" file in the denormalized database.

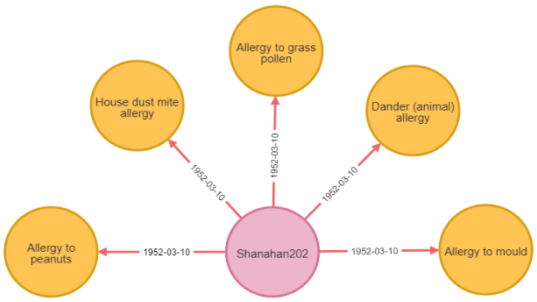

Fig. 4. A subgraph indicating an example of the relationships between the Patient and Allergy nodes.

4.3. Immunization

The third file to consider is the "Immunization" file. After analysing this file, it is found that it is in 1NF since there is a partial dependency between attributes. The file description is as follows:

$A_{Immunization}$ = {Patient, Encounter, Code, Description, Cost, Date}
$PK_{Immunization}$ = {Patient, Encounter, Code}
$NK_{Immunization}$ = {Description, Cost, Date }
$FK_{Immunization}$ = {Patient, Encounter}

Applying the PGC algorithm results in the following:

1. A node is created and labelled as *Immunization*.
2. The functional dependency between attributes is identified as follows:
   - **a.** *Code* → *Description*, *Cost*
     i.e., partial dependency; therefore, *Code*, *Description* and *Cost* are mapped as node properties for the *Immunization* node.
   - **b.** The *patient* attribute is mapped as a relationship between the *Immunization* node and the *Patient* node, whereas the *Encounter* attribute is mapped as a relationship between the *Immunization* node and the *Encounter* node.
   - **c.** $PK_{Immunization}$ → *Date*
     Therefore, *Date* is mapped as a relationship property for the relationships between *Immunization* and *Patient* and between *Immunization* and *Encounter*.

After applying the PGC algorithm, the PGP algorithm is applied to import the immunization data from the CSV file to the property graph. Since the "Immunization" file has a foreign key referencing the primary key of the "*Patients"* file, a loading query is executed for matching two nodes, *Immunization* and *Patient*, to create a relationship named *Patient_Has_Immunization* between them and setting the relationship property *Date* according to the attribute's value in each row within the "Immunization" file. Figure 5 presents a sample of the "Immunization" file as stored in the denormalized database. The patient with ID equals '0288abb6-633c-40c3-ba0c-66c7d957727e' has taken 13 types of immunizations; one of these immunizations has been taken 10 times on different dates. This information is mapped so that the node corresponding to this patient has 13 relationships with four different immunization nodes, as indicated in Fig. 6.

CODE	DESCRIPTION	DATE	Patient
140	Influenza seasonal injectable preservative free	2010-09-15	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2011-09-21	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2012-09-26	0288abb6-633c-40c3-ba0c-66c7d957727e
113	Td (adult) preservative free	2012-09-26	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2013-10-02	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2014-10-08	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2015-10-14	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2016-06-08	0288abb6-633c-40c3-ba0c-66c7d957727e
133	Pneumococcal conjugate PCV 13	2016-06-08	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2017-10-25	0288abb6-633c-40c3-ba0c-66c7d957727e
33	pneumococcal polysaccharide vaccine 23 valent	2017-10-25	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2018-10-31	0288abb6-633c-40c3-ba0c-66c7d957727e
140	Influenza seasonal injectable preservative free	2019-11-06	0288abb6-633c-40c3-ba0c-66c7d957727e

Fig. 5. A sample of the "Immunization" file in the denormalized database.

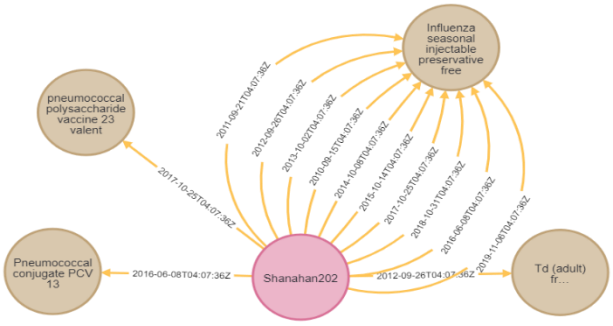

Fig. 6. A subgraph indicating the relationship between the patient and the Immunization nodes.

4.4. Careplans

The "Careplans" file has a technical primary key. This attribute is ignored when mapping the file into a property graph. The file description is as follows:

$A_{Careplans}$ = {ID, Patient, Encounter, Code, Description, Reason_Code, Reason_Description, Start, Stop }
$TPK_{Careplans}$ = ID
$PK_{Careplans}$ = {Patient, Encounter, Code}
$NK_{Careplans}$ = {Description, Reason_Code, Reason_Description, Start, Stop }
$FK_{Careplans}$ = {Patient, Encounter}

Applying algorithm [1] results in the following:

1. A node is created and labelled as *Careplan*.
2. The functional dependency between attributes is analysed as follows:
   - **a.** *Code* → *Description, Reason_Code, Reason_Description*
     i.e., partial dependency; therefore, *Code*, *Description*, *Reason_Code*, and *Reason_Description* attributes are mapped as node properties for the *Careplan* node.
   - **b.** The *patient* attribute is mapped as a relationship between *Careplan* and *Patient*, whereas the *Encounter* attribute is mapped as a relationship between *Careplan* node and *Encounter* node.
   - **c.** $PK_{Careplans}$ → *Start*, *Stop*.
     Therefore, *Start* and *Stop* attributes are mapped as relationship properties for the relationships between *Careplan* and *Patient* and between *Careplan* and *Encounter*.

After applying the PGC algorithm, the PGP algorithm is applied to import data from the "Careplans" file to the property graph. Since *Careplan* has a foreign key referencing the primary key of the "*Patients*" file, a loading query is executed for matching two nodes, *Careplan* and *Patient* to create a relationship *Patient_Has_Careplans* between them and setting the relationship properties *Start* and *Stop* according to the attribute's value in each row within the *Careplans* file.

Figure 7 represents a sample of the "Careplans" file as stored in the denormalized database. The patient with ID equals '0288abb6-633c-40c3-ba0c-66c7d957727e' has five careplans. This information is mapped so that the node

corresponding to this patient has five relationships with five different careplans nodes, as indicated in Fig. 8.

CODE	DESCRIPTION	START	STOP	Patient
408869004	Musculoskeletal care	2019-05-11	2019-06-15	0288abb6-633c-40c3-ba0c-66c7d957727e
699728000	Asthma self management	1952-12-04	NULL	0288abb6-633c-40c3-ba0c-66c7d957727e
225358003	Wound care	2017-07-29	2017-08-19	0288abb6-633c-40c3-ba0c-66c7d957727e
47387005	Head injury rehabilitation	2011-11-27	2012-01-26	0288abb6-633c-40c3-ba0c-66c7d957727e
384758001	Self-care interventions (procedure)	1952-02-21	NULL	0288abb6-633c-40c3-ba0c-66c7d957727e

Fig. 7. A sample of the "Careplans" file in the denormalized database.

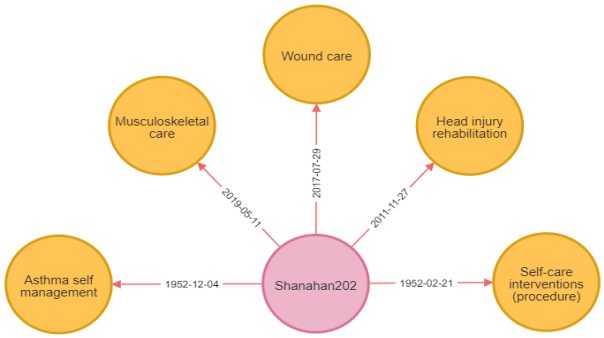

Fig. 8. A subgraph indicating the relationships between the *Patient* and *Careplan* nodes.

Basically, the "Careplans" file has 3,484 rows. Applying the PGP algorithm results in a 32 Careplan nodes in addition to 3,484 relationships between the *Careplan* and the *Patient* nodes, as well as 3,484 relationships between the *Careplan* and the *Encounter* nodes. Table 2 shows the number of nodes created for each entity when mapping the Synthea database, as well as the time required to populate these nodes with property values imported from the CSV files. The time varies due to the differing number of properties each node possesses.

Table 2. Node count and mapping time for populating entities from the Synthia database.

Entity	No. of Nodes	Population Time (ms)
Allergies	15	120
Careplans	32	142
Conditions	32	142
Devices	78	34
Encounters	53,346	1169566
Imaging Studies	9	70
Immunizations	18	239
Medications	131	1300
Observations	125	9112
Organizations	1,119	234
Patients	1,171	299
Payers	10	45
Procedures	137	1257
Providers	5,855	4402

Table 3 provides a comprehensive description of each CSV file within the Synthea Database, detailing whether it contains a technical primary key and specifying which attributes are mapped as node properties, relationships, or relationship properties. For instance, the *Patient* attribute functions as a foreign key referencing the ID attribute in the "Patients" file, while the *Encounter* attribute serves as a foreign key referencing the ID attribute in the "Encounters" file. Notably, both the "Careplans" and "Imaging Studies" files feature technical primary keys, which are excluded during the mapping process into a property graph. All CSV files include attributes that are mapped as node properties. Alternatively, the "Organizations", "Patients", and "Payers" files do not contain any attributes mapped as relationships, indicating the absence of foreign keys within these files. Moreover, certain files contain attributes mapped as relationship properties.

Table 3. Synthea database with its tables and attributes and their mapping into a graph database model.

CSV file name	Tech. PK	Attributes mapped as node properties	Attributes mapped as relationships	Attributes mapped as relationship properties
Allergies	-	Code, Description	Patient, Encounter	Start, Stop
Careplans	✓	Code, Description, Reason_Code, Reason_Description,	Patient, Encounter	Start, Stop
Conditions	-	Code, Description	Patient, Encounter	Start, Stop
Devices	-	UDI, Code, Description	Patient, Encounter	Start, Stop
Encounters	-	ID, Code, Description, EncounterClass, Base_Encounter_Cost, Total_Claim_Cost, Start, Stop	Patient, Organization, Provider Payer	- PayerCoverage
Imaging Studies	✓	BODYSITE_CODE, BodySiteDescription, Modality_Code, Modality_Description, SOPCode, SOPDescription	Patient, Encounter	Date
Immunizations	-	Code, Description, Cost	Patient, Encounter	Date
Medications	-	Code, Description, Base_Cost, Dispenses, TotalCost	Patient, Encounter Payer	Start, Stop Payer_Coverage
Observations	-	Code, Description, Value, Units, Type	Patient, Encounter	Date
Organizations	-	ID, Name, Address, City, Revenue, Utilization	-	-
Patients	-	ID, Birthdate, SSN, First, Last, Race, Ethnicity, Gender, Birthplace, Address, City, State, Healthcare_Expenses, Healthcare_Coverage	-	-
Payers	-	ID, Name, Address, Amount_Covered	-	-
Procedures	-	Code, Description, Base_Cost	Patient, Encounter	Date
Providers	-	ID, Name, Gender, Address, Utilization	Observation	-

5. Conclusions

In this work, two algorithms that offer a transformative approach for converting a denormalized database into a graph database model are proposed: PGC and PGP. The PGC algorithm constructs a property graph model framework comprising nodes, relationships, and associated properties but devoid of actual data. Conversely, the PGP algorithm is tailored to import data from CSV files into the property graph model generated by PGC, thereby populating it with data and rendering it suitable for visualization and utilization within the graph database. The proposed algorithms were applied to Synthea, a denormalized database consisting of several CSV files. Some files are already in 2NF, while others are in 1NF and require analysis to identify the functional dependency between the primary key and non-key attributes. The PGC and PGP algorithms are executed sequentially, resulting in a graph database containing approximately 72,000 nodes and 880,000 relationships.

A promising path of research involves applying various graph algorithms applicable to the problem domain. By leveraging algorithms that accommodate the inherent nature of the data, such as similarity algorithms, valuable insights can be extracted to optimize data analysis processes. This exploration into graph algorithm applications stands as a key direction for enhancing the utility and efficiency of the graph database solution.

Nomenclatures

A_t	Set of attributes "a" of *t*, $t \in$ DDB $A_t = \{a_1, a_2, \ldots, a_k\}$, *k: no. of attributes in t.*
c	The number of CSV files
DDB	The set of CSV files $\{t_1, t_2, \ldots, t_c\}$
FK_t	Set of foreign key attributes, $FK_t \subseteq A_t$
k	The number of attributes in t.
M_t	Set of multivalued attributes, $M_t \subseteq A_t$, $M_t = \{m_1, m_2, \ldots, m_z\}$, *z: no. of multivalued attributes in* t_i., $m = \{v_1, v_2, \ldots, v_e\}$, *e*: no. of values of *m*.
NK_t	Set of non-primary key attributes, $NK_t \subseteq A_t$, $A_t = PK_t \cup NK_t$
NFK_t	Set of non-foreign key attributes, $NFK_t \subseteq A_t$
PK_t	Primary Key, $PK_t \subseteq A_t$
Rel	Set of relationships between nodes in $\mathcal{N}$, $rel_{s,d} \in$ Rel, $s,d \in \mathcal{N}$.
R_t	$R_t = \{ row_1, row_2, \ldots, row_j\}$, *j*: number of rows of table *t.*
row	$row = [value_{a1}, value_{a2}, \ldots, value_{ak}]$
TPK_t	Technical Primary Key, $TPK_t \in A_t$
$\mathcal{N}$	Set of nodes composing the property graph
$\mathcal{N}\_Pro_n$	Set of node properties of *n*.
$\mathcal{R}\_Pro_{s,d}$	Set of relationship properties between *s* and *d*, *s* and $d \in \mathcal{N}$, $\mathcal{R}\_Pro_{s,d}$, $\mathcal{R}\_Pro_{s,d} \cap \mathcal{N}\_Pro_s \cap \mathcal{N}\_Pro_d = \emptyset$.
$\rightarrow$	Functional Dependency
$\Rightarrow$	Mapping attributes into a node, relationship, or property

Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
CSV	Comma-Separated Values
DBMS	Database Management System
EHR	Electronic Health Records
MySQL	My Structured Query Language
PGC	Property Graph Creation
PGP	Property Graph Population
Postgre SQL	Post Ingres Structured Query Language

References

1. Chen, Q. (2020). Research on the implementation method of database security in management information system based on big data analysis. *E3S Web of Conferences*, 185, 02033.
2. Beath, C.M.; and Orlikowski, W.J. (1994). The contradictory structure of systems development methodologies: Deconstructing the IS-user relationship in information engineering. *Information Systems Research*, 5(4), 350-377.
3. Pearlson, K.E.; Saunders, C.S.; and Galletta, D.F. (2023). *Managing and using information systems*: *A strategic approach*. John Wiley & Sons.
4. Kim, H-J.; Ko, E-J.; Jeon, Y-H.; and Lee, K-H. (2020). Techniques and guidelines for effective migration from RDBMS to NoSQL. *The Journal of Supercomputing*, 76(10), 7936-7950.
5. Timón-Reina, S.; Rincón, M.; and Martínez-Tomás, R. (2021). An overview of graph databases and their applications in the biomedical domain. *Database*, 2021, baab026.
6. Michail, D.; Kinable, J.; Naveh, B.; and Sichi, J.V. (2020). JgraphT-A Java library for graph data structures and algorithms. *ACM Transactions on Mathematical Software*, 46(2), 1-29.
7. Kotiranta, P.; Junkkari, M.; and Nummenmaa, J. (2022). Performance of graph and relational databases in complex queries. *Applied Sciences*, 12(13), 6490.
8. Albarak, M.; Bahsoon, R.; Ozkaya, I.; and Nord, R.L. (2020). Managing technical debt in database normalization. *IEEE Transactions on Software Engineering*, 48(3), 755-772.
9. Vellido, A. (2020). The importance of interpretability and visualization in machine learning for applications in medicine and health care. *Neural Computing and Applications*, 32(24), 18069-18083.
10. Deghmani, F.; AmineAmarouche, I.; and Boukhalfa, K. (2021). Applications of graph databases and big data technologies in healthcare. *Information Processing at the Digital Age Journal*, 26(1), 47-57.
11. Gimadiev, T.; Nugmanov, R.; Batyrshin, D.; Madzhidov, T.; Maeda, S.; Sidorov, P.; and Varnek, A. (2020). Combined graph/relational database management system for calculated chemical reaction pathway data. *Journal of Chemical Information and Modeling*, 61(2), 554-559.
12. Unal, Y; and Oguztuzun, H. (2018). Migration of data from relational database to graph database. *Proceedings of the* 8th *International Conference on Information Systems and Technologies*, Istanbul, Turkey.

13. Yoon, B-H.; Kim, S-K.; and Kim, S-Y. (2017). Use of graph database for the integration of heterogeneous biological data. *Genomics & Information*, 15(1), 19-27.
14. Schäfer, J.; Tang, M.; Luu, D.; Bergmann, A.K.; and Wiese, L. (2022). Graph4Med: A web application and a graph database for visualizing and analyzing medical databases. *BMC Bioinformatics*, 23(1), 537.
15. Palagashvilia, A.M.; and Stupnikovb, S.A. (2023). Reversible mapping of relational and graph databases. *Pattern Recognition and Image Analysis*, 33(2), 113-121.
16. Synthea. Synthetic patient generation. Retrieved March 8, 2024, from https://synthetichealth.github.io/synthea/.
17. Tomar, D.; Bhati, J.P.; Tomar, P.; and Kaur, G. (2019). *Migration of healthcare relational database to NoSQL cloud database for healthcare analytics and management*. In Dey, N.; Ashour, A.S.; Bhatt, C.; and Fong, S.L. (Eds.), *Healthcare Data Analytics and Management*. Elsevier, 59-87.
18. Das, A.; Mitra, A.; Bhagat, S.N.; and Paul, S. (2020). Issues and concepts of graph database and a comparative analysis on list of graph database tools. *Proceedings of the* 2020 *International Conference on Computer Communication and Informatics* (*ICCCI*), Coimbatore, India.
19. Samaan, S.S.; and Jeiad, H.A. (2023). Architecting a machine learning pipeline for online traffic classification in software defined networking using spark. *IAES International Journal of Artificial Intelligence* (*IJ-AI*), 12(2), 861-873.
20. Rajendran, R.K.; and Mohana Priya, T. (2023). *Designing an Efficient and Scalable Relational Database Schema*: *Principles of Design for Data Modeling*. In Sugumaran, V. (Ed.), *Advances in Systems Analysis*, *Software Engineering, and High Performance Computing*. IGI Global, 168-176.
21. Croock, M.S.; Hassan, Z.A.; and Khuder, S.D. (2021). Adaptive key generation algorithm based on software engineering methodology. *International Journal of Electrical and Computer Engineering* (*IJECE*), 11(1), 589-595.
22. Link, S.; Koehler, H.; Gandhi, A; Hartmann, S; and Thalheim, B. (2023). Cardinality constraints and functional dependencies in SQL: Taming data redundancy in logical database design. *Information Systems*, 115, 102208.
23. Chen, J.; Song, Q.; Zhao, C; and Li, Z. (2020). *Graph database and relational database performance comparison on a transportation network*. In Singh, M.; Tyagi, V.; Gupta, P.K.; Flusser, J.; Ören, T.; Cherif, A.R.; and Tomar, R. (Eds.), *Advances in computing and data sciences*. Springer Nature Switzerland.

Appendix A

The PGC Algorithm

Algorithm 1: Property Graph Creation [PGC]
Input: a denormalized database DDB.
Output: a property graph.
Step 1: **for each** $t \in$ DDB, **create** a labeled node $n \in \mathcal{N}$.
Step 2: */* Transform DDB files to 1st NF */*
for each *row* $\in R_t$ **where** $M_t \neq \emptyset$
for each m $\in M_t$
$\overline{R}_t = \{[value_{a1}, value_{a2,\ldots,}value_{ak-1}, v_1], [value_{a1}, value_{a2,\ldots,}value_{ak-1}, v_2],\ldots., [value_{a1}, value_{a2,\ldots,}value_{ak-1}, v_e]\}$
$R_t = R_t \setminus \{row\}$
$R_t = R_t \cup \overline{R}_t$
Step 3: **for** $t \in$ DDB
create $\mathcal{N}$_Pro$_n$
create $\mathcal{R}$_Pro$_{s,d}$ /* set of relationship properties between *s* and *d* */
create Rel
for $a_j \in PK_t$, $a_k \in NK_t$, $a_j, a_k \in A_t$
/ Find the functional dependency between NK_t and PK_t */*
If PK_t **is** non-composite
$PK_t \rightarrow a_k$
$\mathcal{N}$_Pro$_n$.add[$A_t \setminus FK_t$]
If PK_t **is** composite
$a_j \rightarrow a_k$
$\mathcal{N}$_Pro$_n$.add[a_j, a_k]
else if $PK_t \rightarrow a_k$
$\mathcal{R}$_Pro$_{s,d}$.add[a_k]
for $a \in FK_{ti}$, $a \equiv PK_{tj}$, $t_i, t_j \in$ DDB
$a \Rightarrow rel_{s,d}$, $s,d \in \mathcal{N}$
$rel_{s,d}$.add[REL]

Appendix B

The PGP Algorithm

Algorithm 2: Property Graph Population [PGP]
Input: a property graph.
Output: a populated property graph.
Step 1: **for** $t \in$ DDB
Load *t* as Row_t
for each *row* $\in Row_t$
match $n \equiv t$
for each $p \in \mathcal{N}$_Pro$_n$, $n \in \mathcal{N}$
Set $p = row.value_{ax}$, $p \equiv a_x$
Step 2: */* only load CSV files that have foreign key[s] */*
for $t \in$ DDB **where** $FK_t \neq \emptyset$
Load *t* as row
for each *row* $\in Row_t$
match *s* where $p = row.value_{ai}$, $p \in \mathcal{N}$_Pro$_s$, $p \equiv a_i$, $s \in \mathcal{N}$
match *d* where $q = row.value_{aj}$, $q \in \mathcal{N}$_Pro$_d$, $q \equiv a_j$, $d \in \mathcal{N}$
create $rel_{s,d}$, $rel_{s,d} \in$ Rel
Step 3: **for each** $p \in \mathcal{R}$_Pro$_{s,d}$
set $p = row.value_{ak}$, , $p \equiv a_k$