# Comparison of Graph- and Collection-Based Representations of Early Modern Biographical Archives

Meadhbh Healy Thomas O'Connor
John Keating

Maynooth University
Maynooth, Ireland

## Abstract

The ingestion and digital storage of historical records has had a profound impact on scholarly practices. Yet in order for digitized archives to be used to full advantage, it is imperative that they are searchable and organized in a coherent and consistent way. These requirements are particularly evident in the case of historical records pertaining to real-world individuals: personal data is likely to be intricate and may originate from disparate sources, whose rules of data collection and data storage vary greatly. While this makes graph-oriented databases a very attractive option for storing historical records due to their emphasis on attributes and relationships, document-oriented databases may offer similar advantages in terms of flexibility and precision of record storage. In the present paper, both kinds of database are analyzed and compared in terms of the ease of ingestion and accuracy of record presentation within the database. The historical data used was gathered from a number of diverse collections of historical records referring to persons of Irish descent who served in European armies or studied at European universities between the sixteenth and the nineteenth century.

In: Tara Andrews, Franziska Diehr, Thomas Efer, Andreas Kuczera and Joris van Zundert (eds.): Graph Technologies in the Humanities - Proceedings 2020, published at http://ceur-ws.org.

# 1 Introduction

## 1.1 Overview

The data hosted on the Virtual Research Environment (VRE) of the Irish in Europe Project was collected between 2001 and 2008 by researchers at the Universities of Leuven, Oxford, Toulouse, Dublin (Trinity College), Madrid (Complutense), and Maynooth. Their work was part of a coordinated effort to capture and host biographical material, held mainly in European archival repositories, on Irish soldiers, merchants, and clerics in Europe and the European empires in the early modern period (1550–1800). In total, basic biographical records on about 15,000 individuals were identified and harvested for hosting in a virtual research environment that would facilitate advanced querying, graphical representation, and mapping. The assembled material provides us with access to information about a specific migrant population and its evolution over a significant time scale. More importantly, within their digital environment, these sources help to deepen our understanding of early modern migrant populations in general, while serving as a template for the reconstitution of parallel migrant populations. They also open up, for the first time, the possibility of quantitative, as well as qualitative, comparative studies.

One of the most crucial issues facing the digital humanities, and the Irish in Europe Project VRE specifically, is the ingestion of 'unstructured' data into a digital database and the manipulation thereof. By its very nature, historical data can consist of uncertain and/or divergent primary source material, which is why systems centered on modeling and representing personal historical data are often extensive, complex, and heterogeneous (Mosquera and Piedra, 2017). We have found that while these complications can be resolved to a certain extent by using NoSQL collection-based systems, such as MongoDB, a great deal of redundancy remains. Similarly, ingestion and search involve complex and time-consuming algorithms. We believe that graph databases provide a more elegant solution to these problems, as they enable efficient storage of data with intricate relationships and dynamic schema. As graph databases are not rigid in their structure and organization, it is important to consider integrity constraint (IC) support to ensure that data insertion and other processes are performed in such a manner as to ensure that data integrity is not affected. For the purposes of this study, we will focus on providing a preliminary overview of the graph-oriented database Neo4j and a comparison with the most prolific NoSQL database, MongoDB, using the prosopographical data described above. We will compare ingestion and search performances for the collection and graph systems together with an evaluation of schema and IC creation approaches for both systems.

## 1.2 Topics to Be Addressed

NoSQL – commonly referred to as 'Not Only SQL' (with SQL standing for Structured Query Language) – databases have been developed as a response to the limitations of existing relational database management systems (RDBMS). While traditional RDBMS are capable of large-scale data management for structured data, NoSQL databases adeptly manage large amounts of structured, unstructured, semi-structured, and hybrid data at reduced complexity and cost (Mohmmed and Osman, 2017). Yet graph databases, a particular form of NoSQL databases, are becoming increasingly significant in real-world applications: they provide an adequate framework for representing complex relationships in diverse datasets, which can lead to the discovery of causal relationships by combining disparate sources of information (Le May et al., 2020). With the development of graph databases it has become possible to combine the performance of NoSQL databases and the representativity of graphs (Castelltort and Martin, 2018).

NoSQL databases vary greatly in how they store and manage data. One of the primary distinctions is that in relational models the schema is extremely rigid and possible relationships are fixed in advance. In graph databases, the information is stored in a schemaless format, (key-value pairs), which allows several related values to be stored at the same node (Čerešňák and Kvet, 2019). When it comes to archiving historical data in particular, many NoSQL databases provide a distinct advantage over SQL databases. This is again due to the rigidity of the SQL schema, which is inferior where the modeling and absorbing of unstructured historical data is concerned. Both types of database compared in this paper are NoSQL; MongoDB is the most prominent collection-based database available at present, and Neo4j is a top performing graph database.

## 1.3 Motivation

One of the immediate incentives for historians and digital archivists who wish to digitize historical data is that the archives become more accessible instead of remaining in storage with no public access. The immense changes taking place in the past number of years have begun to reshape digital curation and digital historiography, as detailed extensively by Sabharwal (Sabharwal, 2015). It is imperative for history scholars and academics, however, that certain precautions be taken when reproducing a historical collection of miscellaneous nature in a digitized format, in order to precisely replicate the original (Borissova, 2018).

In particular, this means detailing relationships between the information chunks of a record, as well as delineating common relationships between records in a precise manner. This is a significant challenge when compiling a database of historical data, as it is easy to convolute multiple records from differing datasets if they are not correctly represented. What results is an interpolation problem, as it must be determined how best to reproduce intricate, heterogeneous data without loss of complexity, making an analysis of integrity constraint support vital for digital archivists.

The data being examined in this study originally formed part of a now obsolete and no longer publicly accessible VRE website where each dataset was modeled, and each record stored, as an intricate XML file. XML databases could not be considered due to the volume of data, as the syntactical redundancy of XML would have resulted in a prohibitively large transport and storage effort. Moreover, the records collected have been archived in multiple different languages, which means that it is entirely possible for a military record in the French collection, for example, to have details listed in English, Irish (Gaelic), Latin, and French. Given that any or all of the names Sean De Paor, John Power, or Jean LePoer could be cataloged in a record for the same individual, querying a database for precisely one of the above would not return results for the others. This is why a database that permits regex (short for regular expression) queries, which allow the fields being searched to be matched to a query pattern, is an essential requirement. XML databases are limited in this regard, as nested tags are very difficult to manage with regular expressions (Taktek and Thakker, 2020). Due to the way information is stored in MongoDB and Neo4j, the two platforms are much better suited to the use of regular expressions.

## 1.4 Problem Statement

Integrity constraints are rules that restrict the information that may be present in the database. They play a major role in maintaining the precise structure of a record or set of records. The constraints in SQL databases can be divided into two main components: entity integrity, which tests the validity of the data by providing primary keys; and referential integrity, which adds meaningful structure to the data by using foreign keys to tie relations together (Bono, 2007). As referenced previously, many NoSQL databases suffer from a lack of strictly designed schema structure, which can impede any prospect of securely defining and confirming the rules of referential integrity, and can result in a lower degree of control over data values (Bjeladinovic et al., 2020). However, this is not the case in graph-oriented databases such as Neo4j, where referential integrity is maintained by establishing a relationship between two nodes. Much of the support for data integrity is solely available in SQL databases as it is implemented in the SQL language, and this can leave non-relational databases at a severe disadvantage when attempting to maintain data integrity. The justification for this is easy to fathom: by deliberately maintaining low-level systems in a NoSQL environment, it is possible to create, store, and analyze vast quantities of data at high speeds. In order to obtain higher data consistency and an increase in reliability of NoSQL applications, transactional services, which vary according to the NoSQL system, have been developed (González-Aparicio et al., 2018). Various measures can be implemented on the user application side, including designing test cases which check various possibilities during the execution of a transaction in order to detect potential faults or inconsistencies in the data (Agnelo et al., 2020).

## 1.5 Approach

When processing large amounts of real-world data, entities may be represented in a variety of formats, such as JSON (JavaScript Object Notation) or relational records, and representations may contain redundant or inconsistent information (Simonini et al., 2019). Data entity types are another important issue, as they prevent certain schematic anomalies – for example, it should not be possible to enter a string into an integer-specific column and vice versa (Šestak et al., 2016). Several distinct factors can be identified when discussing data integrity in graph database models (Angles, 2012):

- schema-instance consistency, which prevents incomplete or inconsistent data from being inserted into the database and ensures that each entity can only have the attributes and relations previously established in the desired schema
- node or edge identity, which demands that each value in the database can be identified by a value, such as name or id, or the values of its attributes
- cardinality integrity, which stipulates that each node in the database has a unique identifier which is the equivalent of a primary key constraint in relational databases
- referential integrity, which requires that only existing nodes in the database can be referenced, similar to foreign key constraint
- functional dependencies, which tests if an entity determines the value of another database entity
- graph pattern constraints, which identify structural restrictions such as path constraints in the data

The elimination or control of data redundancy, which would decrease the volume of superfluous information stored in the database, is another important consideration. For the purpose of archiving historical data, functional dependencies are largely not a priority – as the data is static, the entities do not depend on each other and the values do not change. However, as will become clear over the course of this paper, the other factors listed above are all crucial when it comes to ensuring adequate referential integrity constraint support. In order to determine the optimal method of archiving historical data-sets, a comparison test was conducted between two types of NoSQL databases, Neo4j and MongoDB. In MongoDB, a document-oriented database which processes semi-structured data, each record and its associated attributes are considered a document. MongoDB stores data as a hierarchy of key-value pairs and provides a rich query language allowing for easier transition from relational databases. Neo4j is a native graph database which is geared towards the storage and processing of graphs, and allows the management of interconnected data. Graph databases help find relationships between data, and have index-free adjacency. This means that while relational or other non-native graph databases have central indexes and processing overheads with every index lookup, graph databases load every relationship associated with an entity when a node is accessed (Henderson, 2020).

## 1.6 Metrics

In order to accurately compare the performance of MongoDB against Neo4J, the research has been divided into three distinct categories:

- to observe and determine the difficulty, or lack thereof, of ingesting a record into both types of database, for an individual record and a set of records
- to examine the structure of the data in both sets of software, placing particular emphasis on the precision of conversion from the XML record to its reproduction within the database and the lack of difficulty of merging records between datasets
- to analyze the complexity at which an individual record can be extracted from both types of database

The databases were evaluated according to both shared and unique features.

## 1.7 Data

The data being examined and archived comes from a variety of sources. It is diverse, real world data that is both complex and heterogeneous. The library containing the data consists of five primary components:

- Brockliss & Ferté; this data was gathered by historians Laurence Brockliss and Patrick Ferté in collaboration with institutions in Oxford and Toulouse, and details Irish students that studied in Irish colleges in Europe in the 1700s and 1800s
- Spain; records of Irish soldiers serving in Spain during the seventeenth and eighteenth century, collected by Óscar Recio Morales of the Complutense Institute for International Studies.
- France; records of Irish regiments in France in the 1700s and 1800s, collected by Colm Ó Conaill of Trinity College Dublin
- King's Inns; records of Irish students who attended King's Inns in the pre-Cromwellian era, archived by Brid McGrath of Trinity College Dublin

Each dataset contains intricate and varied records, which can make it difficult to establish a comprehensive pattern. On the other hand, as mentioned above, the records are currently stored in XML files, and the fact that each class of data has already been resolved into elements and attributes is of great help when it comes to examining the structure of an individual record.

## 2 Technical Background

### 2.1 Topic Material

Neo4j and MongoDB are compared in terms of both speed of processing and the formation of the syntax necessary to create a query. As stated above, the records that are being processed are complex and involve nested data, which makes it advisable to create an encoding of each data library before they are absorbed by the software. The model is based on the structure of the XML file of each collection, and therefore no assumptions were made in transforming the records to the model produced. A parent-child model of the Brockliss & Fertè dataset is shown in Figure 1.

### 2.2 Technical Material

MongoDB (as well as Neo4j) does not use Structured Query Language (SQL) to interact with the database, but it is compatible with a number of languages, including Go, C++, and Python. MongoDB's primary querying language is JavaScript.[1] The documents are hierarchical tree data structures which can consist of maps, collections, and primitive values. MongoDB uses the following hierarchy: database, collection, and document (Mahajan et al.,

[1]When commands are composed in this language, it creates a JSON document, BSON (Binary JSON) object, or sub-documents, which are the primary components of collections in the database (Jose and Abraham, 2020).

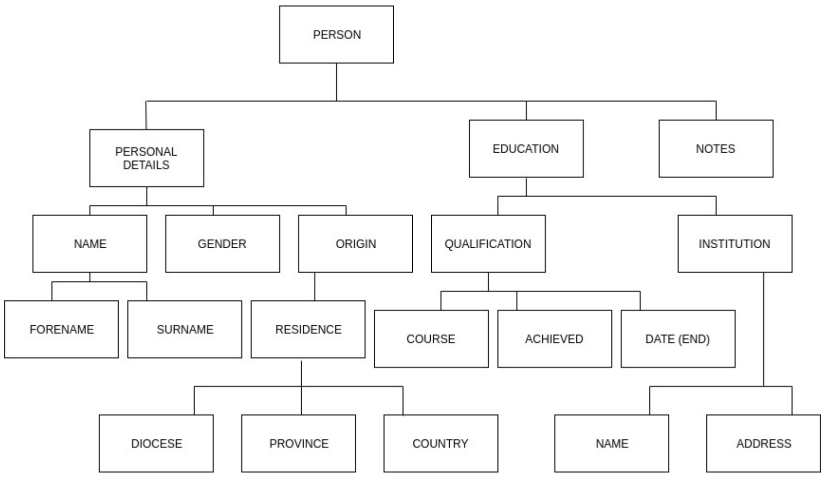

Figure 1: Parent-child model of Brockliss & Fertè military records

2019). There are a number of bespoke querying languages that can be used in conjunction with Neo4j, the most popular of these being Cypher. The syntax of Cypher is similar to SQL and uses an ASCII-Art syntax to denominate patterns. Ideally, Cypher queries are constant strings, so they can be cached by the database as compiled queries. The returned entities can be nodes with all attributes, selected attributes, or aggregated data, depending on the query (Holzschuher and Peinl, 2016). Both types of database are fully ACID compliant.[2] ACID properties guarantee that data integrity is maintained for every transaction in the database. This ensures strict consistency in the sense that all read operations must return the precisely same entities as the latest completed write operation (Lotfy et al., 2016).

## 3 Software and Syntax Analysis

In MongoDB, records are stored in collections. These are analogous to tables in relational databases, although the implementation and operational concept of MongoDB and RDBMS is different. Unlike in SQL databases, the schema of a table does not have to be determined before inserting data. For MongoDB, it is not imperative for all records within a collection to have the same schema, which makes it possible to change the framework

[2]ACID stands for atomicity, which means that a database transaction must be entirely finished, or it will not commit; consistency, which means that a database must remain consistent before, during, and after the transaction occurs; isolation, which means that when multiple transactions are executed simultaneously, transactions are processed exclusively and consecutively – data from one transaction cannot be transferred to another when the transaction has not been completed; and durability, which indicates that once database transactions are registered, events are recorded to a permanent medium which will not be modified outside a transaction.

of a schema within a collection by adding, removing, or updating fields (Čerešňák and Kvet, 2019). Creating an entity-relationship model is vital in order to display the record or object nesting correctly (Edward and Sabharwal, 2014). It can assist in illustrating the embedding and denormalization necessary for scaling the data in MongoDB. Denormalization may be thought of as tables being refined and transformed into secondary simplified data structures, where redundancy is regulated in order to optimize performance. Denormalization allows the data to be wholly retrieved without using a join (Kingdon et al., 2016). To insert a record, the following syntax is used:

```
db.Brocklissferte.insert({ id : "1",
    personalDetails : {"name" :
        {"forename" : "Dionysius", "surname" : "O'Beirne"}
        gender : "male",
        origin : {"residence" : {"address" :
            "diocese" : "Ardagh",
            "province" : "Armagh",
            "country" : "Ireland"}}}},
        education : {"qualification" : {"course" :
            "received the four minor orders",
            "achieved" : "yes",
            "date" : {"end" : "1771-05"},
            "institution" : {"name" : "University of Paris",
                "address" :
                {"town" : "Paris",
                "country" : "France"}}}},
        notes : {"note" : "Boyle, 'St. Nic.', p. 490;
        Boyle, I.C.P., p. 200."}})
```

The fields of the schema can be then adjusted for different records within a collection (Figure 2).

Graph databases, such as Neo4j, provide the most sophisticated and evolved method of data modeling, making it easy to update the schema according to the user's needs (Perçuku et al., 2017). Normalization and denormalization are largely redundant here, as graph databases provide as much or as little structure as the data requires. Traditional SQL databases have rigid schema and a convoluted schema migration process, making the creation and continuous ingestion of records a challenging exercise, particularly for highly relational data (Schulz et al., 2016). Alternatively, graph databases can be employed as an ideal method of managing highly connected data, as they prioritize the modeling and retrieval of relationship-rich data (Pokorný

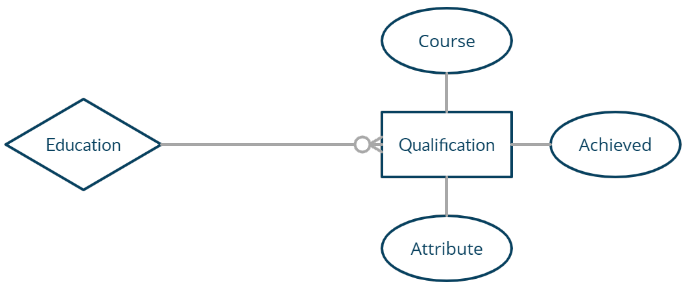

Figure 2: E-R One-to-many diagram

et al., 2017). In fact, once a comprehensive graph data model which incorporates the complex nested fields present in each particular record has been composed, it can be absorbed directly into the database given that the structure of the graph model corresponds exactly to the structure of the schema within Neo4j (Vágner, 2018). This then automatically generates a Cypher command (the domain language of Neo4j), which can be run in the database to create the schema. The following Cypher command was used to create the schema in the Brocklissferte database:

```
CREATE('0':Person),
    ('1':personalDetails),
    ('2':Name {forename:'$forename,',
    surname: '$surname'}) ,
    ('3' :gender {gender:'$male'}) ,
    ('4' :origin),
    ('5':residence {diocese:'$county,',province:
'$province,',country:'$country'}) ,
    ('6':Education),
    ('7':Qualification
        {course:'$course,',achieved:'$achieved,',
        date: '$date'}),
    ('8':Institution
    {institutionName:'$institutionName,',
    address:'$address'}),
    ('9':Notes {notes:'$notes'}) ,
    ('1')-[:'INDIVIDUAL' ]->('2'),
    ('0')-[:'DESCRIPTION' ]->('1'),
    ('1')-[:'RELATED\_TO' ]->('3'),
    ('1')-[:'HOMEPLACE' ]->('4'),
    ('4')-[:'ADDRESS' ]>('5'),
    ('0')-[:'STUDIES']->('6'),
    ('6')-[:'QUALIFIED']->('7'),
    ('6')-[:'ATTENDED']->('8'),
    ('0')-[:'EXTRAINFO' ]->('9')
```

Although graph databases are constantly under development in order to increase their stability and expand their range of features, they have not yet reached the maturity level of other data management solutions, such as relational databases. Nonetheless, a number of mechanisms exist which can help to increase the applicability of graph database technology in a real-world context. In some instances, traditional solutions developed for relational databases can be reworked and adapted to the context of graphs, integrity constraints being a case in point: (Šestak et al., 2021) for example, much of the integrity constraint support available for Neo4j has been written into the Cypher query language in a similar style to SQL (e.g. `UNIQUE` or `NOT NULL`) (Ma et al., 2020).

In order to directly ingest documents into the MongoDB database, the mongoimport command can be used. This utility allows data to be imported from JSON, CSV, or TSV files. It is not necessary to specify a collection when importing data into the database; however, a collection will be created upon the addition of the records. Like other NoSQL databases, MongoDB has a dynamic schema design, allowing the documents in a collection to have varied fields and structures.

While Neo4j offers an option to effectively import large datasets in several different formats, it recommends the `LOADCSV` command as the optimal method. This facilitates the conversion from relational or other type databases to a graph database format (Karan, 2016). This command will directly map input data into a complex graph/domain structure. When performing the operation on a significant amount of data, the command can be appended with the fragment `USING PERIODIC COMMIT`, which reduces memory overhead when the transaction is being conducted.[3]

Within the MongoDB database, each document is stored in the JSON format. As JSON documents support embedded fields and nested data, related data can be included within the document instead of having to be stored externally within the collection. Embedded fields act as placeholders that can be added to text fields to dynamically display entity-specific content. Each JSON field consists of unordered key-value pairs, a form of NoSQL database that has become increasingly prevalent in recent years, with each pair stored in a key-based lookup structure (Agnelo et al., 2020). The value

[3]However, other file types such as JSON and XML are also supported and can be processed using APOC, an add-on library in Neo4j that is accessible with a very simple command.

is represented as a document encoded in standard semi-structured format.

As mentioned previously, Neo4j has native graph storage, which means that each part of the graph data model is stored separately. There are different notions as to what makes up the key components of a graph database, one of them being the property graph model, which is schemaless and allows the user to represent the data close to a real-world conception. The records in a Neo4j graph database are structured with each entity as a node, which are linked through directed connections named relationships. Nodes and relationships refer to their attributes as properties (Giabelli et al., 2021).

A fundamental challenge in Neo4j data modeling is classifying a categorical variable as a property, label, or node. A categorical variable may be described as having two or more distinct categories with no intrinsic ordering (Baak et al., 2020). The records being ingested here are rich in categorical variables; for example, in the Brocklissferte architecture, residence, diocese, province, and country all have a finite, discrete set of values. In graph data modeling, categorical variables can often result in more irregularity, due to the options there are for representing them. By way of example, let us consider the category of gender, which is characterized as a label as follows: `(:Person:male)`. As a property value, it can be represented as `(:Person {gender:'male'})`, and as a distinct node as `(:Person)-[:GENDER]->(:Gender {name: 'male'})`. These selections become even more complex when considering the cardinality for each categorical variable – for example, almost fifty dioceses are represented in the Brocklissferte dataset, along with five provinces. While by the standards of Big Data neither of these attributes can be described as having high cardinality, consistency must nonetheless be maintained in the approach to modeling them (Moeyersoms and Martens, 2015). The merits of each property type become apparent when endeavoring to retrieve variables using regular queries. A regular query may be described as a non-recursive query that traverses the graph and returns pairs of nodes connected by a common expression (Bagan et al., 2020). Labels attach simple types to nodes and relationships. They provide fast look-ups for Neo4j and are used to describe the nodes' role in a graph as well as for grouping nodes in fixed sets. However, they are a poor quality option for medium or high cardinality values, as a variable with a large amount of categories can make a data model extremely unwieldy. In addition, while relationships between connected data can be referenced directly by Neo4j, relationships between corresponding labels are hidden inside nodes and are not expressed explicitly (Zhu et al., 2019). Properties are expressed as name-value pairs and can store any data type. Thus, they can accommodate high cardinality data with ease, and can be subjected to database constraints which preserve integrity (Jiménez et al., 2016). (They cannot be set to NULL, as this is equivalent to deleting the property). A small-scale property lookup is quicker that traversing a node. The disadvantage of properties is that when multiple small categoric variables are applied to a node, performance can be adversely affected. The same can be said for parsing multiple properties to a large string or large array. When searching for all nodes that share a specific property as part of a regular query pattern, properties are not the optimal choice (Ruetter et al., 2015). The command below illustrates the syntax that was used in an attempt to return all common dioceses of a particular record (under the assumption that all fields had been modeled as properties of the 'person' node):

```
MATCH (b:Person \{ surname: 'Moore' \})
WITH b
MATCH (allCommonDiocese:Person \{ diocese: b.diocese \})
RETURN allCommonDiocese;
```

When searching for nodes that share a common property, or if the cardinality of the categorical variables is inordinately high, modeling each variable as a separate node can be the most efficient choice. Retrievability of nodes has been the subject of extensive research for a number of decades (Gacem et al., 2020). In this instance, if each record were to be filtered by shared diocese, the following query might be used:

```
MATCH (b:Person \{ surname: 'Moore' \})
WITH b
MATCH (b)-[:HAS]->(j:PersonalDetails)-
[:HAS]->(o:Origin)-
[:HAS]->(r:Residence)-
[:HAS]->(d:diocese)<-[:HAS]-(r:Residence)
    <-[:HAS]-(o:Origin)<-[:HAS]<-(j:PersonalDetails)
    <-[:HAS]-(other:Person)
RETURN count(other);
```

We can see how cumbersome this syntax can be, as the levels of the graph descend from the root. Moreover, if data is too densely connected, it can result in the creation of supernodes, i.e. vertices with a disproportionately high number of relationships, which will have a negative impact on any queries that attempt to access them.

Therefore, our approach to modeling the data in Neo4j had to be tailored to the questions the user will need to ask of the database, and an adaptable attitude is required. Unlike traditional RDBMS, there is no significant decrease in performance for highly connected data ingested, stored, and nav-

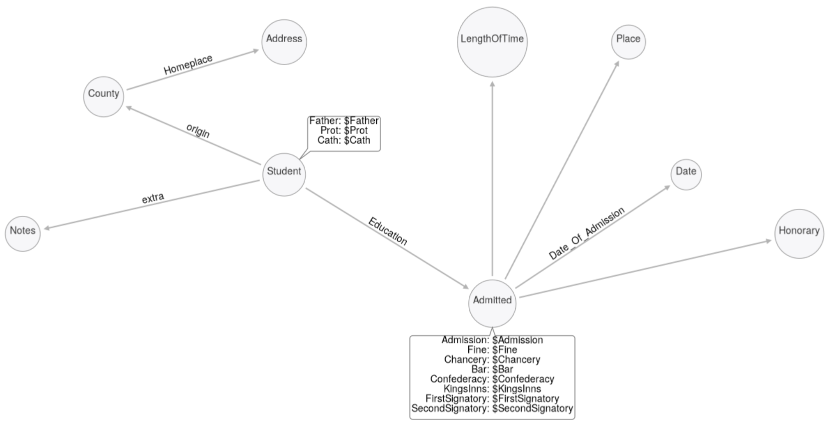

Figure 3: Property graph model of King's Inns data

igated in Neo4j – as can be noted from Figure 3, it has no difficulty in representing structured, semi-structured, or unstructured data (Perçuku et al., 2017).

## 4 The Solution

### 4.1 Analytical Work

Both types of database support a Python driver.[4] Although both types of database support several different methods of bulk import, some of which have been outlined in the previous chapter, this approach was deemed the fastest and most objective way of conducting a fair comparison test for multiple data ingestion and retrieval.

### 4.2 High Level

Retrieving data from the database in MongoDB is also done via JavaScript queries, using a simple command.

This can then be appended depending on the specificity of the results that need to be recovered. Each record in a collection will have a unique ID, which can also be accessed when the 'find' command is used with a single criterion.

The node always has its own variable name, an alias that is assigned by the user the first time the node is referenced within a query. This makes the retrieval of individual nodes a less complex process. However, returning an entire record is a more formidable task in Neo4j. Graph databases, on the other

[4]PyMongo is the recommended method of interacting with MongoDB from Python, while the official Python driver for Neo4j is 'neo4j-python-driver.'

hand, are well suited for ontology-oriented data, representing a record as a set of concepts and relationships. These entities are all dispersed and stored separately, meaning that their retrieval requires a more explicit approach.

### 4.3 Schema and Data Migration

Due to the static nature of the data, updating the records would be an infrequent operation, but it is still prudent to examine the intricacy of such an undertaking. In MongoDB, the command is straightforward:

```
db.brocklissferte.update_one(
    {'id' : origin.get('id') },
    {'$set': {"occupationalHistory":
        ({"occupation": "Cleric", "role": "Priest"})}})
```

Indexing is paramount in MongoDB, and the ID is required in order to retrieve and update the record. Updating an individual record in Neo4j is a similarly easy task:

```
MATCH (n)
    WHERE n.surname = "Clarke"
    SET n.address = "Meath"
```

The situation becomes more complicated when making changes to the structure of an individual record. Although MongoDB is commonly referred to as ‘schemaless' data, it is important to contemplate how data is stored in order to optimize database performance. If we consider the Spanish Military parent-child model, it can be observed that the framework of the record is as follows:

```
<employment>
    <employer>
        <address>
            <country>"Spain"</country>
        </address>
    </employer>
</employment>
```

In some of the records, an extra set of parameters is present, recording the details of military inspection:

```
<employment>
    <employer>
        <address>
```

```
            <country>"Spain"</country>
        </address>
        <date>1748</date>
        <record field="age">21</record>
        <record field="height">5.2</record>
        <record field="eyes">Black</record>
        <record field="hair">Black</record>
    </employer>
</employment>
```

In order to accurately represent this structure across multiple records in MongoDB, it is necessary to invoke the `$addFields(aggregation)` function, which appends extra fields into each document.

The format of a Neo4j update query is quite different. A node can be created and then accessed and added to directly using the `SET` clause.

This is an important issue when attempting to accurately represent this data in the database: the complexity of the data is such that it is necessary to append and update the data encoding habitually in order to ensure accurate reproduction. All told, both of the command syntaxes in question provide a distinct advantage over schema migration in SQL databases, as they are less complex and more intuitive.

When considering multiple collections, as is being done in this instance, one feature that needs to be discussed is merging multiple records on a particular field or attribute. In database technology, these are known as aggregate functions. The dominant aggregation framework of MongoDB uses a pipeline concept, as it provides efficient data aggregation using native operations. A pipeline is an array consisting of distinct operators which modifies a collection (or sub-collection) in stages. As the collection passes through each stage, certain operators modify the collection documents according to various techniques (Mahajan et al., 2019). A less efficient alternative is the MapReduce framework, a data processing technique that uses two stages or tasks, namely Map and Reduce. The map function takes a MongoDB document and maps each individual element to a key-value tuple. During the subsequent reduce stage, the elements are condensed and aggregated data is collected. An aggregation framework is generally faster than MapReduce, but MapReduce is useful for aggregating extremely large collections. If an archivist wishes to merge records from two of the previously described collections on a particular value – for example, the Brockliss & Fertè and King's Inns collections, both of which include a sponsor or employer address field and contain records from a similar time period – the following syntax could be used:

```
db.collection.aggregate([
    { "$lookup": {
   "from": Brocklissferte,
   "let": { "address": "$address" },
   "pipeline": [
     { "$match": { "$expr": { "$in":
       [ "$employer", "$$address" ] } } },
     { "$lookup": {
       "from": Innstudents,
       "let": { "address": "$address" },
       "pipeline": [
         { "$match": { "$expr": { "$in":
         ["$place", "$$address" ] } } } ],
       "as": "address" }} ],
    "as": "address"}}])
```

An updated parent-child model of the Brockliss & Fertè records may be observed in Figure 4. This syntax, while looking quite intricate upon initial observation, is of great help in the consolidation of nested data, as it covers the range of each record to find and merge the relevant field. In this instance, it provides an assortment of records to the user, giving access to information regarding relevant benefactors and their level of influence in a particular area. Meanwhile, `$lookup` is a pipeline operator that allows the user to perform a left join to combine two collections in the same database. A left join will contain all documents from the collection referenced first, as well as the matching documents from the second collection. All unmatched documents from the second collection will be omitted. The aggregation functions in Neo4j are comparable to those in relational databases with a syntax similar to SQL. The amount of matching rows can be queried using the `count(*)`. The `collect()` command returns a list of heterogeneous elements in a single list.

One major advantage NoSQL databases have over relational databases is the increase in scalability. A highly scalable database is one which can increase its workload and throughput when additional resources are added to it. Scalability may be supported by the following approaches: horizontal scaling, the process of adding more hardware to the system; and vertical scaling, which increases the memory of the existing server. Although our data is static, and is infrequently increased or altered, it is worth considering this aspect to fully understand the capabilities of our database. MongoDB supports horizontal scaling through sharding; a method for distributing data across several servers (Ravat et al., 2020). Neo4j is both vertically and hori-

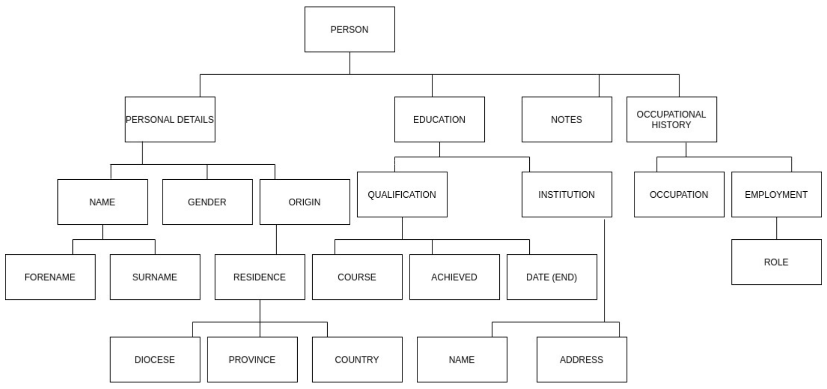

Figure 4: Modified Parent Child model of Brockliss & Ferté records

zontally scalable. The graph database platform provides high quality scaling, and Neo4j uses cypher query language, an external Domain Specific Language (DSL), which is tailored to a specific application domain. DSLs are expressive and concise and therefore aid scalability by reducing complexity (Yoon and Lee, 2018).

## 5 Evaluation

### 5.1 Solution Verification

The comparison between both types of non-relational database yielded a number of illuminating observations and results. The primary objective was to ingest the data with as little loss of data integrity as possible. As the complexity of the data was illustrated using a parent-child relationship model, it made the composing of queries and interactions with both types of database a less arduous task. The syntax of creating this structure with both MongoDB and Neo4j was exacting. However, the fact that a Cypher command could be automatically generated in Neo4j upon the creation of a graph data model, as opposed to being painstakingly produced for each individual dataset by the user, gives graph databases a distinct advantage.

### 5.2 Examination of Schema Traversal

It is worth noting the contrast between the syntax of retrieving a nested data object from both types of database, especially with regards to the complexity of each query. From traversing the parent-child model in Figure 1, it can be observed that the entity 'country' is a child of the entity 'residence,' which is a child of the entity 'origin,' which is a child of the entity 'personal details.' The parent entity is described as 'person.' To correctly traverse the data in order to retrieve this entity in MongoDB, each property has to be labeled in the command.

This results in the following record being retrieved:

```
{'name': {'forename': 'Thomas', 'surname': 'Williams'},
'_id': ObjectId('5e6396629578d4b117ea55fa'),
'origin': {'residence': 'province': 'Connaught',
'country': 'Ireland',
'diocese': 'Wexford'}},
'id': 128016,
'gender': 'male',
'education': {'qualification': {'achieved': 'yes',
'date': 1747-10-22 08:47:32',
'course': 'Law'},
'address': {'town': 'Brussels', 'country': 'Belgium'},
'institution': 'University Of Brussels'}}
```

This is a more complex syntax than the Neo4j variant. As can be observed in Figure 3, a graph data model which is representative of the structure of the record within the database, the entity 'country' is stored as a property of the node address. This node can be directly queried by the user with the following cipher query:

```
MATCH (n:address) RETURN address.country
```

The contrast in complexity between both commands emphasizes that graph-based data models provide a much more effective and adept method of data traversal when interacting with the database.

As stated earlier, Neo4j provides a graphical environment which allows the user to observe and examine patterns in the data. This can aid historians in gathering knowledge or forming an impression of a particular aspect of the dataset. To give an example: one of the smaller datasets being ingested in this study is the King's Inns dataset, which provides information regarding Irish students who attended the Inns of Court in London in the pre-Cromwellian era, 1603–1633. The records are divided into four datasets, one for each of the four preliminary colleges that students could attend to gain admittance to King's Inns in Dublin, namely Gray's Inn, Lincoln's Inn, Inner Temple, and Middle Temple. Each dataset has approximately 250 records. While the datasets may have slightly different relationships, they all possess the structure of the property graph model shown in Figure 3.

All Irish students that attended the institution during this period were obliged to be sponsored either by their family or by an affluent member of their community. Therefore, by analyzing each dataset by area as it is absorbed into Neo4j, a pattern emerges of the most wealthy areas during this time period. The Lincoln's Inn dataset linked students to 25 different counties represented by the blue nodes, with the more prominent areas having more student nodes linked to them. This can be observed in Figure 5. When the fourth dataset was added to the database a different visualization can be seen in Figure 6.

After absorbing all four datasets, 399 nodes and 359 relationships are present in the database. Clear areas of prominence have emerged, allowing the user to gain insight into areas of influence during this period:

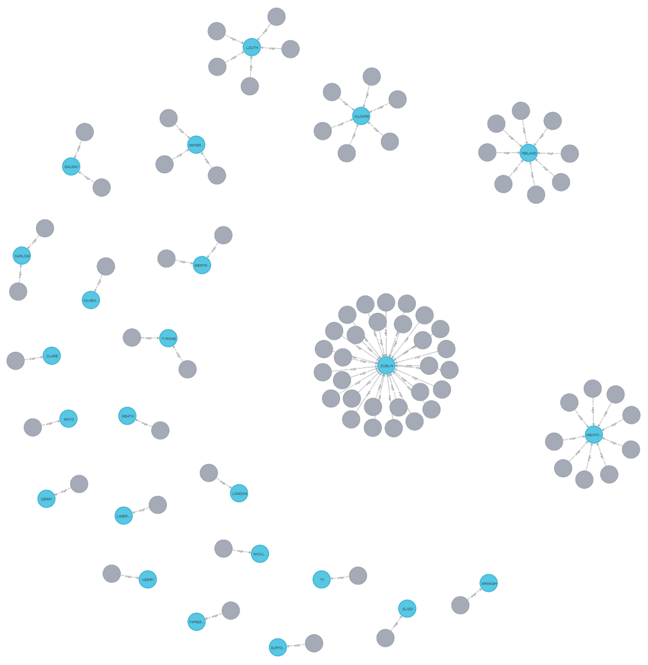

Figure 5: Visualization of Lincoln's Inn dataset by county

## 6 Conclusion

NoSQL databases have grown in popularity over the last decade, with MongoDB emerging as the forerunner for large-scale data management and processing. The advantage of adopting a collection-based approach to data administration is that each set of key-value pairs can be accessed in a flexible manner. This allows a collection to be composed of similar but diversified documents, and makes the storage and retrieval of semi-structured records more convenient for the user. Another benefit is that it is possible to interact with the MongoDB engine using the JavaScript programming language, rather than having to learn and master SQL. However, this can be regarded as a mixed blessing: SQL is a standardized programming language, designed for processing data stored in relational database management systems. Tech-

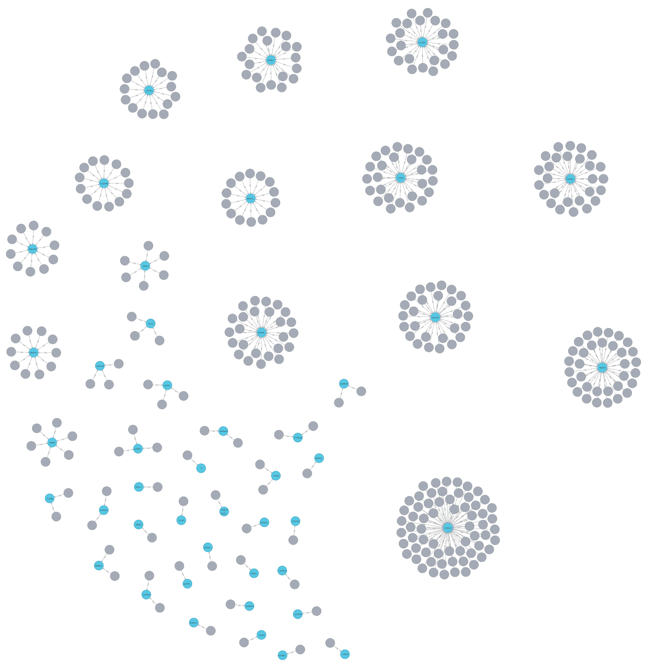

Figure 6: Visualization of four King's Inns datasets by county

nically, it is less complicated to connect different relational DBMS than it is to connect relational systems with NoSQL databases (Vathy-Fogarassy and Hugyák, 2017). In MongoDB, merge join queries and hash join queries, typically a more efficient algorithm than a nested loop join, are not possible, rendering the commands for data retrieval rather convoluted.

Throughout this paper, it can be observed that the graph database Neo4j provides an elegant alternative to other NoSQL databases for users confronted with highly complex relationships and entities. As well as providing a platform which makes the absorption of records into the database significantly easier, it also maintains uniformity between the structure of an individual record and the architecture of the data within the database. With a graph database schema, schema-instance consistency is required (Pokorný et al., 2017). This consistency makes graph databases an ideal tool for digital archivists and historians who wish to observe patterns in semi-structured and hybrid data. The Cypher querying language allows the user to easily establish links and extract linked records for observation. Although there are limitations to using a graphical database to store complex nested data, these obstacles can be surmounted by monitoring and applying the rules of data modeling to ensure the data is reproduced accurately within the database. As we have demonstrated, MongoDB outperformed Neo4j slightly in terms of speed of ingestion. However, for historians and those who work in the digital humanities, the priority will most likely be the accuracy of data transcription rather than speed, which makes Neo4j the more suitable method of data processing and management.

It cannot be concluded from this that traditional relational database management systems are ideal for the digitization of all types of historical source material. When confronted with complex and diverse real-world data, relational systems suffer from the fact that they cannot capture their inherent graph structure (De Virgilio et al., 2014). It is vital to prevent a loss of data integrity when digitally archiving records, so that each record is precisely preserved for those who wish to access it at a future point in time. It is therefore necessary to maintain an accurate representation of the data throughout the entire process of digital archiving, from modeling the structure of the data, to inserting the data into the database, to observing the structure within the database, to retrieving the data. All this makes the analysis of integrity constraints in NoSQL databases a crucial desideratum. While we hope that this paper will prove useful as a preliminary appraisal of the usefulness of nonrelational database technology for the processing of unstructured data, there is still ample potential for further research, be it in the form of a more comprehensive evaluation of NoSQL databases, a more detailed breakdown of domain and query languages used to construct integrity constraints, or extensive testing on other machines to affirm the validity of our tentative findings.

## References

Agnelo, J., Laranjeiro, N., and Bernardino, J. (2020). Using Orthogonal Defect Classification to Characterize NoSQL Database Defects. *Journal of Systems and Software*, 159:110451, DOI: `10.1016/j.jss.2019.110451`.

Angles, R. (2012). A Comparison of Current Graph Database Models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*, pages 171–177. DOI: `10.1109/ICDEW.2012.31`.

Baak, M., Koopman, R., Snoek, H., and Klous, S. (2020). A New Correlation Coefficient Between Categorical, Ordinal and Interval Variables With Pearson Characteristics. *Computational Statistics & Data Analysis*, 152:107043, DOI: `10.1016/j.csda.2020.107043`.

Bagan, G., Bonifati, A., and Groz, B. (2020). A Trichotomy for Regular Simple Path Queries on Graphs. *Journal of Computer and System Sciences*, 108:29–48, DOI: `10.1016/j.jcss.2019.08.006`.

Bjeladinovic, S., Marjanovic, Z., and Babarogic, S. (2020). A Proposal of Architecture for Integration and Uniform Use of Hybrid SQL/NoSQL Database Components. *Journal of Systems and Software*, 168:110633, DOI: `10.1016/j.jss.2020.110633`.

Bono, A. (2007). Historical Seismometry Database Project: A Comprehensive Relational Database for Historical Seismic Records. *Computers & Geosciences*, 33(1):94–103, DOI: `10.1016/j.cageo.2006.05.007`.

Borissova, V. (2018). Cultural Heritage Digitization and Related Intellectual Property Issues. *Journal of Cultural Heritage*, 34:145–150, DOI: `10.1016/j.culher.2018.04.023`.

Castelltort, A. and Martin, T. (2018). Handling scalable approximate queries over NoSQL graph databases: Cypherf and the Fuzzy4S framework. DOI: `10.1016/j.fss.2017.08.002`.

De Virgilio, R., Maccioni, A., and Torlone, R. (2014). Model-Driven Design of Graph Databases. In Yu, E., Dobbie, G., Jarke, M., and Purao, S., editors, *Conceptual Modeling. ER 2014*, volume 8824 of *Lecture Notes in Computer Science*, pages 172–185. DOI: `10.1007/978-3-319-12206-9`$_1$4.

Edward, S. G. and Sabharwal, N. (2014). *Practical MongoDB: Architecting, Developing, and Administering MongoDB*. Apress.

Gacem, A., Papadopoulos, A. N., and Boukhalfa, K. (2020). Scalable Distributed Reachability Query Processing in Multi-Labeled Networks. *Data & Knowledge Engineering*, 130:101854, DOI: `10.1016/j.datak.2020.101854`.

Giabelli, A., Malandri, L., Mercorio, F., Mezzanzanica, M., et al. (2021). Skills2Job: A Recommender System That Encodes Job Offer Embeddings on Graph Databases. *Applied Soft Computing*, 101:107049, DOI: `10.1016/j.asoc.2020.107049`.

González-Aparicio, M. T., Younas, M., Tuya, J., and Casado, R. (2018). Testing of Transactional Services in NoSQL Key-Value Databases. *Future Generation Computer Systems*, 80:384–399, DOI: `10.1016/j.future.2017.07.004`.

Henderson, R. (2020). Using Graph Databases to Detect Financial Fraud. *Computer Fraud & Security*, 2020(7):6–10, DOI: `10.1016/S1361-3723(20)30073-7`.

Holzschuher, F. and Peinl, R. (2016). Querying a Graph Database – Language Selection and Performance Considerations. *Journal of Computer and System Sciences*, 82(1):45–68, DOI: 10.1016/j.jcss.2015.06.006.

Jiménez, P., Diez, J. V., and Ordieres-Mere, J. (2016). HOSHIN KANRI Visualization with Neo4j. Empowering Leaders to Operationalize Lean Structural Networks. *Procedia CIRP*, 55:284–289, DOI: 10.1016/j.procir.2016.08.023.

Jose, B. and Abraham, S. (2020). Performance Analysis of NoSQL and Relational Databases With MongoDB and MySQL. *Materials Today: Proceedings*, 24, Part 3:2036–2043, DOI: 10.1016/j.matpr.2020.03.634.

Karan, K. K. (2016). *Visualizing and Searching Relationships Between Academic Papers Using Neo4j Graph Database*. PhD thesis, Thapar Institute.

Kingdon, A., Nayembil, M. L., Richardson, A. E., and Smith, A. (2016). A Geodata Warehouse: Using Denormalisation Techniques as a Tool Fordelivering Spatially Enabled Integrated Geological Information Togeologists. *Computers & Geosciences*, 96:87–97, DOI: 10.1016/j.cageo.2016.07.016.

Le May, S., Carter, B., Gehly, S., Flegel, S., et al. (2020). Representing and Querying Space Object Registration Data Using Graph Databases. *Acta Astronautica*, 173:392–403, DOI: 10.1016/j.actaastro.2020.04.056.

Lotfy, A. E., Saleh, A. I., El-Ghareeb, H. A., et al. (2016). A Middle Layer Solution to Support Acid Properties for NoSQL Databases. *Journal of King Saud University - Computer and Information Sciences*, 28(1):133–145, DOI: 10.1016/j.jksuci.2015.05.003.

Ma, T., Pan, Q., Wang, H., Shao, W., et al. (2020). Graph Classification Algorithm Based on Graph Structure Embedding. *Expert Systems with Applications*, 161:113715, DOI: 10.1016/j.eswa.2020.113715.

Mahajan, D., Blakeney, C., and Zong, Z. (2019). Improving the Energy Efficiency of Relational and NoSQL Databases via Query Optimizations. *Sustainable Computing: Informatics and Systems*, 22:120–133, DOI: 10.1016/j.suscom.2019.01.017.

Moeyersoms, J. and Martens, D. (2015). Including High-Cardinality Attributes in Predictive Models: A Case Study in Churn Prediction in the Energy Sector. *Decision Support Systems*, 72:72–81, DOI: 10.1016/j.dss.2015.02.007.

Mohmmed, A. G. M. and Osman, S. E. F. (2017). Study on SQL vs. NoSQL vs. NewSQL. *Journal of Multidisciplinary Engineering Science Studies*, 3(6):1821–1823.

Mosquera, N. and Piedra, J. (2017). Use of Graph Database for the Integration of Heterogeneous Data About Ecuadorian Historical Personages. In *2018 7th International Conference On Software Process Improvement (CIMPS)*, pages 95–100. DOI: `10.1109/CIMPS.2018.8625618`.

Perçuku, A., Minkovska, D., and Stoyanova, L. (2017). Modeling and Processing Big Data of Power Transmission Grid Substation Using Neo4j. *Procedia Computer Science*, 113:9–16, DOI: `10.1016/j.procs.2017.08.276`.

Pokorný, J., Valenta, M., and Kovačič, J. (2017). Integrity Constraints in Graph Databases. *Procedia Computer Science*, 109:975–981, DOI: `10.1016/j.procs.2017.05.456`.

Ravat, F., Song, J., Teste, O., and Trojahn, C. (2020). Efficient Querying of Multidimensional Rdf Data With Aggregates: Comparing NoSQL, RDF and Relational Data Stores. *International Journal of Information Management*, 54:102089, DOI: `10.1016/j.ijinfomgt.2020.102089`.

Ruetter, J., Romero, M., and Vardi, M. Y. (2015). Regular Queries on Graph Databases. In Arenas, M. and Ugarte, M., editors, *18th International Conference on Database Theory (ICDT'15)*, number 31 in Leibniz International Proceedings in Informatics (LIPIcs), pages 177–194. DOI: `10.4230/LIPIcs.ICDT.2015.177`.

Sabharwal, A. (2015). *Digital Curation in the Digital Humanities : Preserving and Promoting Archival and Special Collections*. Chandos Publishing, Oxford.

Schulz, W. L., Nelson, B. G., Felker, D. K., Durant M.D., T. J., et al. (2016). Evaluation of Relational and NoSQL Database Architectures to Manage Genomic Annotations. *Journal of Biomedical Informatics*, 64:288–295, DOI: `10.1016/j.jbi.2016.10.015`.

Simonini, G., Gagliardelli, L., Bergamaschi, S., and Jagadish, H. (2019). Scaling Entity Resolution: A Loosely Schema-Aware Approach. *Information Systems*, 83:145–165, DOI: `10.1016/j.is.2019.03.006`.

Taktek, E. and Thakker, D. (2020). Pentagonal Scheme for Dynamic XML Prefix Labelling. *Knowledge-Based Systems*, 209:106446, DOI: `10.1016/j.knosys.2020.106446`.

Vágner, A. (2018). Store and Visualize EER in Neo4j. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, ISCSIC '18, pages 1–6, New York, NY. Association for Computing Machinery, DOI: 10.1145/3284557.3284694.

Vathy-Fogarassy, A. and Hugyák, T. (2017). Uniform Data Access Platform for SQL and NoSQL Database Systems. *Information Systems*, 69:93–105, DOI: 10.1016/j.is.2017.04.002.

Čerešňák, R. and Kvet, M. (2019). Comparison of Query Performance in Relational a Non-relation Databases. *Transportation Research Procedia*, 40:170–177, DOI: 10.1016/j.trpro.2019.07.027.

Šestak, M., Heričko, M., Družovec, T. W., and Turkanović, M. (2021). Applying K-Vertex Cardinality Constraints on a Neo4j Graph Database. *Future Generation Computer Systems*, 115:459–474, DOI: 10.1016/j.future.2020.09.036.

Šestak, M., Rabuzin, K., and Novak, M. (2016). Integrity Constraints in Graph Databases – Implementation Challenges. In Hunjak, Tihomir; Kirinić, V. K. M., editor, *Proceedings of Central European Conference on Information and Intelligent Systems*, pages 23–30. https://urn.nsk.hr/urn:nbn:hr:211:102684.

Yoon, J. and Lee, S. (2018). A Method and Tool to Recover Data Deleted From a MongoDB. *Digital Investigation*, 24:106–120, DOI: 10.1016/j.diin.2017.11.001.

Zhu, Z., Zhou, X., and Shao, K. (2019). A Novel Approach Based on Neo4j for Multi-Constrained Flexible Job Shop Scheduling Problem. *Computers & Industrial Engineering*, 130:671–686, DOI: 10.1016/j.cie.2019.03.022.