Querying Financial Data using Neo4j and MongoDB

Federico Giacardi

University of Turin

Abstract. This brief paper summarises the development of an application to query financial data generated using the LDBC Financial Benchmark project. This work was developed as final assignment for the course of Advanced Databases Models and Architectures, held at the University of Turin in 2025, by Professor Pensa.

Keywords: Graph Database · Document based datastores · Financial data

1 Introduction

The main aim of the project was to put in practice what learnt during theory lessons on NoSql databases. Among the wide range of alternatives, Neo4J and MongoDB where chosen as the architectures to test. Taking into consideration the main strengths of these two systems, namely efficient processing of complex relations between domain concepts, and the possibility to store and process, in a distributed manner, a huge number of data with complex and variable structure, the LDBC FinBench data generator was chosen as source of data for the application. The rest of the work is organized as follows:

1. Section two describes the data modelling and the database creation processes
2. Section three describes the queries designed, as well as the web application to test them
3. Section Four discusses queries performances in different scenarios

2 Data Modelling and Database creation

The main idea behind the data modelling process was to replicate the data schema presented in [1] in the graph-based data model of Neo4j, and the document-based one of MongoDB. The LDBC FinBench generator produces, for each class and relation of the schema, a set of TSV files. A preprocessing step was therefore necessary before importing data, where TSV files related to the same class/relation were merged, and the result converted to CSV (with proper escaping for commas in textual fields). This step is implemented in the csv_converter.py module. The development process made use of data generated using scale factor

0.1, which produces around 100 MB of data (for a complete breakdown of documents distribution between classes, refer to [1]). Both local and cloud (using Neo4J Aura and MongoDB Atlas, respectively) instances of this database were created, the first for testing and development of the web application, and the second to test performance.

2.1 Neo4J Model

The Neo4J implementation is straightforward, and it suffices to remember that the final model will be a graph, where each node corresponds to a UML class, and each edge to a relation between classes. Furthermore, the attributes of classes and their relationship constitute the set of properties associated to corresponding nodes and edges. For the cloud version, the visual editor offered by Aura [2] was used for both schema definition and CSV import. A cypher script to create the schema is contained in the project in the setup/neo4j folder, should it be needed for a deeper analysis. The sole problem of the whole process was matching Cypher data types with those required in the original schema. Numerical values were modelled using Integer and Float (both 64-bit wide [3]) which avoids precision problems even for bigger versions of the database, where ID values could grow quite large, while dates were also modelled as Integers, since the data generator outputs them in milliseconds elapsed from Unix Time. For the local version, the Neo4J data importer [4] was used to replicate the same process (and data schema).

2.2 MongoDB Model

For MongoDB, the first step of the modelling process involved the identification of main concepts, to be represented as separate collections, and side ones, to be embedded inside the main collection they depend from. To make this choice, one must consider that embedding

1. makes operations on the individual embedded document, such as update, more complex and expensive, and may introduce duplicates that must be kept aligned. It may also slow down a complete scan of side documents
2. speeds up the traversal of the relation between main and side document, avoiding an expensive join operation, and reducing execution time for queries that exploit such relation

Consequently, this decision should really be evaluated on a case-by-case basis, but overall, embedding small documents that represent relations that we will navigate in our queries and storing complex documents that correspond to important domain concepts in separate collections should be a sensible strategy. Applying these considerations leads to the following main collections, featuring all the fields indicated in [1], plus those specified in the following bullet list:

1. Person, with additional fields
   (a) own, which embeds the person-own-account relation

   (b) invest, which embeds the person-invest-company relation
   (c) guarantee, which embeds the person-guarantee-person relation
   (d) apply, which embeds the person-apply-loan relation
2. Company, with additional fields
   (a) own, which embeds the company-own-account relation
   (b) apply, which embeds the company-apply-loan relation
   (c) guarantee, which embeds the company-guarantee-company relation
   (d) invest, which embeds the company-invest-company
3. Account, with additional fields
   (a) transfer, which embeds the account-transfer-account relation
   (b) withdraw, which embeds the account-withdraw-account relation
   (c) repay, which embeds the account-repay-loan relation
4. Loan, with additional fields
   (a) deposit, which embeds the loan-deposit-account relation
5. Medium, with additional fields
   (a) signIn, which embeds the medium-signIn-account relation

Note that Mongo also allows for the definition of collections' schemas, which could be used to enforce the structure described in [1]. However, this project doesn't make use of this feature, since it would contradict the spirit of document-based datastores, which rely on a flexible structure that may also differ between documents of the same collection. Before importing data, a merging module, named merger_mongo.py, has been developed to convert CSV files into JSON and merge them according to the embedded structure described above. Again, both cloud and local versions of the database were created by importing the JSON files with the import_mongo.py. The structure of these two scripts is straightforward, and their few complex points are clarified with comments inside the code. The sole points worth mentioning are the usage of a JSON file to specify the embeddings as described above, and the groupBy Python function, to collect documents according to the value of a certain attribute. Replicating the process described in this section is not necessary to run the web application, but can be done as follows

1. Run the LDBC Finbench data generator
2. copy the output/raw folder to setup/dumps
3. In create_db.py change the value of the DUMP variable to the name of the folder from step two
4. Change the MONGO_URI value to the address of local Mongod server
5. Run create_db.py
6. Connect to the Neo4j Data Importer [4] to load CSV files and associate them to graph nodes and edges (which can be created using the GUI)

3 Queries and Application Development

3.1 Analytic Queries

The first analytic query computes the Jaccard similarity between the set of companies that two users, whose ids are passed as parameters, invest in. This implements an analytic, parametric function, as requested in the project's specification, while exploiting the ability of graph databases to quickly navigate relations between nodes. Having said C1 the set of companies the first user invests in, and C2 the set of companies the second user invests in, Jaccard's similarity can be defined as $\frac{|C1 \cap C2|}{|C1 \cup C2|}$. Actually, this computation is implemented using three separate queries. The first one, defined as follows

```
MATCH (p:Person)-[:PersonInvest]->(:Company)
RETURN DISTINCT p.id as id, p.name as name
ORDER BY name
```

returns a list of persons that invest in at least one company. This stage of the pipeline is mainly for visualisation purposes, so that the application user is given a list to choose the persons of interest from, without having to remember their ids. The second query, defined as

```
MATCH (p1:Person)-[:PersonInvest]->(c1:Company)<-[:PersonInvest]-(p2:Person)
WHERE p1.id = $usrOne AND p2.id = $usrTwo
RETURN DISTINCT c1.id
```

computes the numerator of the similarity definition. This is the first example where the possibility of formulating clean and efficient queries on the relations between domain concepts, thanks to the graph data model and the ASCII-Art-based language, can be appreciated. $usrOne and $usrTwo are query parameters that are assigned the ids of the persons selected from the result of the previous query by the user. Note that the DISTINCT clause is needed because p1 and p2 will be assigned to both users, which means that the same company is returned twice, one for the (usrOne, usrTwo) pair, and the other for (usrTwo, usrOne). Lastly, the final query, defined as

```
MATCH (p1:Person)-[:PersonInvest]->(c1:Company)
WHERE p1.id = $usrOne OR p1.id = $usrTwo
RETURN DISTINCT c1.id as id
```

computes the denominator of the similarity definition. The query is self-explanatory, with the DISTINCT clause being necessary to avoid counting companies that belong to the intersection twice. The computation of the Jaccard similarity is finalised by the method of the web application's backend that executes the three queries and later computes the cardinalities of the two sets and their fraction.

The MongoDB query computes, for each of the companies returned by the union query, a summary sheet, which is a recap of its financial situation. The summary includes the number of investors (companies that invest in the current one), the number of accounts owned (to be interpreted as an indicator of available financial resources), and the number of loans contracted (to be interpreted as an indicator of debt level). The query is implemented as follows

```
db.company.aggregate(
    [
        { $match: { id: { $in: ids } } },
        {
            $addFields: {
                numInvest: { $cond: [{ $isArray: "$invest" }, { $size: "$invest" }, 0] },
                numAccounts: { $cond: [{ $isArray: "$own" }, { $size: "$own" }, 0] },
                numLoans: { $cond: [{ $isArray: "$apply" }, { $size: "$apply" }, 0] }
            }
        },
        {
            $group: {
                _id: "$id",
                investors: { $first: "$numInvest"},
                accounts: { $first: "$numAccounts"},
                loans: { $first: "$numLoans"},
                name: {$first: "$name"}
            }
        }
    ]
)
```

where ids is the array returned by the Neo4j query. The aggregate function has been introduced to execute multi-stage queries as a more efficient and versatile variant of the mapReduce command. In the first stage, a $match operator is executed to select only documents referring to companies returned by the Neo4J query. In the second stage, three different fields, containing the data that make up the summary, are created and evaluated to either the number of secondary documents representing the relations of interest embedded in the main collection, or to zero, if such documents do not exist. In the third and last stage, a $group operator maps each company to its summary. Note that this is not a map-reduce query in the strict sense of the term, since there is no reduce stage, but it can still be implemented efficiently using that idea. It is worth noting how this query, that ultimately counts the number of matches between related instances of different domain concepts (e.g. a company and its investors), can be evaluated without computing expensive join operators or scanning additional collections, as it would be in the relational model. This is an example of how document aggregation, one of the main features of document-based datastores can speed up certain kind of queries. The query provides an interesting example of polyglot persistence, where different databases, based on different data models, are coupled within the same application, so that many kinds of queries can be handled efficiently. In this case, recovering the companies of interest would be quite inefficient in Mongo, given the way data was modelled, as it would require the equivalent of a join operation between Persons and Companies. On the other hand, having different copies of the data induces additional creation and alignment costs.

3.2 Lookup Queries

The only lookup query tested is designed to identify possible money laundering. The idea described in the following is actually a variant of what was proposed in [1]. A possible money laundering transaction is defined as a transfer, that starts and completes in a certain time frame selected by the user, of money between a source account, src, and a destination account, dst, via an intermediate account, mid, where at least one of the three is suspect (which is modelled by setting the isBlocked property to true). The detailed definition of the query is the following:

```
MATCH (pSrc:Person)-[:Own]->(src: Account)-[rcv:Transfer]
->(mid:Account)-[snd:Transfer]->(dst:Account)<-[:Own]-(pDst:Person)
WHERE (src.isBlocked or dst.isBlocked or mid.isBlocked)
AND (apoc.date.convert(rcv.createTime, 'ms', 'd') > $startWindow)
AND (apoc.date.convert(rcv.createTime, 'ms', 'd') < $endWindow)
RETURN pSrc.name as srcNick, mid.nickname as midNick,
pDst.name as dstNick,
apoc.date.convert(rcv.createTime, 'ms', 'd') as startDate
```

It is clear that this query aims at testing a point already made several times, namely the ability of graph-based datastores to efficiently process complex relations between domain concepts. In this case, the relation exploited is actually multi-layered (from Person, to Account, to Transfer and then back). Expressing it in a relational environment would require the usage of many join operators which are well known for their computational cost, as well as being quite difficult to formalise. The apoc.date.convert() function is used to convert the createTime property of the transfer from milliseconds to days, so that choosing the desired time window is easier for the user.

3.3 Web Page

As requested by the project's specifications, a web application was built to test queries described above. It interacts with the databases populated with data generated using a scale factor of 0.1, which yields around 100 MB of data, as described above, stored in cloud, so that no local copy is needed. The application was developed using React and Typescript, mainly because these two technologies allow for a modular, clean organisation of the code. Typescript was chosen because its type system reduces errors and speeds up development, while retaining access to the vast libraries ecosystem of Javascript. The NodeJS Express framework was used to develop the application's backend in Javascript. The code of the application is divided into two folders: frontend, which contains the React modules that implement the pages of the application and its components, and backend, which instead contains, under the router folder, the implementation of the queries discussed above, further split in the analytic and lookup modules. All the configuration parameters needed to connect to the two databases are stored in the .env file. The backend code is straightforward: the sole point that deserves special attention is the disableLosslessIntegers option when connecting to Neo4J. As described above, Neo4J uses 64-bit wide integers, while JavaScript can use at most 53 bit to represent them. One is then left with two options: either accept the default configuration, which doesn't return a single value, but rather an interval estimate of the form (high, low), which is quite difficult to handle, or disable lossless integers. In this case, given that queries are only run against databases of limited size, it is safe to adopt the second option, which might however become inadequate if tests with bigger databases were to be implemented. The frontend of the application is organized in the following modules:

– Home, which briefly describes the project and the rest of the application
– Lookup, that executes queries described in the lookup section and display their results
– Analytic, that executes the analytic query and displays the graph result using [5]

For a description of the application's setup procedure, please refer to the README file of the GitHub repository of the project. The content of the .env file must be the following:

```
NEO_URI="neo4j+s://66745bf0.databases.neo4j.io"
NEO_USERNAME="neo4j"
NEO_PASSWORD="TLpvzTbUbZZ8hXCVnqVlv9d-OT7nRC1v6m_b0AdNn2o"
MONGO_CONN_STRING=
"mongodb+srv://federicogiacardi:6xBk16ATJUobqLEE@finbenchproject.jgcqnka.mongodb.net/?
retryWrites=true&w=majority&appName=FinBenchProject"
MONGO_DB_NAME="LDB_FIN"
```

4 Performance Evaluation

The following table summarizes the performance of the queries All times are

Query	SF 0.1	SF 0.3	SF 1	SF 3
investors	62	133	152	215
intersection	12	23	48	56
union	26	40	57	71
money laundering	215	317	543	1993
indexed money laundering	60	196	369	1316
summary sheets	19	55	293	409

expressed in milliseconds elapsed from start to end of query execution. All tests were carried out using local copies of the database, so that no network latency is included in the results. The machine used for testing has an Intel i7-9750H processor, with 12 cores. Query execution was carried out using range indexes (note that in Neo4J range indexes also support strict equality predicates) on Person.id and Company.id, for both copies of the database. Given the high execution time of the money laundering detection query, a range index on Transfer.createTime was created: as expected, it contributed to a significant reduction in execution times, as visible in the penultimate row of the table. The results obtained align with the initial expectations. The investors list query is slower then the other involved in the Jaccard similarity computation, which is normal, given that it has to retrieve all Person nodes, as well as the associated Companies. On the other hand, the other two queries only retrieve companies for two Person nodes. The money laundering detection query is the slowest, which is totally understandable, given the complex, multi-level relations it has to traverse. Note as well that the total execution time is significantly lower than what could be obtained using a relational database and a join query. As expected, the performance of the queries downgrades significantly as the amount of data to process increases, as a proof of the huge computational (and storage) capacity needed to process big data. Bigger scale factors were also available, but they could not be tested due to the limited storage capacity of the machine used for testing. However, it can be speculated that results would follow the trend highlighted above.

References

1. LDBC FinBench project specification, `https://ldbcouncil.org/ldbc_finbench_docs/ldbc-finbench-specification.pdf`
2. Neo4J Aura, `https://neo4j.com/product/auradb/`
3. GQL Conformance, `https://neo4j.com/docs/cypher-manual/current/appendix/gql-conformance/`
4. Neo4J Data Importer, `https://neo4j.com/docs/data-importer/current/`
5. Reagraph network graph visualization library for React, `https://github.com/reaviz/reagraph?tab=readme-ov-file`