# CAS Report
# Neo4j in comparison with PostgreSQL

## CAS Data Science and Machine Learning

20.06.2022

by Roman Schmocker, Ergon Informatik AG

Supervisors:
Prof. Dr. Sven Helmer

Department of Informatics
University of Zurich

University of Zurich[UZH]

# Abstract

Graph databases such as Neo4j provide a viable alternative for traditional relational databases if the underlying data model can be translated into a graph structure. We have analyzed five different queries on the same data set in Neo4j and PostgreSQL and compared response time and query complexity. The result shows that PostgreSQL had a faster response time for all but one query, whereas queries written for Neo4j were generally more readable than their SQL equivalent.

# Contents

# 1 Introduction

Relational databases have been around for many years now and form a cornerstone of most modern IT applications. In recent years however, NoSQL databases have gained more widespread use. NoSQL databases renounce the table model and the underlying relational algebra of traditional database systems and instead provide alternative data storage solutions, which are specialized to a specific task.

Graph databases are one such type of NoSQL databases. They internally model all data as a graph, and as such are optimized towards graph traversal, path finding and visualizing relationships within graph structures.

A graph database might therefore be an ideal addition for the software project that we are working on at Ergon Informatik AG. We want to build a dossier management system for public prosecutors and the police. A graph database could be used for example to visualize relations between different suspects or to track the routes where money has been paid to in a money laundering scheme.

In this report we will try to compare and find advantages of a graph database over relational databases. We will pick PostgreSQL as a relational database, because it is a popular solution that is already in use within our project. For the graph database, we will use Neo4j - mostly because there are data sets readily available for our analysis. As we already use PostgreSQL in our project, the goal of this analysis is not to decide whether to use one database system or the other, but rather to find out whether it is worthwile to add Neo4j in addition to PostgreSQL. Therefore, all of the queries we analyze specifically target a scenario where the underlying data model is already a graph, as we would otherwise simply use the existing PostgreSQL database.

# 2 Database Systems

## 2.1 PostgreSQL

PostgreSQL [posa] is an open source project that has been in development since 1996. Its theoretical foundation is relational algebra [Cod70] and therefore it represents a typical relational database system. Data is stored in tables which can reference each other using foreign key relations. The database imposes a strict schema on tables and columns, meaning all data has to conform to certain rules before it can be stored. This is in contrast to most NoSQL solutions where there is a greater degree of freedom with regards to the type of data that can be stored. PostgreSQL uses SQL as a query language to filter and retrieve data from the database.

A graph can be stored within a relational database using a `node` table for all the nodes and an `edge` table for all edges of a graph. The `edge` table has two foreign key columns `start_node` and `end_node` which both relate to the `node` table.

## 2.2 Neo4j

Neo4j [neo] is an open source graph database system with commercial support. It has been in development since 2007. As a graph database, Neo4j stores all its data as a labelled property graph - a graph composed of nodes and edges which both can have any number of named properties. Additionally, both nodes and edges can have a type (or label) which can be used to classify nodes and impose some rules on a node regarding its properties.

For efficient graph traversal, Neo4j uses index-free adjacency internally [VWA+14]. Index-free adjacency is a graph storage technique which allows to follow along graph edges simply by looking up a pointer in memory. This is different from a relational database where a foreign key relation needs to be resolved by looking up the physical address of another entry via a database index and then loading the corresponding table entry to memory in a second step.

The data in Neo4j is queried through Cypher query language, an SQL-like language specifically desigend for graphs. One of the main features of Cypher is the $(a) \rightarrow (b)$ operator, which is used to find two nodes a and b connected by an edge.

# 3 Methods

## 3.1 The ICIJ offshoreleaks data set

The data set used for the comparison are the offshoreleaks papers [off] from the International Consortium of Investigative Jorunalists (ICIJ). The ICIJ provides the database as a Neo4j dump file, and thus it can be directly imported into Neo4j. The data set is a huge graph which shows connections between letterbox companies (Entities) with their addresses, their board of owners and directors (Officers) as well as any intermediaries that provide legal services. In total the graph contains about 1.97 million nodes and 3.27 million edges.

To load the same data set into a PostgresSQL database we used the "Export as CSV" function in Neo4J and then imported the resulting file into a temporary PostgreSQL table using the `COPY` statement. As nodes and edges are mixed into a single table using this technique, we further copy the data into a `node` and an `edge` table with proper foreign key relations in between. The type and labelled properties of the graph are mapped to individual columns in the database table. We further added indices in PostgreSQL on the name and all foreign key columns.

Although it is possible to partition the `node` and `edge` tables based on their labels, we decided against it to keep the flexibility of traversing the whole graph in SQL.

## 3.2 Queries

For our comparison we select four queries as in [PZLO17]. In their paper they analyzed a simple point query to fetch just a single result, a one-hop and two-hop query to traverse along one or two edges respectively, and a shortest path query. Additionally, we also add a slightly more complex query which is aimed towards finding a subgraph pattern within the whole graph. Note that - with the exception of the point query - all of our queries require some kind of graph traversal. This means the benchmark works in favor of Neo4j, which is designed to understand and operate on graphs data structures whereas PostgreSQL only understands tables and relations. But as already mentioned, the idea is not to pit one system against the other but rather to identify scenarios where a graph database is so superior to a relational database that the integration of a second database management system might be justified.

### 3.2.1 Point Query

The point query is just a lookup of a graph node by its name property. Edge traversal should not be necessary and therefore also no join in a relational database.


Figure 3.1: Example result of a point query

### 3.2.2 One-Hop Query

The one-hop query will search for a specific node by its name and look for any neighbors. It should return the found node as well as all directly adjacent nodes.


Figure 3.2: Example result of a one-hop query

### 3.2.3 Two-Hop Query

The two-hop query is similar to the one-hop query, but will also search for neighbors of neighbors of the start node and return them as well. It is important that these neighbors exist, otherwise the result would be empty. This can be seen by the address node "Av. Santa Maria" in Figure 3.2, which is present in the one-hop query but missing in the two-hop query as it has no neighbors on its own.

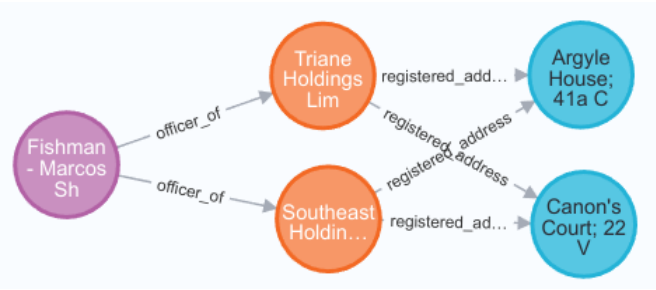


Figure 3.3: Example result of a two-hop query

### 3.2.4 Shortest Path

The goal is to find the shortest path between two nodes, if they are connected at all. In our example we want to investigate if Marc Rich, the founder of Glencore who was later convicted for tax evasion and several other crimes, still has ties to the Glencore group. The query should try to find a path regardless of the directionality of edges, which means travelling back along a directed edge is also valid. This allows us to identify a connection between letterbox companies which are registered to the same address.


Figure 3.4: Example result of a shortest path query

### 3.2.5 Subgraph Matching

We want to find two different officers for the same company who live at the same address. This query can for example be used to find family relations within the offshoreleaks data set.

The type of query might be interesting because it requires the database to perform a pattern matching algorithm of a small subgraph structure against the whole graph. We would like to see if Neo4J has some tricks up its sleeve to faciliate the evaluation of these queries.

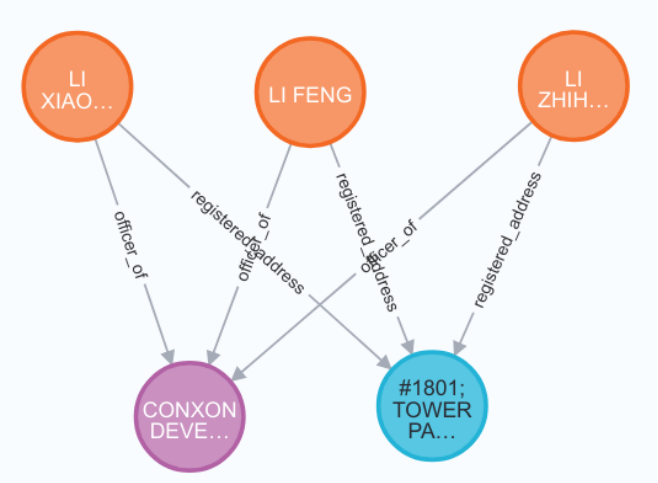


Figure 3.5: Example result of a subgraph matching query

# 4 Results

## 4.1 Performance

We repeatedly executed the above queries and measured average response time and standard deviation. The first three queries are each repeated 100 times with varying strings as search input. The shortest path and subgraph matching queries were repeated only 10 times with a fixed input. The measurements were done with Neo4J 4.4 and PostgreSQL 14.3, each database running in its own Docker container. The host system has an AMD Ryzen 7 3700X CPU with 8 Cores and 32GB RAM. The result of the measurements is shown in table 4.1.

| | no. of queries | Neo4j | PostgreSQL |
|---|---|---|---|
| Point Query | 100 | $0.008 \pm 0.016$ | $0.002 \pm 0.001$ |
| One-Hop | 100 | $0.006 \pm 0.012$ | $0.002 \pm 0.003$ |
| Two-Hop | 100 | $0.009 \pm 0.014$ | $0.004 \pm 0.003$ |
| Shortest Path | 10 | $2.638 \pm 0.027$ | $14.161 \pm 0.113$ |
| Subgraph Matching | 10 | $5.360 \pm 0.175$ | $1.557 \pm 0.225$ |

Table 4.1: Response time in seconds (mean $\pm$ standard deviation).

## 4.2 Query complexity

The following listings provide a side-by-side comparison of Cypher and SQL queries.

### 4.2.1 Point Query

```
-- Cypher
match (a:Officer)
  where a.name = 'Fishman - Marcos Shulim'
return a;

-- SQL
select *
from node a
where a.name = 'Fishman - Marcos Shulim'
  and a.labels = ':Officer';
```

### 4.2.2 One-Hop Query

```
-- Cypher
match (a:Officer)-->(b)
  where a.name = 'Fishman - Marcos Shulim'
return a, b;

-- SQL
select *
from node a
         join edge ab on a.id = ab.node_start
         join node b on ab.node_end = b.id
```

```
where a.name = 'Fishman - Marcos Shulim'
  and a.labels = ':Officer';
```

### 4.2.3 Two-Hop Query

```
-- Cypher
match (a:Officer)-->(b) -->(c)
  where a.name = 'Fishman - Marcos Shulim'
return a, b, c;

-- SQL
select *
from node a
         join edge ab on a.id = ab.node_start
         join node b on ab.node_end = b.id
         join edge bc on b.id = bc.node_start
         join node c on bc.node_end = c.id
where a.name = 'Fishman - Marcos Shulim'
  and a.labels = ':Officer';
```

### 4.2.4 Shortest Path

```
-- Cypher
match
  (a {name: 'Glencore Group'}),
  (b {name: 'Marc Rich Real Estate GmbH'}),
  p = shortestPath((a)-[*..10]-(b))
return p;

-- SQL
with recursive
    edge_both(node_start, node_end) as (
        select distinct *
        from (select node_start, node_end
              from edge
              union
              select node_end, node_start
              from edge) combined),

    path(p_start, p_end, distance, nodes) as (
        select e.node_start, node_end, 1, array [node_start, node_end]
        from edge_both e
        where e.node_start = (select id from node where name = 'Glencore Group')
        union all
        select distinct on (path.p_start,e.node_end) path.p_start,
                                                     e.node_end,
                                                     path.distance + 1,
                                                     path.nodes || e.node_end
        from path
                 join edge_both e on path.p_end = e.node_start
        where e.node_end != any (path.nodes)
          and path.distance < 10
    )
select *
from node source
         join path p on (source.id = p.p_start)
         join node target on (p.p_end = target.id)
where target.name = 'Marc Rich Real Estate GmbH'
order by distance limit 1;
```

### 4.2.5 Subgraph Matching

```
-- Cypher
MATCH
    (a:Officer) -[:officer_of]-> (entity:Entity) <-[:officer_of]- (b:Officer),
    (a) --> (address:Address) <-- (b)
RETURN a, b, entity, address
  LIMIT 20

-- SQL
select a.*,
       b.*,
       entity.*,
       address.*
from node a
         join edge e1 on a.id = e1.node_start
         join node entity on e1.node_end = entity.id
         join edge e2 on e2.node_end = entity.id
         join node b on e2.node_start = b.id
         join edge address_a on (address_a.node_start = a.id)
         join node address on address_a.node_end = address.id
         join edge address_b on (address_b.node_start = b.id and address_b.node_end =
            address.id)
where a.labels = ':Officer'
  and b.labels = ':Officer'
  and entity.labels = ':Entity'
  and e1.type = 'officer_of'
  and e2.type = 'officer_of'
  and address.labels = ':Address'
limit 20;
```

# 5 Discussion

## 5.1 Performance

Generally PostgreSQL performs better than Neo4j by a factor two to four in all but the shortest path query. This is similar to the findings in [PZLO17], although the performance gap has become smaller in the meantime. A possible explanation why PostgreSQL is still faster than Neo4j might just be the overall maturity of the system. Another reason could be that our queries were just not "graph-like" enough for Neo4j to really shine, with at most four edges considered in the subgraph matching query.

The one exception where Neo4j performs significantly better than PostgresSQL is in the shortest path query. This is actually an interesting case in SQL, as it was the only one where we first had to apply a few optimizations to even get the query to run below one minute. For example, PostgreSQL always materializes a common table expression and does not push predicates into the CTE itself [posb]. Therefore it is imperative that the `WHERE` clause to select the start node is within the recursive common table expression itself, or else PostgreSQL will happily calculate the shortest path from every node to every other node at once. Another problem was that there can be multiple paths with the same length between two different nodes, and PostgreSQL would continue the search from both paths, leading to a lot of duplicate work. A `SELECT DISTINCT ON` operator can solve this problem however. Both of these optimizations can be seen in the code listing in section 4.2.4

Even with all these optimizations, Neo4j performs significantly better than PostgreSQL for shortest path calculations. One possible explanation is that the former can apply specialized algorithms like Djikstra [Dij59], which are not easily doable in plain SQL. Neo4j also has the possibility to abort the query as soon as a viable solution has been found, whereas PostgreSQL will first fully evaluate its recursive common table expression and only then select a shortest path.

But even though Neo4j is faster for `shortest_path()` calculations, we would not recommend its use for the general case solely for performance. PostgreSQL is more heavily optimized and can calculate a result faster in four out of the five queries we analyzed. If an application requires extensive use of path finding algorithms, one might consider adding Neo4j for this case alone, although it might also be possible to pre-calculate results or compute them in the client application.

## 5.2 Ease of use

The graph traversal operator $(a) \rightarrow (b)$ offered by Neo4j is very useful to reduce query complexity. In order to achieve something similar in PostgreSQL, every graph "hop" needs to be modeled as two joins between the `node` and `edge` table. The effect of this can be seen nicely in the two-hop query code in 4.2.3, which has two graph traversal operators in Cypher but requires four joins in SQL. The net effect is only three lines of code for Cypher and almost 8 lines of code to achieve the same in SQL.

Another useful mechanism is the built-in `shortest_path()` function in Cypher. It is used to find connections between two nodes efficiently and with few lines of code. To achieve the same functionality in in PostgreSQL we have to implement a kind of breadth-first search with a recursive common table expression and stop when the target node has been found. The algorithm to do this in SQL is shown in section 4.2.4. Here the difference in number of lines of code between Cypher and SQL is even more pronounced - there are only 5 lines of Cypher code but 28 lines of SQL code to calculate the shortest path between two nodes.

The shortest path query also nicely shows how Cypher can optionally ignore the directionality of an edge. If we do not specify an arrowhead in the $(a) \rightarrow (b)$ operator, Neo4j will just use any edge from or to a node to calculate the shortest path. To achieve the same behavior in PostgreSQL we need to first calculate a union of the regular and reversed `edge` table and perform all joins against this derived edge table.

Finally, in the last code listing in section 4.2.5 we can see the full power of the graph traversal operator. In the form `(a:Officer)-[:officer_of]-> (entity:Entity)` it will match against a node of type `Officer` with an `officer_of` edge towards an `Entity` node. An equivalent operation in SQL requires two `JOIN` clauses and three `WHERE` conditions.

Cypher therefore provides a very convenient way to write queries for a graph database where the query language almost feels natural. This is in contrast to PostgreSQL, where traversal along an edge needs to be modelled by two `JOIN` clauses and optionally a `WHERE` clause, which is somewhat harder to understand - not just because of the query itself is longer but also because these related clauses may be located in different parts of the query. On the other hand, the results also show that it is generally possible to perform graph queries with plain SQL.

In our opinion, the only case where PostgreSQL really came to its limit is with the shortest path query. There is no direct equivalent for the `shortest_path()` operator SQL, so we had to write a recursive SQL query (which is already complex to begin with) and do a lot of additional optimizations to get the query to run within acceptable response time. This is in stark contrast to the predefined `shortest_path()` function in Cypher, which just worked out of the box with reasonable performance.

Overall, whether to use Neo4j in addition to PostgreSQL depends on the use case. The powerful Cypher query language provides a strong argument for its use, if the application has a lot of graph-related queries or if a query console is exposed to the end user for explorative queries. Otherwise a traditional database like PostgreSQL might just be sufficient, as the additional overhead of maintaining two database systems may not be enough to justify the increase in development speed.

# 6 Conclusion and future work

Graph databases are storage solutions which are optimized towards graph processing. Since they understand and implement a graph as their data model, a graph database can efficiently traverse edges along a graph or find a shortest path between two nodes.

In this report we have tried to find use cases where the graph database Neo4j might be used in addition to a relational database. We analyzed five different queries on the graph from the ICIJ offshoreleaks papers and executed them in both databases while measuring the response time. In terms of performance the result was rather disappointing, with PostgreSQL still performing better in four out of five queries. Only the query to find a shortest path between two nodes was faster in Neo4j than PostgreSQL.

On the other hand Neo4j and its Cypher query language has some powerful built-in operators, such as the edge traversal operator $(a) \rightarrow (b)$ or `shortest_path()`, which make it very expressive compared to SQL. The same query can typically be written in with less than half the amount of code than SQL, which also makes it easier to understand for a reader.

An ideal solution would be to combine the best of both worlds: by using an established relational database with years of performance optimizations but with a Cypher-like language extension for SQL. There is currently an ISO standard being developed [sql] which may bridge this gap between relational and graph databases.

We believe this resolves all remaining questions on this topic. No further research is needed [Mun].

# Bibliography


[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, dec 1959.

[Mun] Randall Munroe. Further research is needed. https://xkcd.com/2268/. Accessed: 2022-06-16.

[neo] Neo4j. https://www.neo4j.com. Accessed: 2022-06-16.

[off] The ICIJ offshore leaks database. https://offshoreleaks.icij.org. Accessed: 2022-06-16.

[posa] PostgreSQL. https://www.postgresql.org. Accessed: 2022-06-16.

[posb] PostgreSQL CTE materialization. https://www.postgresql.org/docs/current/queries-with.html. Accessed: 2022-06-16.

[PZLO17] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. Do we need specialized graph databases? Benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems*, GRADES'17, New York, NY, USA, 2017. Association for Computing Machinery.

[sql] ISO/IEC CD 9075-16.2 - Information technology — Database languages SQL — Part 16: SQL property graph queries (SQL/PGQ). https://www.iso.org/standard/79473.html. Accessed: 2022-06-19.

[VWA+14] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. Neo4j in action. chapter 11. Manning Publications Co., USA, 1st edition, 2014.