About the Fairness of Database Performance Comparisons

Uwe Hohenstein(✉) and Martin Jergler(✉)

Corporate Technology, Siemens AG, Otto-Hahn-Ring 6, 81730 Munich, Germany
{Uwe.Hohenstein,Martin.Jergler}@siemens.com

Abstract. Whenever a new database technology appears, several comparisons also come up to attest that the new database technology is better than the traditional relational one. Even more, an outstanding performance is shown quite often by conducting performance comparisons. This paper attempts to illustrate that these performance comparisons should be taken with a pinch of salt. Revisiting published statements about comparisons between the Neo4j graph database and relational systems, we investigate several causes why relational systems show a worse performance. One possible reason is – among others – applying a default database configuration or configuring the system inadequately. Next, most tests are implemented in a straightforward manner, particularly not considering alternatives or applying useful features. In order to support our findings, we use a PostgreSQL database and implement some scenarios that are commonly used in comparisons. Thereby, we invalidate some stated results about the bad performance of relational systems in those scenarios. Concluding the discussion, we present some general considerations how fairness of comparisons can be improved.

Keywords: Performance · Comparison · Benchmark · Neo4j · PostgreSQL

1 Introduction

New database technologies are regularly entering the database market to compete with traditional relational database management systems (RDBMSs). In the 90s, object-oriented DBMSs implemented a new way to store objects in C++ or Java including their related objects on physical disk more or less directly. Moreover, they claimed a much higher performance than RDBMSs and aimed at substituting them. Many benchmarks, for example 001 and 007 [4] among others, showed the superiority. A little time later, XML databases concentrate on storing and retrieving XML documents efficiently. More recently, the NoSQL movement has attracted a lot of interest since 2009. The acronym NoSQL has been selected to indicate a deviation from SQL-based RDBMSs, although NoSQL should nowadays be understood as "not only SQL" (http://www.nosql-database.org). In general, NoSQL covers new approaches macerating relational concepts such as strong consistency that seemed to be settled for decades. NoSQL comes with several categories and many products within each category. One category of products attempts to better scale with large amounts of data by using a large amount of commodity computers and to benefit from distribution. Others use different internal structures to store complex graph structures in a specialized graph database etc.

© Springer Nature Switzerland AG 2020
S. Hammoudi et al. (Eds.): DATA 2019, CCIS 1255, pp. 136–156, 2020.
https://doi.org/10.1007/978-3-030-54595-6_8

The promoters of every new technology predicate that their systems are better than traditional RDBMSs. The literature contains many discussions about technologies and comparisons between NoSQL and SQL, some of which are quite emotional and are referred to techno-religious debates by Moran [12]. In fact, those discussions are shallower than really focusing on technical issues. However, some deeper technical performance comparisons exist with many enthusiastic statements (see also [6]), for instance for the subcategory of graph databases such as:

- "The main benefit of native graph databases are performance and scalability" [9]
- "Graph databases outperform RDBMS on connected data" [10]
- "While MySQL did not finish within 2 h, Neo4j finished a traversal in less than 15 min" [14]
- "So the graph database was 1000 times faster for this particular use case" [1].

Those statements emphasize that graph databases are superior to traditional RDBMSs. Accompanying performance measurements underline the advantages. Certainly, graph databases follow an interesting approach and provide special features for handling graph structures that are useful for specific applications. However, we think that those statements are quite general and should be looked at carefully.

In this paper, we want to underline our assumption. We have already started an investigation in a previous paper [6]. This investigation is here extended by further test scenarios. Please note we do *not* want to argue that RDBMSs are still better than any other upcoming technology such as graph databases. Our intention is to investigate the unfairness of those statements and related performance comparisons. In particular, we aim at proving our point of view by measurements. We illustrate that many assumptions are disputable. For instance, many performance comparisons rely on default settings and ad-hoc configurations. While other technologies behave well, RDBMS heavily necessitate appropriate settings, e.g., for the database cache. Similarly, simple tuning such as creating indexes improve performance drastically, however, are usually not set in comparisons. Furthermore, other data structures than the obvious or traditional ones are advantageous and might affect performance just as implementing some logic in stored procedures instead of using SQL does. Moreover, the test scenarios and conditions such as testing only warm start-ups are questionable particularly if data sets do not completely fit into main memory.

We investigate in detail the huge impact of appropriately adjusting database configurations and simple tuning activities such as defining indexes compared to relying on ad-hoc configurations and settings. We also illustrate how performance can benefit from different table structures. Similarly, using stored procedures to program logic instead of sticking to pure SQL can improve performance dramatically. And finally, it must be understood what the tested scenario means. Indeed, similar scenarios, especially variants of a traversal, behave quite differently in terms of performance.

In the following, Sect. 2 gives a brief introduction into graph databases, particularly the Neo4j system, because we focus on typical Neo4j scenarios used for performance comparisons. Section 3 underpins the novelty of our investigation by discussing related work as far as it is relevant for our paper. In Sect. 4, we present our main concerns with fairness of performance comparisons and summarize some general influencing

factors. Afterwards, we elaborate in Sect. 5 upon some commonly used Neo4j test scenarios for which we have performed performance measurements using a PostgreSQL database. In addition to our previous evaluation in [6], we discuss in Sect. 6 some further scenarios where table structures play an important role. Some criteria for achieving a fair performance comparison are conducted in Sect. 7. before the analysis is concluded in Sect. 8 by outlining some future work.

2 Principles of Neo4j

Graph databases are one of the NoSQL database types. They are often considered to perfectly fit to process large connected data sets because native graph databases maintain a natural adjacency index.

Neo4j [13] is a graph database that is based on the so-called property graph model. Neo4j stores data in a graph, the most generic form of data structure, being capable of elegantly representing any kind of data.

The Neo4j data model consists of nodes and relationships. Nodes possess properties and are mainly used to represent entities.

Relationships between nodes are the key feature of graph databases, as they allow for finding related data. A relationship connects two nodes and is guaranteed to have a valid source and target node. Relationships organize nodes into arbitrary richly inter-connected structures. A relationship might possess a direction and properties.

Relationships are traversable in either direction. A traversal is a typical way to query a graph, navigating from starting nodes to related nodes by following relationships. In spite of having a direction, there is no need to duplicate relationships in the opposite direction (with regard to traversal or performance).

Both nodes and relationships may have properties. Properties are named values where the name (or key) is a string. The supported property values can be numeric, strings, booleans, or lists of any of the above types.

A label in Neo4j assigns types to nodes. A label is a named graph construct that is used to group nodes into sets; all nodes labelled with the same label belong to the same logical set. Database queries can work with these sets instead of the whole graph, making queries easier to write and more efficient to execute. A node may be labelled with any number of labels, even none. Labels are also used for defining constraints and adding indexes for properties.

The query language Cypher provides a declarative way to query the graph powered by traversals and other techniques.

3 Related Work

In the literature, several performance investigations can be found. Indeed, we extracted enthusiastic statements and test scenarios from these sources in order to qualify them by using own implementations.

For example, Khan [10] explains that the technology of graph databases is better than RDBMSs because RDBMSs require joins that are bad for graph structures. The database schema he used consists of Employees (E), Payments (P), and Departments (D), which are related by one-to-many relationships E-P and D-E, resp. The scenario selects two departments first and determines the related employees (via D-E) and afterwards the payments (via E-P). The complexity is evaluated in Big-O notation. Depending on the join strategy used by the optimizer, RDBMSs achieve a complexity of $O(|E|+|P|)$ with hash joins and $O(|E|*|P|)$ with nested loop joins while Neo4j achieves a constant $O(k)$ behavior. The internals of Neo4j leading to a constant behavior are described by "*Using hash indexing this gives O(1). Then the graph is walked to find all the relevant payments, by first visiting all employees in the departments, and through them, all relevant payments. If we assume that the number of payment results are k, then this approach takes O(k).*" Unfortunately, it is not explained how "visiting all employees" is implemented in Neo4j, what is special to the internal Neo4j data structures leading to the constant behavior compared to hash indexes in RDBMSs.

Adell [1] uses a friendship relationship connecting people to their friends. The friendship database is filled with 1,000,000 users with an average of 50 friends. The implemented scenario takes two arbitrary people and detects whether both are direct or indirect friends considering 4 or fewer hops. While the RDBMS was running several days and was finally stopped, Neo4j required only 2 ms for the check.

In a similar setup, Rodriguez [14] uses a database with 1,000,000 nodes and 4,000,000 arbitrarily selected edges with an average fan-out of 4. Traversing from a selected starting node, all related nodes that can be reached by 1 to 5 hops are determined. The performance comparison shows that Neo4j is more than twice as fast for 4 hops. While Neo4j found the nodes for 5 hops in 14.37 min, MySQL was stopped after 2 h.

Baach [2] uses two data sets with 100,000 and 1,000,000 nodes, resp. In either setup, each node possesses exactly 50 edges. Compared to the previous tests, Baach's implementation does not query all the nodes, but only *counts* the number of friends up to 5 hops. Surprisingly, Neo4j was about 6 times slower than MySQL. Baach presumes that the Cypher query language (CQL) used by him performs much worse than the Pipes framework used by Rodriguez [14]. Baach's opinion is that a comparison of query languages, SQL and CQL, is fair. Forcing RDBMS to use SQL while allowing Neo4j to benefit from an optimized Pipes framework seems to be unfair. Moreover, Baach spent some time configuring MySQL appropriately, which might be another explanation for the results.

Vicknair et al. [15] build a direct acyclic graph with data sets of 1,000, 5,000, 10,000 and 100,000 nodes. They perform several tests that traverse the graph and count the nodes, for 4 and 128 hops and determine all the orphan nodes. Moreover, they consider the payload, e.g., counting the number of nodes that exceed a certain payload value. Since the data sets are small, tests mainly measure in-memory processing. This particularly results in execution times less than 200 ms, without revealing notable differences between Neo4j and MySQL.

Real application scenarios are taken by the following work. Joishi and Sureka [8] and Martinez et al., [11] both compare the performance of MySQL and Neo4j. [8]

take two process-mining algorithms, e.g., analyzing causal dependencies between actors in carrying out a business process and finding similarities between actors based on the intersection of activities. It turned out that Neo4j attains a performance boost of a magnitude of 7× over MySQL for the first scenario. However, Neo4j is 32 times slower than MySQL is for the similarity scenario.

Martinez et al. [11] perform 12 multi-join queries of a health application for three randomly generated data sets with 1,000, 10,000 and 100,000 entries. MySQL is faster than Neo4j for most of the queries but has a poor performance for larger data sets. It is important to note that indexes were not added in both database systems.

[9] compares Oracle 11g and Neo4j using a Medical Diagnostic System. The data set comprises about 28,000 patients, 625,721 patient visits, 869,666 patient-IssueMed records, to mention the main tables. Five count queries join two or three tables. While Oracle performs queries in a few seconds (depending on the query), Neo4j requires about 0.3 s.

All this work focuses on technical comparisons between technologies and products. However, this paper mistrusts the fairness of those comparisons. Indeed, only little work on unfairness of performance comparisons has been published so far. Baach [2] considers a comparison between a query language (SQL) and an optimized Neo4j framework instead of the Cypher query language as "*Comparing Neo4j to MySQL without the use of Cypher is comparing apples and oranges*". The Hacker forum [5] contains some critical statements remarking that many comparisons are bootless. To our knowledge, there is only one paper [7] that criticizes statements such as "ODBMSs are faster by factor 100" made in an evaluation of object-oriented database management systems (ODBMSs) for the OO7 benchmark [4]. In order to critically reflect the OO7 results, [7] conducted a case study. They transformed a real Oracle/C application into an ODBMS/C++ application and measured the performance of realistic scenarios for different ODBMSs. The results were stupendous and diverged from OO7 results since showing that only a single ODBMS-based implementation has the potential to be faster than the original Oracle-based solution, while one ODBMS was definitively much slower. Concluding that the best benchmark is the application itself, the paper suggests a methodology for deriving application-specific benchmarks.

4 Unfairness of Performance Comparisons

In the following, we explain why we think that published performance comparisons are unfair and should be seen with caution.

4.1 Scope of Comparison

As already outlined above, the existing literature on Neo4j contains many exciting claims about its performance. Khan et al. [9], for instance, shows that graph databases are superior to RDBMSs by applying a theoretical comparison of internal algorithms based on Big Os. However, he does not explain why the internal structures of Neo4j are better. A comparison of *technologies* at that level is neither valid nor expedient.

Similarly, a general comparison between a specific product X and class of systems as presented in [1] is wrong per se: Showing that Neo4j is faster than MySQL does not prove that Neo4j is faster than any relational system. There are many other relational products, too.

4.2 Small Test Data Sets

Oftentimes, performance tests are executed with rather small data sets, for instance, 1000, 5000, 10,000 records [15], and also graphs with 100,000 nodes [2] are not really large. The result of such experiment setups is basically a test of the in-memory capabilities of a system: All the data fits into the accordingly sized memory.

While such evaluation scenarios might be representative for certain applications, where all data fits into main memory, the results, however, cannot be generalized and applied to application scenarios with larger data sets, where cache misses are the norm and disk access is heavily involved.

4.3 Warm Start

In many systems, the first few query executions are slow because data must be fetched from disk to populate the cache and the query execution plan must be derived. Further executions, also with different values, are faster because the execution plan is already available and data is in the cache. For this reason, performance comparisons like [2] often first initialize the cache by fetching all the needed data in a warm start. Moreover, the cache size is perfectly adjusted. This sounds reasonable at a first glance, but usually not a few (tested) tables are used and accessed in real applications. Accesses to other tables will interfere and disturb the first cached data but remain untested. Hence, a warm start is representative only if all the data – not only that used in tests – fits into memory completely.

4.4 Using Standard Configurations

Database management systems have many configuration parameters such as the database cache size. Those parameters have a strong impact on performance. Looking into performance comparisons, however, mostly standard configurations are applied. For instance, Martinez et al. [11] notes that "The deployed database servers were not optimized" and "No index was added to the basic implementation".

Usually, the cache size, more precisely the size of the Java Virtual Machine, is configured for Neo4j to be large enough to keep most data in client-side memory. Instead, RDBMSs execute queries at server-side, which requires a different kind of cache configuration. Further parameters, for instance, the space for temporal data, affect sorting and eliminating duplicates. In fact, having configured all those parameters appropriately improves performance enormously.

4.5 Over-Tuning

Over-tuning means that a benchmark focuses on only a few scenarios of an application, which are then highly tuned for exactly that scenario. Those scenarios are over-tuned since other scenarios are not taken into account and might suffer from that particular tuning. A typical case is to focus on queries and maybe inserts while ignoring updates and deletes or mixed operation sets.

4.6 Artificial Test Scenarios

Most of the published benchmarks and comparisons are rather artificial: They use a configurable number of nodes and relationships (e.g., Vicknair et al. [15]), maybe a configurable fan-out for relationships. Using such a data set, they take traversals along connections between nodes as the typical use case. The main reason is conduct multiple tests with varying parameters in an easy manner thus keeping the effort for implementing and performing tests low. Barry [3] states that it is easy to spend $100,000 for benchmarking, especially if several systems should be compared with presumably different implementions. Those simplified tests help to reduce the implementation effort.

However, it is arguable whether those generic tests, which basically abstract from concrete applications, are able represent the behaviour of a concrete application by just configuring the parameters. Results of performance comparisons are only representative if tests coincide with the application in mind, i.e., if the amount of data and the distribution is similar as well as the access patterns. Or in other words, the tested operations must reflect the characteristic accesses of a given application. Configuring a couple of parameters generally do not let a benchmark represent the demands of a specific application. Indeed, benchmarks for graph databases are rather artificial in such a sense. Thus, it is usually impossible to adapt tests and their results by simply changing a few factors such as the number of nodes [2, 15] or the fan-out of relationships. Such a parameterization does not help to let a benchmark become more representative. It is a valid question whether simple and slightly configurable tests could be representative for an application at all.

Furthermore, performance comparisons focus on queries (i.e., traversals) and sometimes inserts. Corresponding tests are conducted independent of each other. Only little work considers updates and deletes or interleaving all those operations in one mixed scenario such as real-life applications do. Consequently, just a few features are compared in an isolated manner.

As already mentioned, traversals are the main scenario for graph databases. However, there are various interpretations what a traversal is. A traversal can be understood as selecting all connected nodes via up to n hops, or only counting the number of related nodes. Sometimes, all possible connections between two nodes are determined, while some other tests simply detect whether two nodes are related via up to n hops. These scenarios look quite similar, but we will illustrate later how huge differences in performance are. Anyway, such tests are scenarios that are advantageous for graph databases.

4.7 Implementation Issues

Most performance comparisons implement scenarios in pure SQL for RDMSs, ignoring the possibility to use stored procedures, for instance. Neo4j also provides a query language named Cypher QL, which is however often not used. Instead, tests are implemented in the procedural Neo4j Pipes framework which seems to yield better results as the test of Baach [2] reveals. Baach implemented a common scenario with the Neo4j Cypher query language, which was much slower than a comparable MySQL solution. Obviously, the Cypher language does not perform as well as the Pipes framework. Baach and we, too, think that comparing SQL with the Pipes framework is unfair. At least, RDBMSs should be given the freedom to choose an implementation language.

Similarly, most test implementations for a RDBMS apply a straightforward database schema. However, there are often other options using specific features, which should be tried out to improve performance.

4.8 Data Distribution

Beyond the chosen test scenarios and test data, especially the number of nodes and the number of edges, the distribution of data for individual nodes plays a crucial role for performance, too. For instance, the selected start node might have an impact on the result size because each node has a different number of related nodes over n hops. The best implementation solution can change when using different start and end nodes!

4.9 Evaluation

Even if a benchmark seems to be representative, the evaluation results may be unfair and may diminish the value of the results. Typically, several test scenarios for traversals, inserts, removals, queries are performed, being simple in nature and executed in isolation and independent of each other. Also, each test is often parameterized leading to several results.

Thus, a benchmark comprises a collection of independent results. This means particular performance values have to be somehow aggregated in order to get an overall result. Detailed analyses are possible, but it is questionable how to correctly extrapolate from results for simple operations to complex logic of the real application. A particular system is able to win a comparison by just aggregating and interpreting the results in the right way – a system might have won most test cases, have best average over all the test cases, be leading for some “relevant” weighting of test cases etc.

5 Performance Evaluation

The previously mentioned concerns frequently lead to published statements showing that graph databases are 100 times faster than traditional relational ones [1]. Indeed, these statements are quite general and must be seen relative to the test scenarios and the test implementations.

Thus, using common scenarios for Neo4j, we have implemented and performed some tests with a PostgreSQL database. We intentionally used an older version 9.5 because

several comparisons of RDBMS vs. Neo4j are also older. This means that we did not take any advantage of using the most recent state of relational technology.

5.1 Scenarios

The published performance comparisons for Neo4j mainly use three "traversal" scenarios:

1. Scenario **ALL(n)** starts with a random node and determines all nodes reachable by less than n hops.
2. The **PATHS(n)** scenario computes only the paths between two given nodes related by k hops for $k \leq n$.
3. **EXISTS(n)** is not interested in the concrete paths, but checks whether two given nodes are related by k hops, $k \leq n$.

5.2 Test Conditions

All the tests we performed used the same laptop running Windows 7 with a dual-core processor, 12 GB of RAM, and a 465 GB SSD disk. We can certainly state that the machine is not oversized. Moreover, the tests ran in an isolated environment, i.e., no other application were running during our test, particularly no concurrent database accesses. Each test was performed 3 times. The average of measurements was taken.

It is important to note that we did not directly compare Neo4j with PostgreSQL since our main statement is that any comparisons are unfair anyway. Hence, we aim at putting some published statements into perspective.

We experimented with a database that contains 500,000 nodes with 50 edges to other randomly selected nodes. That is, using the commonly applied database schema of Adell [1] for handling friendship relationships with one single table `Friends(id int, friend int)`, we keep 50*500,000 = 2,500,500 records in that table. This is more data than performance comparisons typically use. Both fields `id` and `friend` comprise the primary key of the `Friends` table. Each person (or node) is identified by a unique `id`; the foreign key `friend` is a reference (or edge) to a friend's id, i.e., all those records that refer to the same `id` form the collection of friends for that person.

5.3 Impact of Indexes for the PATHS Scenario

The goal of this first test is to illustrate the huge impact of setting up simple indexes. The test starts with a PostgreSQL standard configuration, i.e., particularly a very small database cache of 128 MB. The already mentioned `Friends(id, friend)` table structure of Adell [1] has been used for representing a friendship relationship.

We have implemented the scenario PATHS(5); the corresponding SQL query is presented in Fig. 1.

```
select f1.id, f2.id, f3.id, f4.id, f5.id, f5.friend
from Friends f1
join Friends f2 on f2.id=f1.friend
join Friends f3 on f3.id=f2.friend
join Friends f4 on f4.id=f3.friend
join Friends f5 on f5.id=f4.friend
where f1.id = :s and f5.friend = :e
union
select f1.id, f2.id, f3.id, f4.id, f4.friend, null
from Friends f1
join Friends f2 on f2.id=f1.friend
join Friends f3 on f3.id=f2.friend
join Friends f4 on f4.id=f3.friend
where f1.id = :s and f4.friend = :e
union
select f1.id, f2.id, f3.id, f3.friend, null, null
from Friends f1
join Friends f2 on f2.id=f1.friend
join Friends f3 on f3.id=f2.friend
where f1.id = :s and f3.friend = :e
union
select f1.id, f2.id, f2.friend, null, null, null
from Friends f1
join Friends f2 on f2.id=f1.friend
where f1.id = :s and f2.friend = :e
union
select f1.id, f1.friend, null, null, null, null
from Friends f1 where f1.id = :s and f1.friend = :e;
```

Fig. 1. SQL statement for Scenario PATHS(5).

The query computes the complete paths between a start and end node `:s` and `:e`, resp. The result includes all the intermediate nodes. As already mentioned in Subsect. 5.2, the first execution was done with a cold start after a restart of the computer. The same query was then immediately executed additional three times in the sense of a warm start taking the average of numbers ("Ø"); "stdev" denotes the standard deviation. In addition, the same query has been run another time with different start and end points (i.e., the "Second" column). Results are summarized in Table 1 for cold and warm start.

Table 1. PATHS(5) results for investigating indexes.

Index	Cold start		Warm start			
	First [s]		First [s]		Second [s]	
	Ø	stdev	Ø	stdev	Ø	stdev
No	62.88	3.26	59.67	0.60	60.25	1.74
Yes	23.78	5.27	4.54	1.55	6.43	1.17

The test results underline the huge impact of indexes on performance. In fact, it is not surprising that indexes are essential for RDBMSs to achieve an acceptable performance. The difference is even higher for a warm start. A lack of basic indexes – maybe due to a too naive implementation or a standard configuration (cf. Subsect. 4.4) – can heavily falsify benchmark results.

Due to the obvious need for indexes, all the further tests will be done with indexes.

5.4 Cold/Warm Start

Table 1 also shows the difference between a warm and cold start for PATHS(5). Most performance comparisons only apply a warm start scenario. The intention of this test is to investigate how the system behaves in case of freshly loading data from disk. This is relevant if not all the data fits in memory, which might be the case for larger applications.

The difference between cold and warm start is small for not applying any indexes; then, table data must be loaded anyway. However, there is a large difference between cold and warm start when using indexes.

Consequently, a test should not only be restricted to warm start setups (cf. Subsect. 4.3).

5.5 Implementation Variants

Beyond simple indexes, query tuning can have a large impact. We take Scenario EXISTS(5) that determines the existence of connections between two nodes. In contrast to the PATHS scenario, where all existing paths are collected, EXISTS only returns a Boolean – connected or not. We checked three possible queries:

1. The simplest one executes the query for PATHS (cf. Fig. 1). If the result is not empty, then there is a connection.
2. We can also add a `LIMIT 1` clause at the end of the previous query in order to limit the result to just a single first connection or none.
3. In another alternative, a `LIMIT 1` is added to each sub-query. This let the execution stop early after the first hit:

```
(select f5.friend … limit 1)
union … union
(select f1.friend … limit 1)
```

Table 2 illustrates the enormous speed-up for the last variant 3). Hence, searching for alternative implementations or queries can be very effective. Indeed, this modification is quite simple, but obviously very effective. Taking one straightforward solution does not give best results (cf. Subsect. 4.7).

Please note that there are no connections for 1 to 4 hops in our test data. Consequently, all the sub-queries are executed.

Table 2. Results for implementation variants.

Variant	Cold [s]		Warm [s]	
	Ø	stdev	Ø	stdev
1)	23.78	5.27	4.54	1.55
2)	19.81	1.61	4.79	1.07
3)	2.39	0.13	0.057	0.02

5.6 Larger Cache Size

As already mentioned, the default cache in PostgreSQL is far too small: 128 MB are just able to keep 5% of our table data (864 MB) and index data (1607 MB). As a consequence, many reads have to access the disk.

In another test setup, we thus increase the cache size to 1024 MB, i.e., by factor 8. Anyway, this is still not enough memory to keep all the table and index data.

Table 3. Results for smaller vs. larger cache.

Cache size	Cold [s]		Warm [s]	
	Ø	stdev	Ø	stdev
128 MB	23.78	5.27	4.54	1.55
1024 MB	25.58	0.92	0.46	0.03

The results for a larger cache are presented in Table 3 for the PATHS(5) scenario. For a cold start, the difference between a default and large cache is very small because data must be fetched from disk anyway. However, for a warm start, we detect more than factor 8 of speed up. This means that sticking to default configurations does not yield correct performance numbers (cf. Subsect. 4.4).

Please note there are many further database parameters that can be tuned. For example, increasing the temporal space can speed up sorting and duplicate elimination.

5.7 Different Implementations and Structures

Nearly all published performance comparisons use some straightforward table structures and pure SQL for "implementing" the comparison tests. However, there are alternatives for data structures and/or implementing the computation logic, which are usually not tried out (cf. Subsect. 4.7). A table `Friends3(id int, friend1 int, friend2 int, friend3 int)` can be used to store up to three friends in one single row. If there are less than three friends, some columns are left empty. In case of more than three friends, one or more continuation records are stored with the same `id`.

More flexibility is obtained by using the native PostgreSQL array type. We use a table `FriendsWithArray(id int, friends int[])` with an array-valued column instead of the "standard" table `Friends(id, friend)`. Now, a single record represents one node. The array can keep an arbitrary and varying number of friends.

Using this table structure, we perform Scenario ALL. The following query retrieves the nodes reachable by less than four hops starting with a person `:p`:

```
select distinct f1.id as fid, f1.friends              -- (1)
into tmp3
from FriendsWithArray f1
where f1.id = :p;
insert into tmp3
select distinct f2.id, f2.friends
from FriendsWithArray f1, FriendsWithArray f2,
     generate_subscripts(f1.friends,1) i1,
where f1.id = :p and f1.friends[i1] = f2.id
union ... union
select distinct f4.id, f4.friends
from FriendsWithArray f1, FriendsWithArray f2,
        generate_subscripts(f1.friends,1) i1,
        generate_subscripts(f2.friends,1) i2,
        FriendsWithArray f3, FriendsWithArray f4,
        generate_subscripts(f3.friends,1) i3,
where f1.id = :p and f1.friends[i1] = f2.id
and f2.friends[i2] = f3.id and f3.friends[i3] = f4.id;
select distinct t3.id, friends[i] as friend           -- (2)
into tmp4
from tmp3 t3, generate_subscripts(t3.friends,1) i
```

Fig. 2. Query for the ALL(4) scenario with arrays.

`f1.friends` is an array that contains the friends of the 1st hop. The built-in function `generate_subscripts` is applied to an array-valued column and returns a set of indices to which a variable `i` can then be bound. The term `friends[i]` uses the variable `i` to access the i-th field in the array and is then used to join array elements (i.e., friends) and persons' ids, e.g., `f1.friends[i1]=f2.id`. The query of Fig. 2

stores the result in a temporary table `tmp3` (cf. (1) in Fig. 2). A stored procedure takes the temporary table and performs the next step (2) to unnest the node ids. The result is again stored in a temporary table `tmp4` to compute ALL(4).

This query can be extended to implement the ALL(5) scenario, i.e., all nodes with less than 5 hops, by two additional steps as shown in Fig. 3:

```
select f5.id, f5.friends
into tmp5
from FriendsWithArray f5, tmp4 t4
where f5.id = t4.friend;
select distinct t5.friends[i]  -- unnest
from tmp5 t5, generate_subscripts(t5.friends,1) i;
```

Fig. 3. Query for the ALL(5) scenario with arrays.

Table 4. Results for Scenario ALL.

Variant [s]	Small cache		Large cache	
	Ø	stdev	Ø	stdev
ALL(4) “Old”	16.43	3.21	16.28	2.87
ALL(4) “New”	8.76	3.54	4.54	1.98
ALL(5) “New”	28.67	3.26	24.55	1.75

Table 4 displays the improvement against the usual implementation. The test ALL(4) “New” refers to the new logic, while ALL(4) “Old” is computed with a single SQL query similar to Fig. 1, however, returning related nodes for a start node `:s` instead of paths (Fig. 4):

```
select f5.friend
from Friends f1
join Friends f2 on f2.id=f1.friend
join Friends f3 on f3.id=f2.friend
join Friends f4 on f4.id=f3.friend
join Friends f5 on f5.id=f4.friend
where f1.id = :s
union
select f4.friend from Friends f4 ... where f1.id = :s
union
select f3.friend from Friends f3 ... where f1.id = :s
union
select f2.friend from Friends f2 ... where f1.id = :s
union
select f1.friend from Friends f1 where f1.id = :s;
```

Fig. 4. Traditional 1-SQL-Query for the ALL(4) scenario.

There is a large difference for ALL(4) of factor 2 for the small and factor 4 for the large cache. What is even more important is that the computation of related nodes over 5 hops, i.e., ALL(5), is possible in the new implementation with about half a minute – as opposed to several hours as stated in [1, 14]. Hence, the new approach is performing well, however, the improved performance gain is paid by some drawbacks. The table violates the first normal form. Anyway, even the SQL standard introduced non-first-normal-form features. Moreover, the table structure is still easy (maybe even easier) to understand. What is more critical is that inserts and deletes become more complicated. Inserting a new relationship must add the friend to the array. Deleting a friend relationship must remove the friend from the array. A stored procedure might help to handle the logic.

5.8 Data Distribution

Another point is about choosing the (right) test data. There is again some impact on performance (cf. Subsect. 4.8). For instance, if we use PATHS(5) with different start and end nodes, we obtain different numbers of connections, and as a consequence, a different size of the result set. Table 5 illustrates the impact of using different start and end nodes. The smaller the number of retrieved connections is, the faster the query performs.

Table 5. Results for different result sizes.

Result size	Cold [s]		Warm [s]	
	Ø	stdev	Ø	stdev
479 recs	15.18	1.27	3.32	0.64
797 recs	23.78	5.27	4.54	1.55

The order of sub-queries is also relevant for the implementations of Scenario EXISTS(5) in Subsect. 5.5. If we know that there are no hop-1 and hop-2 but hop-3 connections between nodes in the test data, we should put the hop-3 sub-query in front in order to avoid the execution of sub-queries immediately. That is, knowing the data set, the implementation can be "improved". This is also a form of over-tuning (cf. Subsect. 4.5) by consciously "tuning" the order of sub-queries according to data.

In order to elaborate more on another facet of Subsect. 4.8, we use a larger database with 5,000,000 nodes each having four randomly chosen friends. Using the bad performing One-SQL-statement of ALL(n) based upon Fig. 1 with a large cache yields the results for n = 9 and n = 10 as shown in Table 6. Test ALL(9) returns 334,247 rows in 8 s, whereas ALL(10) yields 1,172,908 rows in 26 s with a warm start. With a cold start, execution times are a little slower. Anyway, even n = 9 and n = 10 achieve moderate execution times with the non-optimized One-SQL-statement query and the standard table structures despite the higher number of hops.

That is, compared to the previous database, the fan out seems to be one decisive factor for performance results, too (cf. Subsect. 4.8).

Table 6. Results for ALL scenario (5,000,000 nodes, large cache).

	Cold [s]		Warm [s]		Number of returned values
	Ø	stdev	Ø	stdev	
ALL(9)	20.66	0.22	8.01	0.05	334,247
ALL(10)	37.44	3.86	26.40	0.41	1,172,908

5.9 Differences in Traversal Scenarios

In the previous tests, we have considered different scenarios that implement traversals. However, traversals have different interpretations, which we named ALL, PATHS, and EXISTS: returning all related nodes, all the paths between two given nodes, or determining whether a relationship exists between two persons. In particular, we recognize quite a different performance behavior of the interpretations of traversal scenarios. In a warm start, PATHS(5) is performed in about 5 s, while EXISTS(5) is even much faster (a few milliseconds) using an optimized query. However, finding nodes in ALL(5) is slow with half a minute in an optimized version while the straightforward solution takes some hours.

This discussion underlines the importance of the chosen scenario and its semantics, even if scenarios seemingly look very similar (cf. Subsect. 4.6).

5.10 JDBC Configuration

So far, we have performed tests with the PostgreSQL pgadmin console for interactive query processing, i.e., executing SQL or stored procedures interactively with a query browser. However, database access will usually be done in a programming language using JDBC, ADO.NET, an object/relational mapping tool such as Hibernate etc. Hence, another factor enters the game having impact on performance comparisons.

We consider fetching the query result in Java with JDBC. Again, there are several options. One important parameter in JDBC is the fetch size, which can be set by `setFetchSize(n)`. The fetch size determines how many records are transferred from the database server to the client program: If a record is requested by the client, a bulk of n records is physically prefetched and transferred, already serving this and the next $n - 1$ successive requests, too. Any further request will then fetch the next bulk of n records.

We have used ALL(4) and executed the "Old", non-optimal query with different fetch sizes. The results are summarized in Table 7. The query executed in 30 s with a

Table 7. Results for ALL(4) scenario with different JDBC fetch sizes.

JDBC fetch size	Warm start	
	Ø	stdev
1	30.468	1.04
1000	18.309	1.21

size of 1 (typically being the default) and 18 s with a size of 1000. This a huge difference, especially since the query execution itself consumes about 15 s.

This a huge difference, especially since the query execution itself consumes about 15 s. Again, relying on some defaults affects the performance negatively (cf. Subsect. 4.4).

In case of Hibernate, a query is typically executed with

```
List result = session.createQuery("<query>").list();
```

Similar to JDBC, a fetch size can be configured with the same effect and behavior. It is important to note that the whole result is materialized in the `result` Java list. This obviously requires a lot of memory, especially for ALL(4) result sets with about 500,000 records, and even more for ALL(5).

An alternative is to apply some streaming of results where data is returned in a cursor-like fashion.

6 Further Considerations About Data Structures

There are often alternatives or optimizations as Subsect. 5.7 has already illustrated by using proprietary array-based columns. As another example, we want to reveal the typical relational structure for storing trees:

```
Tree(id, properties, fatherId)
```

The above table structure is commonly used but is often considered as inappropriate for querying whole subtrees. Starting with the sons of a given father node, the sons of each son have to be found recursively until the leaves are reached:

```
with recursive Sons (id, fatherId, lev) as
(select id,fatherId,1 from Tree where fatherId = :x
 union
 select t.id, t.fatherId, s.lev+1 from Tree t, Sons s
 where t.fatherId=s.id
) select * from sons;
```

Such a query is said to perform badly due to the need of recursion. Another structure has been suggested in the literature:

```
Tree(id, properties, fatherId, firstLeft, lastRight)
```

`firstLeft` and `lastRight` are auxiliary columns that help to determine subtrees. Figure 5 illustrates the principle of setting these columns.

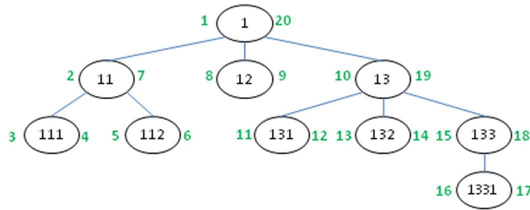

Fig. 5. Improved tree data structure for retrieval.

The idea is that `firstLeft` (left hand side of a node) and `lastRight` (right hand side) of a node determine the range of its subtree. To set the values, the tree is traversed in a depth-first manner, thereby passing each node twice, left and right. For example, node 13 possesses `firstLeft=10 and lastRight=19`. This means that the subtree of node 13 has values `firstLeft>=10 and lastRight<=19`. The subtree of node 11 lies between 2 and 7 analogously. Hence, in order to return the subtree of a node `:x`, the following query can be issued without any recursion:

```
select sub.*
from Tree t, Tree sub
where t.Id=:x -- to retrieve starting node
and sub.firstLeft >= t.firstLeft
and sub.lastRight <= t.lastRight
```

The condition for checking whether a node is a leaf node is simply:

```
sub.lastRight - sub.firstLeft = 1.
```

The total number of nodes can be computed by: `root.lastRight /2.`

This structure is supposed to optimize the performance of common tree operations at the expense of maintaining the ordering for inserts, deletes, and movements by updating `firstLeft` and `lastRight` columns of many nodes.

Another possibility is to attach a path to each node. Starting with 1/ for the root, the sons of the root obtain a path 1/1, 1/2 etc. The next level contains three path elements accordingly. Concerning retrieval, the subtree for node i/j/k can be queried by LIKE 'i/j/k/%'. The effort to maintain the schema is moderate since deletions and updates do not require any action. Moving a node in the hierarchy requires computing and changing the path of that node only. For inserts, the largest value for siblings has to be determined and incremented to a path.

We made some performance measurements for retrieving subtrees at various levels. The results are summarized in Table 8 for a tree fanout of 5 and a depth of 10 with a total number of 12,207,031 nodes. The recursive query turned out to be the fasted for choosing a start node at any level, maybe because of the small fanout, the number of levels, or the equal distribution (see Subsect. 4.8). One particular reason why the path expression solution behaves worse is that the index on the path expression is not used; unfortunately, PostgreSQL cannot be forced to use the index.

Table 8. Variants for determining subtrees.

Level	Returned nodes	Variant [ms]		
		Recursive	Left/right	Path
3	488280	12,695	29,037	35,140
4	97655	2,581	6,466	11,821
5	19530	556	1,867	7,413
6	3905	103	850	6,679
7	780	40	694	6,371
8	155	13	592	6,233

7 Some Thoughts About Improving Fairness

So far, we have discussed some typical performance scenarios that have been published for graph databases by revealing tests with a PostgreSQL database. As shown, it is easily possible to achieve different – better – results than presented in the literature.

Now, we turn to the question how to improve fairness of performance comparisons.

First of all, it is an absolute precondition for fairness to supply the same environment, especially the same amount of resources such as hardware, processor, network, operating system, disks, RAM etc. Furthermore, there should be the same test isolation and the same overall test conditions etc.

However, it is becoming unfair if the size of memory for the database system is pinned. For instance, Neo4j is Java-based and execute queries at the client, making a lot of memory in the JVM more advantageous, while RDBMSs process queries in the server instead. Hence, a smaller JVM spares more RAM for the database system. Since the settings do not only depend on an application and its data, but also on the type of DBMS resources, resources cannot be set equally for all the test candidates.

The rules for conducting comparisons are also important. In principle, every test should implement the *same universe of discourse* with the same functionality. Conditions must not be too restrictive. But a tester should not be forced to use a specific database schema, a query language etc. For example, benchmarks for object-oriented database systems, e.g., [4] often dictated testers the same neutral implementation. Instead, there should be freedom for using SQL or not.

The execution of benchmarks or tests often relies on default configurations. Parameters such as the database cache size are set to default or configured at the most at good guess; tuning is more or less neglected. The OO7 benchmark [4] sanctifies this by noting that normal programmers are unable to tune a database system effectively. In our opinion, this proceeding is arguable if ease of use is not an issue. Most database systems *do* require an appropriate tuning in order to achieve an acceptable performance. Especially, RDBMSs heavily rely on indexes for primary and foreign key. Indeed, we have demonstrated how even simple tuning measures like creating indexes already improve performance substantially. Usually, there are much more screws to turn. This potential must be used for sustainable results.

Performance comparisons are fair if each tester obtains infinite time for implementing and also improving and tuning. However, since this seems to be infeasible in practice, the following attenuation makes sense: Each tester of a particular system should obtain equal and adequate time. The available time limit must not be too short. This enables programmers to try out several variants and to tune the overall system instead of choosing too straightforward, often naive solutions. If implementers do not possess the same degree of skills and knowledge, the skills should also be taken into account for restricting the time.

Finally, realistic and holistic test scenarios are indispensable. That is, scenarios should cover all the relevant use cases, particularly, representative and complete use cases with a mixed set of operations. Especially holistic scenarios help alleviate the risk of over-tuning or overemphasizing a particular operation, while ignoring other non-tested ones.

8 Conclusions

Whenever a new database technology arises, publications about more or less technical performance comparisons usually occur, too, proving that these technologies are superior to traditional relational database systems (RDBMSs) for various test scenarios.

In this paper, we focused on some exciting statements made for the graph database Neo4j. There are many published comparisons showing that Neo4j has huge performance gains compared to relational systems. We think that those statements are mostly unfair due to several reasons, which we discussed in the paper in depth. Moreover, those published performance comparisons motivated us to revisit some test scenarios common to those comparisons and to re-implement them using a PostgreSQL database. As one major result, this paper illustrated that performance varies drastically depending on several factors such as configuration, choice of database schemas, amount of data, simple tuning activities etc. for those scenarios for which RDBMSs were "proven" to be unsuitable and thus leading to a bad performance in published performance comparison. Instead, we were able to achieve good results for those critical scenarios quite easily by applying simple measures. As a consequence, selecting a database system should not be based upon blindly trusting published comparisons.

However, it was neither our intention to perform yet another performance comparison (which becomes unfair), nor to prove that relational systems are still better than graph databases. The message should not be that Neo4j is not faster than MySQL, Oracle,PostgreSQL or any other product, but rather than each system can be tuned for particular use cases in order to achieve an acceptable performance.

The main intention of this paper was thus to underline the necessity of independent benchmarks and comparisons with realistic and application-specific scenarios. A deeper investigation is indispensable for obtaining reliable and representable results. To this end, we tried to collect some recommendations to achieve fair performance comparisons.

In our future work, we intend to expand our investigation from graph databases to other NoSQL categories. In particular, we aim at analyzing other advantages of NoSQL products such as enhanced scalability by distribution, better flexibility by means of schema-less data structures etc.

References

1. Adell, J.: Performance of graph vs. relational databases (2013). https://dzone.com/articles/performance-graph-vs. Accessed 26 Nov 2019
2. Baach, J.: Neo4j performance compared to MySQL (2015). https://baach.de/Members/jhb/neo4j-performance-compared-to-mysql. Accessed 26 Nov 2019
3. Barry, D.: Should you take the plunge? Object Mag. **3**(6), 1994 (1994)
4. Carey, M., DeWitt, D., Naughton, J.: The OO7 benchmark. ACM SIGMOD **22**, 12–21 (1994)
5. Hacker. NoSQL vs. RDBMS: let the flames begin (2010). https://news.ycombinator.com/item?id=1221598. Accessed 26 Nov 2019
6. Hohenstein, U., Jergler, M.: Database performance comparisons: an inspection of fairness. In: 8th International Conference on Data Science, Technology and Applications DATA 2019, Prague, Czech Republic, June 2019
7. Hohenstein, U., Pleßer, V., Heller, R.: Evaluating the performance of object-oriented database systems by means of a concrete application. In: 8th DEXA Workshop, Toulouse (1997)
8. Joishi, J., Sureka, A.: Graph or relational databases: a speed comparison for process mining algorithm. In: Proceedings of 19th International Database Engineering & Applications Symposium, Yokohama (2015)
9. Khan, W., Ahmed, W., Shahzad, E.: Predictive performance comparison analysis of relational & NoSQL graph databases. Int. J. Adv. Comput. Sci. Appl. **8**(5), 523–530 (2017)
10. Khan, Q.: Why graph databases outperform RDBMS on connected data (2016). https://dzone.com/articles/why-are-native-graph-databases-more-efficient-than. Accessed 26 Nov 2019
11. Martinez, A., Mora, R., Alvarado, D., et al.: A comparison between a relational database and a graph database in the context of a personalized cancer treatment application. In: Proceedings of Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City (2016)
12. Moran, B.: RDBMS vs. NoSQL: and the winner is .... (2010). http://www.itprotoday.com/microsoft-sql-server/rdbms-vs-nosql-and-winner. Accessed 26 Nov 2019
13. Neo4j. Neo4j (2019). www.neo4j.org
14. Rodriguez, M.: MySQL vs. Neo4j on a large-scale graph traversal (2011). https://dzone.com/articles/mysql-vs-neo4j-large-scale. Accessed 26 Nov 2019
15. Vicknair, C., Macias, M., Nan, X., et al.: A Comparison between a graph and a relational database: a data provenance view. In: Proceedings of 48th Annual Southeast Regional Conference, Oxford, USA (2010)