Database Performance Comparisons: An Inspection of Fairness

Uwe Hohenstein and Martin Jergler
Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, Munich, Germany

Keywords: Performance, Comparison, Benchmark, Neo4j, PostgreSQL.

Abstract: Special benchmarks and performance comparisons have been published to analyze and stress the outstanding performance of new database technologies. Quite often, the comparisons show that newly upcoming database technologies provide higher performance than traditional relational ones.
In this paper, we show that these performance comparisons are not always meaningful and should not encourage one to jump to fast conclusions. We revisit certain statements about comparisons between the Neo4j graph database and relational systems and indicate a couple of possible reasons for coming up with bad performance such as inappropriate or default configurations, and too straightforward implementations. Moreover, we refute some stated issues about the bad performance of relational systems by using a PostgreSQL database for commonly used test scenarios. We conclude with some considerations of fairness.

1 INTRODUCTION

Despite the dominating market share of relational database management systems (DBMSs), new database technologies permanently come up. Object-oriented DBMSs proclaimed in the 90s a revolution, promising to substitute traditional relational DBMSs (RDBMSs). A little time later, XML databases arose focussing on storing XML documents efficiently.

Since 2009, the NoSQL movement is gaining a lot of attention. NoSQL stands for "not only SQL" (http://www.nosql-database.org) although the name has been chosen provokingly. Many products bring up new ideas taking benefit from distribution, i.e., using a large amount of commodity computers to scale out, or storing complex graph structures in a specialized graph database.

Every new technology and its promoters claim their products to be superior to traditional RDBMSs. There are a lot of, sometimes emotional, discussions on which technology is better, NoSQL or SQL. (Moran, 2010) talks about techno-religious debates.

Those discussions are indeed very shallow from a technical perspective. To make our discussion more specific, we focus on the NoSQL category of graph databases. Looking for existing comparisons, we detected enthusiastic statements stressing the advantages over RDBMSs accompanied by performance measurements:

- "Graph databases outperform RDBMS on connected data" (Khan, 2016)
- "The main benefit of native graph databases are performance and scalability" (Khan, Ahmed, and Shahzad, 2017)
- "So the graph database was 1000 times faster for this particular use case" (Adell, 2013)
- "While MySQL did not finish within 2 hours, Neo4j finished a traversal in less than 15 minutes" (Rodriguez, 2011).

Even if graph databases possess advantages which are useful for specific applications, such general statements must be treated carefully.

This led to our motivation for investigating those statements in this paper. Thereby, our goal is *not* to state that RDBMSs are still better. Rather we want to stress on fairness – better unfairness – of such statements and comparisons and prove our point of view by measurements. This discussion of fairness tackles several aspects such as:

- Is it equitable to take rather ad-hoc configurations and settings?
- Is a warm start-up fair (especially if data sets are larger than available main memory)?
- Is it fair to compare a query language (SQL) with programming (e.g., Neo4j Pipes)?
- Are synthetic scenarios reasonable?

In particular, we show that:

- seemingly similar scenarios behave differently;
- database configurations and tuning are important for performance and comparisons;

Hohenstein, U. and Jergler, M.
Database Performance Comparisons: An Inspection of Fairness.
DOI: 10.5220/0007926602430250
In *Proceedings of the 8th International Conference on Data Science, Technology and Applications (DATA 2019)*, pages 243-250
ISBN: 978-989-758-377-3

- other data structures than the obvious or traditional ones are advantageous;
- programming instead of using a query language can improve performance.

The remainder of this paper is structured as follows: In Section 2, we collect some related work to underline the novelty of our investigation.

Afterwards, we elaborate in Section 3 upon unfairness of existing comparisons and various influencing factors before we conduct performance measurements on PostgreSQL for common Neo4j test scenarios in Section 4. In Section 5, we deduct criteria for achieving a fair performance comparison.

Section 6 concludes the investigation and presents some future work.

2 RELATED WORK

An early paper of (Hohenstein, et al., 1997) criticizes standard benchmarks for object-oriented database management systems (ODBMSs) like OO7 (Carey, et al., 1994) and statements made therein like "ODBMSs are faster by factor 100". In a case study, a real application using Oracle was transformed to several ODBMSs. The surprising result of the performance measurements then conducted 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. Consequently, the authors state that the best benchmark is the application itself. The paper presents a methodology for deriving application-specific benchmarks.

To our knowledge, no further work on fairness of performance comparisons has been published so far. Indeed, there are only some critical statements from (Baach, 2015) "*Comparing Neo4j to MySQL without the use of Cypher is comparing apples and oranges*". In the forum (Hacker, 2010) others also argue that some comparisons are meaningless.

However, several performance investigations can be found in the literature, which we tried to qualify.

(Khan, 2016) states that the technology of graph databases is better than RDBMSs by explaining why joins are bad for graph structures. He uses a simple scenario that consists of Employees (E), Payments (P) and Departments (D), related by one-to-many relationships E-P and P-D. Then, qualifying two departments by a query, the related payments are retrieved via the employees. The complexity is evaluated in Big-O notation. While RDBMS achieve O(|E|*|P|) with nested loop joins and O(|E|+|P|) with hash joins, Neo4j has an O(k) behavior. Neo4j's constant behaviour is explained as follows: "*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).*" However, it remains unclear what "visiting all employees" in Neo4j means and how the internal data structures contribute to a better performance compared to hash indexes in RDBMSs.

(Rodriguez, 2011) uses 1,000,000 nodes and 4,000,000 edges with a synthetic distribution: Despite an average fan-out of 4, some nodes have a higher number of edges. A test measures traversal from a starting node to related nodes via 1 to 5 hops. The result reveals that Neo4j is more than twice as fast for 4 hops. For 5 hops, Neo4j required 14.37 minutes while MySQL was stopped after 2 hours.

The test of (Adell, 2013) detects if one person is connected to another in 4 or fewer hops. The data set contains 1,000,000 users with an average of 50 friends. Neo4j required 2ms for the check, while an RDBMS was stopped after running several days.

Another comparison (Baach, 2015) uses 100,000 and 1,000,000 nodes with exactly 50 edges each. A test *counts* the number of friends up to 5 hops. As a surprising result, MySQL was about 6 times faster than Neo4j. One potential reason for that might be the use of the Cypher query language to perform queries while (Rodriguez, 2011) sticks to the Pipes framework, which seems to be very beneficial. (Baach, 2015) considers a comparison SQL vs. the Cypher language as fair, whereas SQL vs. Pipes being unfair. Another reason for the result might be some deeper thoughts about configuring MySQL.

(Vicknair et al, 2010) experiment with data sets of size 1,000, 5,000, 10,000 and 100,000 nodes. In contrast to others, they set up a direct acyclic graph. Several tests traverse the graph, and count the nodes, with 4 and 128 hops, count the number of nodes with a certain payload, particularly with "<" comparisons, and find all the orphan nodes. In general, the execution times are less than 200 ms, and do not show huge differences between Neo4j and MySQL. In fact, the data sets are small and enable in-memory processing.

(Khan, et al., 2017) compare 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 sec.

(Joishi and Sureka, 2015) use some process-mining algorithms for their comparison of MySQL and Neo4j: finding similarity between actors based on the intersection of activities and analysing causal dependencies between actors in carrying out a business process. MySQL is 32 times faster than Neo4j for similarity, while Neo4j attains a performance boost of a magnitude of 7x over MySQL for the second test.

(Martinez et al., 2016) also compare the performance of MySQL and Neo4j. Using three randomly generated data sets (1,000, 10,000 and 100,000 entries), 12 multi-join queries of a health application are tested. MySQL performs better than Neo4j in most cases but has a poor performance for larger data sets. It is important to note that no indexes were added in both database systems.

3 FAIRNESS OF COMPARISONS

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

3.1 Scope of Comparison

As already mentioned, the literature contains many exciting statements about the Neo4j performance. For example, (Khan, 2016) proves that graph databases are better than RDBMSs by a theoretical comparison of internal algorithms based on Big Os without explaining in detail why the internal Neo4j structures are better. A comparison of *technologies* at that level is not valid anyway.

Similarly, a comparison between a product X and RDBMSs in general as in (Adell, 2013) is wrong per se: Showing that Neo4j is faster than MySQL does not prove that Neo4j is faster than any relational DBMS. There are other products, too.

3.2 Small Test Data Sets

Performance tests are often performed with small data sets, e.g., 1000, 5000, 10,000 (Vicknair et al, 2010). Even graphs with 100,000 nodes (Baach, 2015) are not really large. This means that a test is basically testing in-memory capabilities: All the data fits into the accordingly sized memory.

These evaluations are only representative for applications for which the memory is available w.r.t. the amount of data. Results cannot be generalized for larger data sets since they do not cover disk accesses, which will then certainly be required.

3.3 Warm Start

The first execution of a query is slow because data is fetched from disk and the query execution plan has to be derived. Further executions, also with different values, are faster because the execution plan is already available and data is in the cache. That is why performance comparisons like (Baach, 2015) 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 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.

3.4 Using Standard Configurations

DBMSs possess many parameters, which are important for performance. However, in performance comparisons, standard configurations are used more or less. For example, (Martinez et al., 2016) state that "The deployed database servers were not optimized" and "No index was added to the basic implementation".

The cache size is one important parameter, which is partially considered. Other parameters, e.g., the space for temporal data, affect sorting and eliminating duplicates. Indeed, tuning database configurations can speed up accesses drastically.

3.5 Over-tuning

If a benchmark stimulates only a few parts of an application, tuning the benchmark can lead to a highly optimal test program for exactly that portion. There is a danger of over-tuning a specific scenario or query (especially with a warm start). However, such a specific tuning might have a negative and invisible impact on other – potentially non-tested – scenarios such as inserts or deletes.

3.6 Synthetic Test Scenarios

Most of the published benchmarks and comparisons are synthetic in the sense that they abstract from concrete applications. They aim at being generic hand and reducing the effort for implementing and performing tests. (Barry, 1994) states that it is easy to spend $100,000 for implementing a benchmark, especially if tests have to be implemented on severval systems. Certainly, standard and simplified

benchmarks help to reduce the implementation effort. However, it is questionable whether those tests are representative for a particular application. Results of comparisons are only representative if tests coincide with the application in mind. Thus, the tested operations must reflect the characteristic accesses of a given application.

Most comparisons fail in this respect. For example, benchmarks for graph databases use a configurable number of nodes and relationships (e.g., Vicknair et al, 2010), thereupon performing typically traversals along connections between nodes as *the* use case. The tests are thus rather synthetic. Thus, it is not possible to adapt a benchmark to the demands of a specific application beyond configuring some few parameters. Tests are performed by changing a few factors such as the number of nodes (Vicknair et al, 2010) (Baach, 2015) or the fan-out of relationships. Such a parameterization does not help to let a benchmark become more representative. It is legitimate to question whether simple and slightly configurable tests could be representative for an application at all.

There are mostly no tests for mixed scenarios with queries, inserts, updates, and deletes combined in one test. Hence, just a few isolated features are compared. Real life applications surely perform other accesses.

Furthermore, there are different understandings of what a traversal is. Sometimes, a traversal retrieves all related nodes via up to n hops, sometimes it only counts the connected nodes. Other tests determine all possible connections between two nodes, or simply detect whether two nodes are related via up to n hops. Beside the fact that such tests are scenarios that are advantageous for graph databases, these similar scenarios show huge differences in performance as we will see later.

3.7 Implementation Issues

Some comparisons compare tests written in pure SQL with the procedural Neo4j Pipes framework instead of the Cypher query language. Moreover, the test of (Baach, 2015), comparing SQL with the Neo4j Cypher query language, comes along with a winner MySQL. Obviously, the Cypher language does not perform as well as the Pipes framework. We doubt that comparing SQL with the Pipes framework is fair.

Another point is about using a straightforward database schema. There are other options partially requiring stored procedures, which should be tried out in a comparison.

3.8 Data Distribution

Performance typically depends not only on the test scenario and test data such as the number of nodes and the number of edges, but also on the distribution of data for individual nodes. For example, the selected starting node is relevant, since 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!

3.9 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.

4 PERFORMANCE TESTS

All these issues often lead to results proving that graph databases are 100 times faster than relational systems, (Adell, 2013). Such statements, worded quite general, must at least be seen relative to the test scenarios and their relevance.

In order to support our statements, we performed some experiments with a PostgreSQL database. We intentionally used an older version 9.5 because several comparisons of RDBMS vs. Neo4j are also older. Hence, there is no advantage for the RDBMS by using the most recent state of technology.

We take three “traversal” scenarios from published comparisons: Scenario **ALL(n)** starts with a random node and determines all nodes reachable by less than n hops. Another scenario **PATHS(n)** determines only the paths between two given nodes related by k hops, while **EXISTS(n)** checks whether two given nodes are related by k hops, k<=n.

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

Test1	Cold Start		Warm Start			
	First [sec]		First [sec]		Second [sec]	
Index	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 tests ran in an isolated environment without any parallel database accesses or other running applications. Each test was performed 3 times. The average of measurements was taken. All the tests use the same laptop running Windows 7 with a dual-core processor, 12 GB of RAM, and a 465 GB SSD disk. Hence, the machine is not oversized.

It is important to note that we do not perform a direct comparison with Neo4j – because benchmarks are unfair. Instead, our goal is to put some published statements into perspective.

4.1 Test Results for 500,000 Nodes

We experimented with two differently sized databases. The first database contains 500,000 nodes with 50 edges to other randomly selected nodes.

Test 1: Impact of Indexes for Scenario PATHS.
Our test series starts with a PostgreSQL standard configuration (especially a very small cache size of 128 MB). We apply the frequently used database schema of (Adell, 2013) consisting of one table Friends (id int, friend int). Each node is identified by a unique id; friend is a foreign key that refers back to a friend's node, i.e., all those records that refer to the same id form the collection of friends for that node.

The first test has the purpose to illustrate the impact of indexes. A corresponding SQL query for Scenario PATHS(5) is sketched out in Figure 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 = :x and f5.friend = :y
union
select f1.id, f2.id, f3.id, f4.id, f4.friend, null  …
union
select f1.id, f2.id, f3.id, f3.friend, null, null  ...
union
select f1.id, f2.id, f2.friend, null, null, null  ...
union
select f1.id, f1.friend, null, null, null, null
from Friends f1 where f1.id = :x and  f1.friend = :y;
```

Figure 1: SQL statement for Scenario PATHS(5).

The query computes the complete paths including all the intermediate nodes; :x and :y represent the start and end nodes, resp. The query is executed after a restart of the computer (cold start), and afterwards simply immediately executed additional three times, now with a "warm" cache. In a second step, different start and end points are taken for the same query. Results are summarized in Table 1.

The test pinpoints a huge difference: Indexes are essential for achieving performance – this is not a surprise. 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. Section 3.4) – can heavily falsify benchmark results.

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

Test 2: Cold/Warm Start.
Table 1 also contains the PATHS(5) results for a comparison between warm and cold start. The intention of this test is to investigate how the system behaves in case of loading data from disk. This is relevant since we cannot assume all the data in memory for larger applications.

The difference between cold and warm start is minimal for the test without indexes because table data has to be loaded anyway. Using indexes, there is a large difference between cold and warm start. Thus, a test should not only be restricted to warm start tests (cf. Section 3.3).

Test 3: Implementation Variants.
The next test illustrates the impact of query tuning. Scenario EXISTS checks the existence of connections between two nodes. In contrast to PATHS, we are satisfied with an answer, connected or not. There are at least three possible queries:

a) simply perform the query for PATHS (cf. Figure 1) and check for a non-empty result;
b) add a LIMIT 1 at the end of the PATHS query to obtain just a single first connection;
c) add a LIMIT 1 for each sub-query in order to stop the execution early after the first hit:
   (select f5.friend … limit 1) union … union
   (select f1.friend … limit 1)

Table 2 shows the enormous speed-up for Variant c). Consequently, searching for alternative implementations or queries can be very effective – even if as simple as here! Taking one straightforward solution is not reasonable (cf. Section 3.7). Note that no connections for 1 to 4 hops exist for our test. Hence, the multi-join queries are executed.

Test 4: Data Distribution.
Next, we show that test data and parameters have an impact on performance (cf. 3.8). We use PATHS(5) with different start and end nodes, resulting in different numbers of connections. Table 2 shows how execution times depend on the chosen start and end nodes, and. The smaller the number of retrieved connections is, the faster the query performs.

Table 2: Results for Test 3, 4, and 5.

		Cold [sec]		Warm [sec]	
Test	**Variant**	**Ø**	**stdev**	**Ø**	**stdev**
Test 3	a)	23.78	5.27	4.54	1.55
	b)	19.81	1.61	4.79	1.07
	c)	2.39	0.13	0.057	0.02
Test 4	479 recs	15.18	1.27	3.32	0.64
	797 recs	23.78	5.27	4.54	1.55
Test 5	128MB	23.78	5.27	4.54	1.55
	1024MB	25.58	0.92	0.46	0.03

Similarly, the order of sub-queries is important for Scenario EXISTS(5) in Test 3. If there are no hop-1 and hop-2 but hop-3 connections, the query is fastest if the hop-3 sub-query occurs first and immediately stops execution. The possibility of choosing the right nodes has an influence on results. Knowing the data set, the implementation can be "improved". This is also a form of over-tuning (cf. Section 3.5) by consciously "tuning" the order of sub-queries according to data.

Test 5: Larger Cache Size.
The default cache in PostgreSQL with 128MB is far too small for our table data of 864 MB and index data of 1607 MB. Only 5% of the overall data fits into the cache. Consequently, many reads happen from disk. To improve the cache hit ratio, we increase memory by factor 8 to 1024 MB in order to have more data in memory, but still not sufficient to keep all the data.

The results for PATHS(5) with a larger cache are also presented in Table 2. The difference between a small (default) and large cache for a cold start is ignorable; the data must be fetched from disk anyway. However for a warm start, we recognize more than factor 8 of speed up. That is, sticking to default configurations falsifies results (cf. 3.4).

Please note there are many further tuning parameters, e.g., the use of temporal space to speed up sorting and duplicate elimination.

Test 6: Different Implementations and Structures.
So far, we have used straightforward table structures and SQL for "implementing" the scenarios. However, there are alternatives for data structures and/or implementing the computation logic, which are often not considered (cf. Section 3.7). For example, instead of a table Friends(id, friend), we can use an array-valued column in a table FriendsWithArray(id int, friends int[]). Each node is represented by just a single record independent of the number of friends.

Turning to Scenario ALL, the query for getting the nodes for 4 hops starting with :x looks like:

```
select distinct f1.id as fid, f1.friends  into  tmp3
from FriendsWithArray f1 where f1.id = :x;

insert into  tmp3  select distinct f2.id, f2.friends
from FriendsWithArray f1, FriendsWithArray f2,
      generate_subscripts(f1.friends,1) i1,
where f1.id = :x 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 = :x and f1.friends[i1] = f2.id
and f2.friends[i2] = f3.id and f3.friends[i3] = f4.id
```

Figure 2: Query for ALL 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 variable is used to access a the i-th field in the array by means of friends[i] to be used in joins between array elements (i.e., sons) and Ids.

Test ALL(4) "Old" is computed with a single SQL query similar to Figure 1, however, returning related nodes for a start node :x instead of paths. The "New" variant proceeds stepwise using a stored procedure following the query structure of Figure 2. The result is stored in a temporary table tmp4, which is then used in another query to unnest the node Ids:

```
select distinct t3.id, friends[i] as friend into  tmp4
from tmp3 t3, generate_subscripts(t3.friends,1) i
```

For ALL(5) "New", two additional steps are added:

```
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;
```

Table 3 shows that the computation of related nodes over 5 hops is possible in about half a minute – as opposed to several hours as stated in (Rodriguez, 2011) (Adell, 2013). As expected, the results with the larger cache size are even better.

Table 3: Results for Scenario ALL.

Variant [sec]	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

The improved performance is paid by some drawbacks. The table violates the first normal form but is still easy (maybe even easier) to understand. Also, inserts and deletes become more complicated. A stored procedure might help to handle the logic.

Test 7: Differences in Traversal Scenarios.
Comparing the previous results, we recognize the different performance behaviour of the various "traversal" scenarios ALL/PATHS/EXISTS: In a warm start, PATHS(5) is fast with about 5 seconds, and EXISTS(5) is even very fast (a few milli-seconds) with an optimized query. However, finding nodes in ALL(5) is slower with half a minute. This illustrates how important the chosen scenario and its semantics is, even if scenarios might look quite similar (cf. Section 3.6).

Test 8: JDBC Configuration.
So far, we have executed tests interactively with the PostgreSQL console. However, database access will be typically invoked by means of JDBC, ADO.NET, or an object/relational mapping tool. Hence, another factor enters the game having impact on performance.

We consider fetching the query result in Java with JDBC. One important option 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, already serving this and the next n-1 successive requests, too.

We used ALL(4) and executed the "Old" query with different fetch sizes. The query executed in 30 seconds with a size of 1 (typically being the default) and 18 seconds with a size of 1000. This a huge difference, especially since the query execution itself consumes about 15 sec. Again, relying on defaults affects the performance negatively (cf. Section 3.4).

4.2 Second Database

In order to elaborate more on another facet of 3.8, we use a larger database with 5,000,000 nodes each having four randomly chosen friends. The results for the bad performing One-SQL-statement of ALL(n) (cf. Figure 1) are shown in Table 4. Even n=9 and n=10 achieve moderate execution times despite the higher number of hops. Thus, compared to the previous database, the fan out seems to be one decisive factor for performance results (cf. 3.8).

Table 4: Results for ALL scenario (large cache).

	Cold [sec]		Warm [sec]		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 THOUGHTS ABOUT FAIRNESS

Having discussed some performance scenarios and thereby achieving different results than often presented in the literature, we turn to the question what fairness of performance comparisons means.

At first, it is an absolute precondition for fairness to supply the same environment with the same resources such as hardware, processor, operating system, network, disks, RAM, degree of test isolation and the same overall test conditions etc.

But if the size of memory for the DBMS is pre-fixed, it is starting to become unfair. DBMSs like Neo4j are Java-based and execute queries at the client, making a lot of memory in the JVM more advantageous. RDBMSs process queries in the server instead. Consequently, a smaller JVM might be sufficient in order to leave more RAM for the DBMS. Hence, resources cannot be set equally for all the test candidates since the settings not only depend on an application and its data, but also on the type of DBMS.

The rules for performing benchmarks also have a strong impact on the expressiveness of results. Conditions must not be too restrictive: Every test should implement the *same universe of discourse* with the same functionality. But a tester should not be forced to use a specific database schema, a query language etc. For example, benchmarks for ODBMSs, e.g., (Carey, DeWitt, and Naughton, 1994), 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 good guess, and tuning is neglected. The OO7 benchmark (Carey, DeWitt, and Naughton, 1994) legitimates such a proceeding by stating that normal programmers cannot tune a system effectively. This

statement is doubtful in our opinion. Most DBMSs *do* require an appropriate tuning in order to optimize performance. As we have demonstrated, simple tuning measures like creating indexes already improve performance considerably; but there are much more screws to turn. This potential must be used for sustainable results.

Performance comparisons are certainly fair if infinite time is at the tester's disposal. This is quite infeasible, however, the following attenuation makes sense: Each tester of a DBMS should obtain the same, sufficient, amount of time. Furthermore, each implementer of a test must have the same degree of skills and knowledge of a particular database candidate. Otherwise, different skills should be considered for restricting the time. Anyway, the time limit must not be too tight, there should be sufficient time to avoid too straightforward, often naive, solutions. Having enough time allows a programmer to try out concepts in several variants and to tune the overall system.

Finally, it is important to have realistic and holistic test scenarios. This particularly means that scenarios should cover representative and complete use cases with a mixed set of operations. This also reduces the risk of over-tuning and evaluating or aggregating partial results in a convenient manner.

6 CONCLUSIONS

Publications about new database technologies often claim to be superior to traditional technologies such as relational database products. This is proven by means of technical performance comparisons.

We picked up some published statements that compare the graph database Neo4j with relational systems coming up with some huge performance gains for Neo4j. The overall goal of this paper was to discuss those "sweeping" statements. We revealed some common test scenarios with a PostgreSQL database and illustrated a huge spectrum of performance depending on factors such as configuration, database schemas, tuning etc. Moreover, we achieved good results for those critical scenarios that were proven to be bad for RDBMSs. One important conclusion was that a good and comparable performance can often be achieved if doing it in the right way. The message should be that each system can be tuned for particular use cases: A deeper investigation becomes absolutely indispensable for reliable results.

Hence, we want to encourage people to perform own benchmarks if tools have to be compared instead of blindly believing in published comparisons. We made an attempt to give some recommendations to achieve fair comparisons.

Our future work will enlarge the scope to other NoSQL categories. That said, we want to focus on covering further potential advantages of NoSQL products such as distribution, scalability etc.

REFERENCES

Adell, J., 2013. Performance of Graph vs. Relational Databases. https://dzone.com/articles/performance-graph-vs [last access 2019/5/1].

Baach, J., 2015. Neo4j Performance Compared to MySQL. https://baach.de/Members/jhb/neo4j-per-formance-compared-to-mysql [last access 2019/5/1].

Barry, D., 1994. Should you take the plunge? *Object Magazine 3(6), 1994*.

Carey, M., DeWitt, D. and Naughton, J., 1994. The OO7 Benchmark. *ACM SIGMOD 1994*.

Hacker, 2010. NoSQL vs. RDBMS: Let the flames begin. https://news.ycombinator.com/item?id=1221598 [last access 2019/5/1].

Hohenstein, U., Pleßer, V. and Heller, R., 1997. Evaluating the Performance of Object-Oriented Database Systems by Means of a Concrete Application. *8th DEXA Workshop, Toulouse 1997*.

Joishi, J. and Sureka, A., 2015. Graph or Relational Databases: A Speed Comparison for Process Mining Algorithm. *Proc. Of 19th International Database Engineering & Applications Symposium, Yokohama, 2015*.

Khan, W., Ahmed, W. and Shahzad, E., 2017. Predictive Performance Comparison Analysis of Relational & NoSQL Graph Databases. *Int. Journal of Advanced Computer Science and Applications 8(5), January 2017*.

Khan, Q., 2016. Why Graph Databases Outperform RDBMS on Connected Data. https://dzone.com/articles/why-are-native-graph-databases-more-efficient-than [last access 2019/5/1].

Martinez, A., Mora, R., Alvarado, D. et al., 2016. A Comparison between a Relational Database and a Graph Database in the context of a Personalized Cancer Treatment Application. *Proc. of Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City 2016*.

Moran, B., 2010. RDBMS vs. NoSQL: And the Winner is .... http://www.itprotoday.com/microsoft-sql-server/rdbms-vs-nosql-and-winner, 2010 [last access 2019/5/1].

Rodriguez, M., 2011. MySQL vs. Neo4j on a Large-Scale Graph Traversal. https://dzone.com/articles/mysql-vs-neo4j-large-scale [last access 2019/5/1].

Vicknair, C., Macias, M., Nan. X., et al., 2010. A Comparison Between a Graph and a Relational Database: A Data Provenance View. *Proc. of 48th Annual Southeast Regional Conference, 2010, Oxford, (USA).*