Benchmarking Oracle Property Graph against Neo4j

Oracle

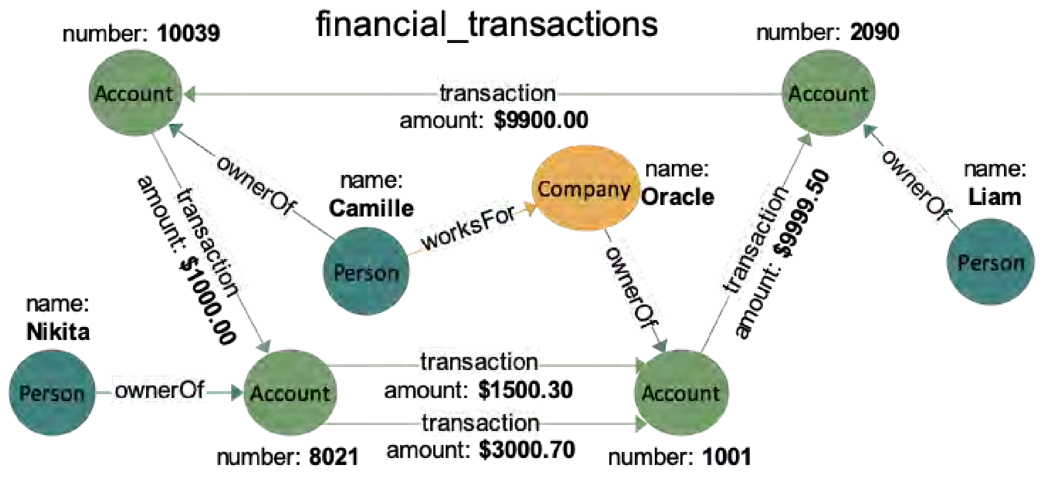
June 30, 2020

Abstract

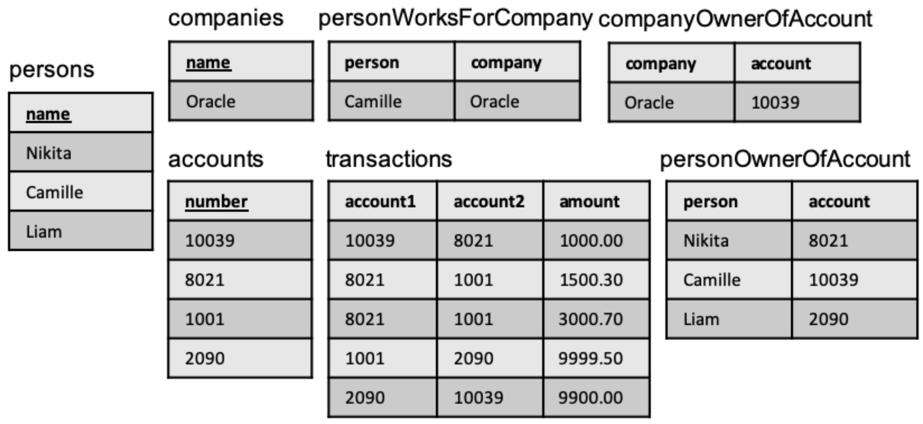
Recently, an in-depth benchmarking paper was published by researchers of UC Merced [19] comparing performance of Neo4j via the LDBC Social Network Benchmark [7, 9]. Here, we use the same benchmark and focus on its complex ”Business Intelligence” workload to compare performance of Oracle Property Graph against those numbers from UC Merced. We show that Oracle outperforms Neo4j on most queries in this workload.

1 Introduction

The Property Graph is a new data model that provides an intuitive abstraction to represent data and especially the connectivity of data. Connectivity is modeled through edges (or relationships), each edge connecting two vertices (or nodes) with one another. Whereas XML, JSON and RDF graphs all have standardized textual syntaxes to represent data for easy sharing on the World Wide Web, the Property Graph is merely an abstract model and instances are easiest shared through visualizations like the one in Figure 1.



(a) An example property graph with 8 vertices and 10 edges. Here, "Person", "Account" and "Company" are vertex labels, "ownerOf", "transaction" and "worksFor" are edge labels, "name" and "number" are vertex properties and "amount" is an edge property.



persons

name
Nikita
Camille
Liam

companies

name
Oracle

personWorksForCompany

person	company
Camille	Oracle

companyOwnerOfAccount

company	account
Oracle	10039

accounts

number
10039
8021
1001
2090

transactions

account1	account2	amount
10039	8021	1000.00
8021	1001	1500.30
8021	1001	3000.70
1001	2090	9999.50
2090	10039	9900.00

personOwnerOfAccount

person	account
Nikita	8021
Camille	10039
Liam	2090

(b) The same example represented in the Relational model.

Figure 1: An example property graph (a) and a corresponding representation in the Relational model (b).)

Although concepts behind property graphs were already developed before Dijkstra's Shortest Path algorithm was invented in 1956, property graphs
only became popular as a practical approach to data management over the last decade or so, in part thanks to community efforts like Apache TinkerPop [1] with its graph traversal language Gremlin [2] and Titan [6] as well as commercial graph database vendors like Neo4j [10].

Since then, industry adoption spurred, and database vendors have adopted graph technology as part of a broader data management approach that allows users to see their data in different ways, for example as both tables and graphs. Examples of this are Oracle Property Graph [11], SAP Hana Graph [12] and graphs in Microsoft SQL Server [4].

To compare the graph offerings by these vendors in a meaningful way, standardized benchmarks are needed. This allows users to identify which products best suit their needs. Furthermore, standard benchmarks also stimulate competition and product growth. To these ends, the Linked Data Benchmark Council (LDBC) [7] – a joint effort between academia and industry – developed two benchmarks for graphs: the LDBC Social Network Benchmark (SNB) [9] and the LDBC Graphalytics benchmark [8]. Together, SNB and Graphalytics target "a broad range of systems with different nature and characteristics' [9]'. To date, the LDBC SNB is the most advanced and complete benchmark for graphs and is therefore used in this paper to compare Oracle Property Graph against Neo4j.

Outline The remainder of this paper is organized as follows. Section 2 introduces LDBC's Social Network Benchmark (SNB). Section 3 compares the graph query languages used by different products. Section 4 presents the benchmarking results, including comparison of loading time and query performance. Finally, Section 5 gives the conclusions.

2 LDBC's Social Network Benchmark

Benchmark Workloads. The LDBC Social Network Benchmark (SNB) [9] consists of two workloads:

- The *Interactive* workload consists of user-centric transactional-like queries. It has three classes of queries: (1) complex read-only queries, (2) short

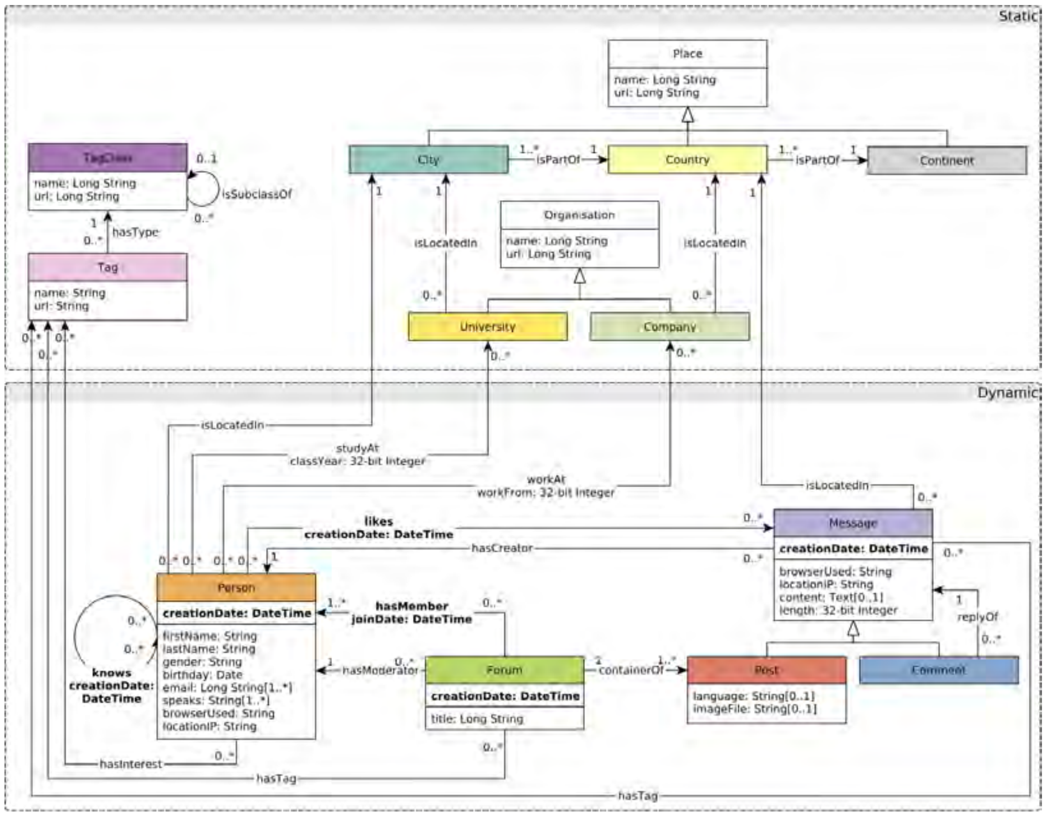


Figure 2: The LDBC SNB data schema.

read-only queries and (3) transactional update queries inserting new entities.

- The *Business Intelligence (BI)* workload includes analytic queries to respond to business-critical questions. A research paper was published on it at the GRADES-NDA workshop at SIGMOD 2018 [20] and presents results for Oracle for 10 of the 25 queries. This paper is the first paper to show results for Oracle for all 25 queries.

Data Schema. Figure 2 shows the data schema of SNB in UML. Data represents a snapshot of the activity of a social network during a period of time and includes data on entities such as persons, organizations and places. The schema also models the way persons interact, by means of the friendship relations, the sharing of content such as messages, replies to messages and likes to messages. People form groups to talk about specific topics, which are represented as tags.

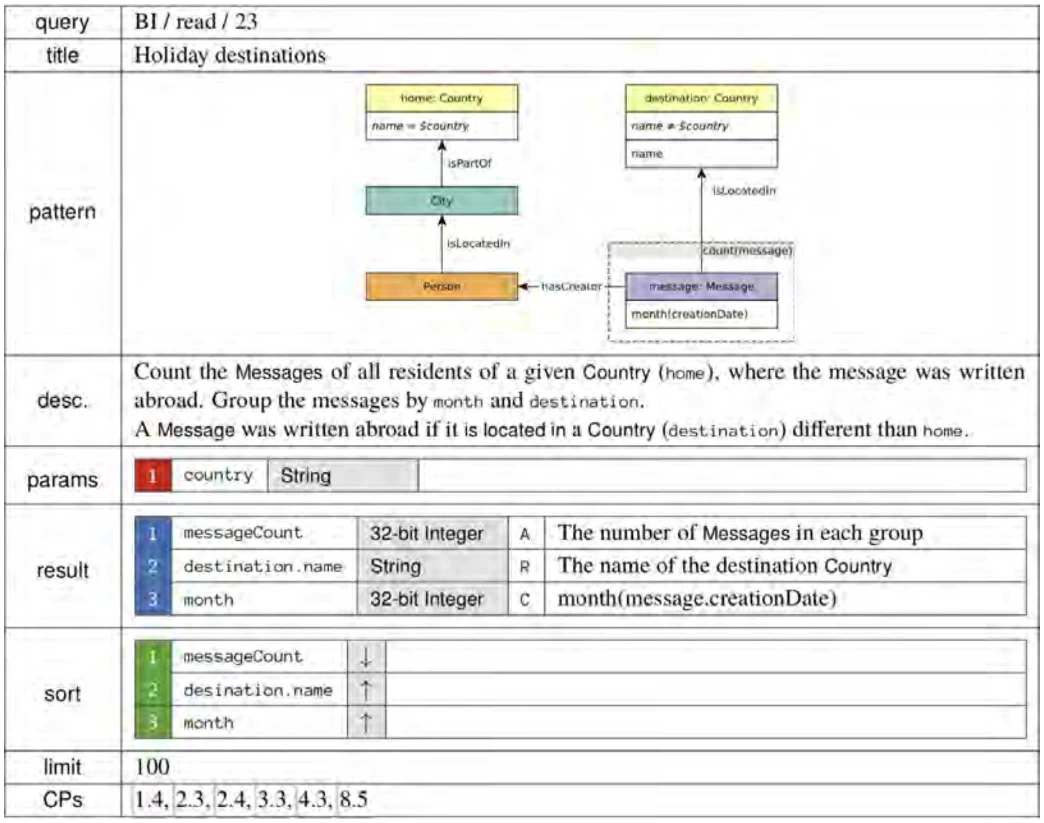
Data Generator. The benchmark comes with a data generator that allows you to generate data sets of user-specified scale factors. Data generated mimics the characteristics of those found in real social networks such as Facebook. Output attributes, cardinalities, correlations and distributions have been finely tuned to reproduce social networks and real data from DBpedia [3] is included to ensure attribute values are realistic and correlated [20].

3 Property Graph Query Languages

In this section we introduce the query languages that were used to express the queries of the LDBC's Business Intelligence (BI) workload.

PGQL and Cypher. Most vendors currently use their own proprietary property graph query languages: Neo4j has Cypher [13], and Oracle has PGQL [16]. These languages are expressive enough to express all 25 queries of the BI workload, the more complex workload of the LDBC SNB benchmark. Cypher and PGQL are more suitable for expressing complex queries when compared to the open-sourced graph traversal language Gremlin that is used by products such as Amazon Neptune; to date, we are not aware of anyone who has attempted to express the BI queries in Gremlin.

Example query. Figure 3 shows one of the LDBC's BI queries (query 23) from LDBC's SNB Specification [9]. Note that this query requires four edge traversals ("isPartOf", "isLocatedIn", "hasCreator" and another "isLocatedIn"), as well as grouping, aggregation and top-n fetching. Listing 1 expresses the query in Oracle's PGQL and Listing 2 expresses the query in Neo4j's Cypher.



query	BI / read / 23
title	Holiday destinations
pattern	
desc.	Count the Messages of all residents of a given Country (`home`), where the message was written abroad. Group the messages by `month` and `destination`. A Message was written abroad if it is located in a Country (`destination`) different than `home`.
params	1 | `country` | String
result	1 | `messageCount` | 32-bit Integer | A | The number of Messages in each group 2 | `destination.name` | String | R | The name of the destination Country 3 | `month` | 32-bit Integer | C | month(message.creationDate)
sort	1 | `messageCount` | ↓ 2 | `desination.name` | ↑ 3 | `month` | ↑
limit	100
CPs	1.4, 2.3, 2.4, 3.3, 4.3, 8.5

Figure 3: Query 23 of the Business Intelligence Workload of the LDBC Social Network Benchmark. See [9] for the full specification of the benchmark.

```
SELECT   COUNT(message) AS messageCount, destination.name, month
FROM     MATCH (person:Person) <-[:hasCreator]-
                                        (message:Post|Comment),
         MATCH (message) -[:isLocatedIn]-> (destination:Place),
         MATCH (person) -[:isLocatedIn]-> (city:Place),
         MATCH (city) -[:isPartOf]-> (homeCountry:Place)
WHERE    homeCountry.type = 'country' AND
         destination.type = 'country' AND
         city.type = 'city' AND
         homeCountry.name = 'Ethiopia' AND
         homeCountry  <> destination
GROUP BY EXTRACT(MONTH FROM message.creationDate) AS month,
         destination.name
ORDER BY messageCount DESC, destination.name, month
LIMIT    100
```

Listing 1: LDBC's BI Query 23 expressed in Oracle's PGQL [17].

```
MATCH
  (home:Country {name:'Ethiopia'})<-[:IS_PART_OF]-(:City)<-[:
    IS_LOCATED_IN]-
  (:Person)<-[:HAS_CREATOR]-(message:Message)-[:IS_LOCATED_IN
    ]->(destination:Country)
 WHERE home <> destination
WITH
  message,
  destination,
  message.creationDate/100000000000%100 AS month
RETURN
  count(message) AS messageCount,
  destination.name,
  month
ORDER BY
  messageCount DESC,
  destination.name ASC,
  month ASC
LIMIT 100
```

Listing 2: LDBC's BI Query 23 expressed in Neo4j's Cypher [13].

Standardization of Property Graph Querying. Standardization of property graph querying is on-going with Oracle, IBM, SAP, Neo4j, TigerGraph, ArangoDB Inc, and Redis Labs participating in the ANSI INCITS DM32 Ad Hoc on SQL extensions for property graphs. SQL/PGQ [5] is the property graph extension of SQL that is planned to come out in the next version of the SQL standard. There is also an effort going on to create a standalone property graph query language named GQL.

The plan for Oracle's PGQL is to align it to SQL/PGQ and to SQL in general, to provide a unified experience to users of the Oracle database and to not require them to learn an entire new language if they want to get started with property graphs. A big part of this effort has already been completed as can be seen in Listing 1, which shows that PGQL closely follows SQL's "SELECT .. FROM .. GROUP BY .. ORDER BY ..." syntax.

Listing 3 shows what the same query looks like according to the latest working draft of SQL/PGQ. The main difference between SQL and PGQL is that PGQL provides a syntactic shortcut that allows the MATCH clause of the SQL/PGQ query to be pulled up into the outer query.

```
SELECT   COUNT(*) AS messageCount, GT.name,
         EXTRACT(MONTH FROM GT.creationDate) AS month
FROM     GRAPH_TABLE (
           ldbcGraph
           MATCH
             (person:Person) <-[:hasCreator]-
                                       (message:Post|Comment),
             (message) -[:isLocatedIn]-> (destination:Place),
             (person) -[:isLocatedIn]-> (city:Place),
             (city) -[:isPartOf]-> (homeCountry:Place)
           WHERE homeCountry.type = 'country' AND
                 destination.type = 'country' AND
                 city.type = 'city' AND
                 homeCountry.name = 'Ethiopia' AND
                 homeCountry  <> destination
           COLUMNS ( message.creationDate, destination.name )
         ) AS GT
GROUP BY EXTRACT(MONTH FROM GT.creationDate),
         GT.name
ORDER BY messageCount DESC, GT.name, month
FETCH    FIRST 100 ROWS ONLY
```

Listing 3: LDBC's BI Query 23 expressed in ISO/IEC's SQL/PGQ [5] following the latest Working Draft.

Graph Algorithms. Although this paper focuses on graph *queries* and their performance, graph *algorithms* are just as essential for many graph use cases. The Graphalytics benchmark of LDBC specifically focuses on the performance of graph algorithms such as Breadth-First Search (BFS), PageRank (PR), Weakly Connected Components (WCC), Community Detection using Label-Propagation (CDLP), Local Clustering Coefficient (LCC) and Single-Source Shortest Paths (SSSP). Although some of these algorithms – particularly BFS and SSSP – can be expressed as a graph query in SQL/PGQ, PGQL and Cypher, the other algorithms have a more procedural nature and require constructs like for-loops, while-loops, etc.

Oracle Property Graph and Neo4j provide many of the popular graph algorithms as a built-in feature so that the algorithms can easily be invoked on arbitrary graphs through a single API call. In addition, Oracle Property Graph also provides an algorithm API called PGX Algorithm [17] that allows you to write your own graph algorithms from scratch and compile them into highly optimized parallel in-memory execution.

4 Benchmark Experiments

First, we provide details about the benchmark setup in Section 4.1, then we show results for loading performance in Section 4.2 followed by results for query performance in Section 4.3.

4.1 Setup

Chosen hardware and scale factor. Rather than re-doing the Neo4j experiments, we reuse the numbers from the UC Merced paper [19] and compare them against the numbers we obtain for the same query workload evaluated against Oracle Property Graph. We used a machine with the same hardware capabilities and we also matched the scale factor of the graph data.

The chosen OCI machine and AWS machine used by UC Merced is shown in Table 1. Details on the scale factor 100 graph are shown in Table 2.

Table 1: Comparison of hardware used for experiments with Oracle and Neo4j.

	Oracle	**Neo4j**
Machine	Oracle OCI - VM.Standard2.16[1]	Amazon AWS - r4.8xlarge[2]
CPUs	16 cores (32 virtual cores)	16 cores (32 virtual cores)
	2.0 GHz Intel® Xeon® Platinum 8167M	2.3 GHz Intel Xeon E5-2686 v4
Memory	240 GB	244 GB
Cost*	735 USD per month	1532 USD per month
	(excl. DB license cost)	(excl. DB license cost)

*As of January 15, 2021

Table 2: Details on the LDBC SNB graph used in the experiments.

Scale factor	100
Number of vertices	0.3 billion
Number of edges	1.8 billion
Total size of CSV files	87 gigabyte

[1] https://www.oracle.com/cloud/compute/virtual-machines.html

[2] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/memory-optimized-instances.html

Setup of Oracle Property Graph. Oracle Property Graph allows PGQL queries to be executed directly against the Oracle database through translation to SQL [14]. There are also ways to speed up query execution further through features like Oracle Database In-Memory [15] and the in-memory graph server (PGX) [18]. The in-memory graph server (PGX) is a solution that creates in-memory graphs from tables that reside in the Oracle database. Alternatively, if data persistence is not a requirement, CSV files can be loaded directly into PGX without first having to store the data as tables. Graphs in PGX are stored in memory in a compressed form that is optimized for graph traversal. To date, PGX provides the most performant way to run graph queries and graph algorithms in Oracle Property Graph and has therefore been used in this performance evaluation.

To set up the loading experiment of Section 4.2, we did not use any special setting when creating the graph from tables. However, before loading CSV into PGX, we did split up each CSV file into multiple files in order to benefit from parallel loading. We paired the OCI machine with the AWS machine based on memory and number of CPUs (see above) and did not compare the disk speed of the two machines.

Query evaluation methodology. First of all, we cross-validated the correctness of queries via published query output [4] of Neo4j for the scale factor 1 graph. Second, we use the same query parameters that were used for Neo4j to execute PGQL queries against PGX. Third, we take the median of 10 query executions. Note that [19] gnored the execution time of the first query and then took the median of the next 9 query executions, but since the first query is always the slowest, the median of all 10 runs should be strictly larger than the median of the last 9 runs (next time, we will also take the median of the next 9 queries as that may give slightly better results). Finally, just like [19] we impose a timeout of 18,000 seconds (5 hours) such that when the timeout expires, the query is terminated.

4.2 Graph Loading Performance

Figure 4 shows the loading performance of Oracle Property Graph compared to Neo4j.

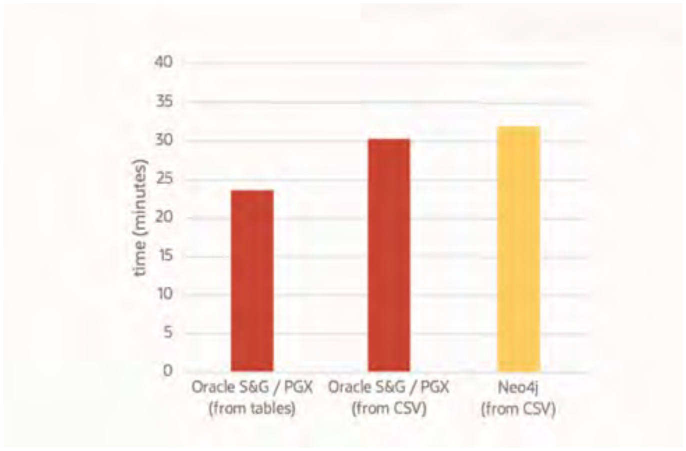


Figure 4: Loading performance of LDBC's scale factor 100 graph (0.3B vertices & 1.8B edges; 87 GB of raw CSV data). Note that PGX can load graphs either from tables in the Oracle database or from CSV files, hence the two bars for Oracle Property Graph.

[4] `https://github.com/tigergraph/ecosys/tree/ldbc/ldbc_benchmark/neo4j/result`

From the results it follows that Oracle Property Graph outperforms Neo4j, no matter if graphs are created from tables or from CSV files. An interesting observation is that Oracle Property Graph performs better when creating graphs from tables than when loading graphs from CSV files, even though graphs were created in parallel in both cases.

4.3 Graph Query Performance

Figure 5 plots the query performance of Oracle Property Graph compared to Neo4j and Table 3 shows the raw numbers and relative speedups. The results can be summarized as follows:

- **Oracle** outperforms Neo4j on 19 out of 25 queries (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q11, Q12, Q14, Q16, Q17, Q21, Q23, Q24, Q25).

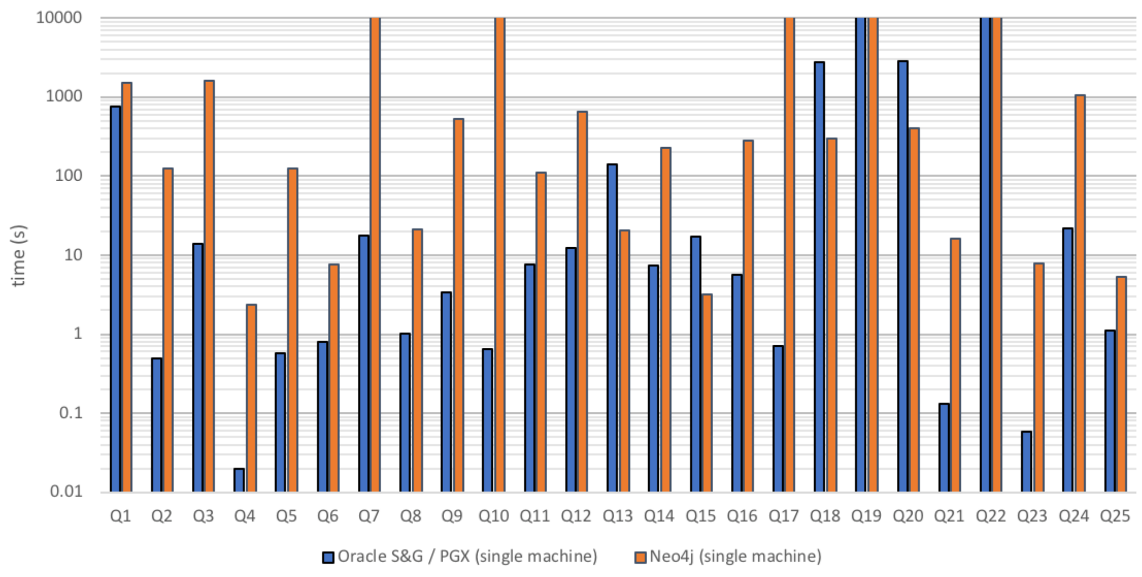


Figure 5: Results for LDBC's Social Network Benchmark (SNB) - Business Intelligence (BI) workload - scale factor 100 (0.3B vertices & 1.8B edges; 87 GB of raw CSV data)

- **Neo4j** outperforms Oracle on only 4 out of 25 queries (Q13, Q15, Q18, Q20).

A timeout was set at 5 hours such that queries that took longer were canceled:

- **Oracle** times out on two queries (Q19 and Q22).
- **Neo4j** times out on five queries (Q7, Q10, Q17, Q19, Q22).

Overall, Oracle handles the workload better than Neo4j. Both engines do store the vertices and edges in the graph in such a way that lists of incoming and/or outgoing neighbors of vertices can be accessed in constant time. This is important since edge traversal is the fundamental graph operation.

However, although constant-time traversal is supported by both engines, Oracle does significantly better than Neo4j and this is likely because the neighbor lists are stored in memory in a more compressed and optimal form that may benefit better from spatial locality when graph data in memory is cached by CPUs. This impacts all queries since all 25 queries require edge traversal.

Table 3: Raw performance numbers and speedup of Oracle over Neo4j.

	Oracle (s)	**Neo4j (s)**	**Neo4j/Oracle**
Q1	765.9110	1,508.6240	1.970
Q2	0.4955	125.7995	253.884
Q3	13.9055	1,601.4695	115.168
Q4	0.0200	2.3326	116.630
Q5	0.5705	125.4889	219.963
Q6	0.8020	7.5908	9.465
Q7	17.466	t/o	> 2,410.930
Q8	1.009	20.9287	20.742
Q9	3.422	523.3391	152.934
Q10	0.6385	t/o	> 28,191.073
Q11	7.6085	109.6632	14.413
Q12	12.4655	659.7865	52.929
Q13	141.4660	20.2515	0.143
Q14	7.3475	229.2400	31.200
Q15	17.0365	3.2056	0.188
Q16	5.6215	277.9866	49.451
Q17	0.7110	t/o	> 25,316.456
Q18	2,802.283	294.7010	0.105
Q19	t/o	t/o	?
Q20	2,835.683	399.9949	0.141
Q21	0.1305	16.2169	124.267
Q22	t/o	t/o	?
Q23	0.0575	7.8020	135.687
Q24	21.684	1,051.4582	48.490
Q25	1.112	5.2952	4.762

5 Conclusions

In this paper we first gave a brief introduction to Oracle Property Graph and compared its graph query language PGQL to Neo4j's graph query language Cypher by taking one of the LDBC benchmark queries as example. Oracle's PGQL is closely aligned to SQL and is able to express all 25 complex queries of LDBC's Business Intelligence workload, which is the more complex workload of the LDBC Social Network Benchmark (SNB). We showed that through the in-memory graph server (PGX), Oracle Property Graph is able to outperform Neo4j on graph loading as well as on the bulk of the queries in the workload.

References

[1] *Apache TinkerPop*, 2020 (accessed June 25, 2020). http://tinkerpop.apache.org/.

[2] *Apache TinkerPop's Gremlin*, 2020 (accessed June 25, 2020). https://tinkerpop.apache.org/gremlin.html.

[3] *DBpedia*, 2020 (accessed June 25, 2020). https://wiki.dbpedia.org/.

[4] *Graph processing with SQL Server and Azure SQL Database*, 2020 (accessed June 25, 2020). https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-ver15.

[5] *ISO/IEC WD 9075-16 Information technology — Database languages SQL — Part 16: SQL Property Graph Queries (SQL/PGQ)*, 2020 (accessed June 25, 2020). https://www.iso.org/standard/79473.html.

[6] *JanusGraph*, 2020 (accessed June 25, 2020). https://janusgraph.org/.

[7] *LDBC - The graph & RDF benchmark reference*, 2020 (accessed June 25, 2020). http://ldbcouncil.org/.

[8] *LDBC Graphalytics Benchmark v0.9.0 - Draft Release*, 2020 (accessed June 25, 2020). https://github.com/ldbc/ldbc_graphalytics_docs/raw/master/doc/LDBC-Graphalytics_tech-specs_v0.9.0.pdf.

[9] *The LDBC Social Network Benchmark (version 0.3.2)*, 2020 (accessed June 25, 2020). http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf.

[10] *Neo4j Graph Platform*, 2020 (accessed June 25, 2020). https://neo4j.com/.

[11] *Oracle Property Graph*, 2020 (accessed June 25, 2020). https://www.oracle.com/database/technologies/

spatialandgraph/property-graph-features.html.

[12] *SAP HANA Graph*, 2020 (accessed June 25, 2020). https://help.sap.com/viewer/f381aa9c4b99457fb3c6b53a2fd29c02/1.0.12/en-US/7734f2cfafdb4e8a9d49de5f6829dc32.html.

[13] Neo4j. *Cypher Query Language*, 2020 (accessed June 25, 2020). https://neo4j.com/developer/cypher-query-language/.

[14] Oracle. *Executing PGQL Queries Directly Against Oracle Database*, 2021 (accessed February 10, 2021). https://docs.oracle.com/en/database/oracle/property-graph/21.1/spgdg/property-graph-query-language-pgql.html#GUID-94F08780-EC3D-4F9B-985F-49984939E61C.

[15] Oracle. *Oracle Database In-Memory*, 2020 (accessed June 25, 2020). https://docs.oracle.com/en/database/oracle/oracle-database/20/inmem/intro-to-in-memory-column-store.html#GUID-BFA53515-7643-41E5-A296-654AB4A9F9E7.

[16] Oracle. *Property Graph Query Language*, 2020 (accessed June 25, 2020). https://pgql-lang.org/.

[17] Oracle. *Using Custom PGX Graph Algorithms*, 2021 (accessed February 10, 2021).
https://docs.oracle.com/en/database/oracle/property-graph/21.1/spgdg/using-inmemory-analyst-oracle-database.html#GUID-0CE6EC02-649E-403B-A61C-61BE7F4CB537.

[18] Oracle. *Using the In-Memory Analyst (PGX)*, 2021 (accessed February 10, 2021). https://docs.oracle.com/en/database/oracle/property-graph/21.1/spgdg/using-inmemory-analyst-oracle-database.html#GUID-C80502F2-67B0-42B3-B80F-6DA297EA655C.

[19] F. Rusu and Z. Huang. In-depth benchmarking of graph database systems with the linked data benchmark council (ldbc) social network benchmark (snb), 2019.

[20] G. Szárnyas, A. Prat-Pérez, A. Averbuch, J. Marton, M. Paradies, M. Kaufmann, O. Erling, P. Boncz, V. Haprian, and J. B. Antal. An early look at the ldbc social network benchmark's business intelligence workload. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, New York, NY, USA, 2018. Association for Computing Machinery.