Data Science and Engineering (2019) 4:309–322
https://doi.org/10.1007/s41019-019-00110-3

Which Category Is Better: Benchmarking Relational and Graph Database Management Systems

Yijian Cheng[1] · Pengjie Ding[1] · Tongtong Wang[1] · Wei Lu[1] · Xiaoyong Du[1]

Received: 16 August 2019 / Revised: 21 October 2019 / Accepted: 28 October 2019 / Published online: 11 November 2019

Abstract
Over decades, relational database management systems (RDBMSs) have been the first choice to manage data. Recently, due to the variety properties of big data, graph database management systems (GDBMSs) have emerged as an important complement to RDBMSs. As pointed out in the existing literature, both RDBMSs and GDBMSs are capable of managing graph data and relational data; however, the boundaries of them still remain unclear. For this reason, in this paper, we first extend a unified benchmark for RDBMSs and GDBMSs over the same datasets using the same query workload under the same metrics. We then conduct extensive experiments to evaluate them and make the following findings: (1) RDBMSs outperform GDMBSs by a substantial margin under the workloads which mainly consist of group by, sort, and aggregation operations, and their combinations; (2) GDMBSs show their superiority under the workloads that mainly consist of multi-table join, pattern match, path identification, and their combinations.

Keywords Relational database · Graph database · Benchmark

1 Introduction

E.F. Codd introduced the relational data model with relations to represent data, with relational algebra and relational calculus to operate data, and relational integrity constraint to control the consistency and completeness of data. Since then, various RDBMSs have been developed with standard SQL to support data definition, data manipulation, and data control operations. The relational data model achieves great success to support a wide spectrum of applications that are related to financial, personnel, manufacturing, and logistical data management. Even until now, RDBMSs still remain as mainstreaming data management systems.

In recent years, graphs have been shown increasingly important to big data applications such as social network analysis, spatiotemporal analysis and navigation, and consumer analytics, as it is able to capture complex relationships and data dependencies. For example, in social networks, users, pictures, and events are modeled as vertices, and relationships between them are modeled as edges. So far, RDBMSs have been shown to be capable of dealing with graph processing and analysis. RDF-3X [1], Hexastore [2], and SW-store [3] are commonly used RDF stores based on RDBMSs to manage semantic Web ontology and RDF knowledge bases. They transform SPARQL [4] queries over RDF [5] data to SQL using sort-merge joins in the relational world, and various relational optimization techniques are utilized to speed up the query processing.

As another alternative solution, Neo4j [6] was proposed as a graph database management system based on the graph model to manage graph data, and many other graph database management systems including ArangoDB [7], gStore [8, 9] are efficient in processing SPARQL queries over RDF datasets. Titan has been designed for graph data management and analysis. As a typical application example, in finance service industry, behaviors (like account statement, calling

✉ Wei Lu
lu-wei@ruc.edu.cn

Yijian Cheng
yijiancheng@ruc.edu.cn

Pengjie Ding
pengjie@ruc.edu.cn

Tongtong Wang
wttrucer@ruc.edu.cn

Xiaoyong Du
duyong@ruc.edu.cn

1 Renmin University of China, Beijing, China

list, loan history) of enterprises and customers are collected and modeled as graphs. Because GDBMSs such as Neo4j and ArangoDB often have rich graph algorithms, they are used to not only manage these behavior data, but also help do the analysis over these data using the algorithms, like personal or enterprise credit analysis.

Thus far, great controversies have been raised for the comparison between RDBMSs and GDBMSs: which category is better. On the one hand, from the perspective of GDBMSs, their advantage lies in schema-less property. They are able to manage structured, unstructured, or semi-structured data and thus are more flexible than RDBMSs. Given that GDBMSs are able to manage relational data, it is argued that RDBMSs may be replaced by GDBMSs. On the other hand, from the perspective of RDBMSs, RDBMSs have been proved to be capable of dealing with graph data management and analysis, and a few recent works have shown that by simply extending SQL languages in RDBMSs to support graph operations, the performance is comparable between RDBMSs and GDBMSs [10–14], verifying this capacity of RDBMSs. Furthermore, due to the lack of a unified graph model and query language across GDBMSs, which often incurs an extra programming and maintenance overhead for users, the necessity of GDBMSs is disputed.

To clearly answer this question, we propose a unified benchmark for both RDBMSs and GDBMSs. Because RDBMSs and GDBMSs have different data models and different query languages, in this benchmark, we target to evaluate RDBMSs and GDBMSs over the same datasets using the same query workload under the same metrics. First, to address the issue that RDBMSs and GDBMSs have different data models, we propose a relation-to-graph mapping scheme, under which relational data are able to be transformed to graph data. In this way, we use TPC-H [15], which is a commonly accepted benchmark in RDBMSs, and extend it to evaluate GDBMSs. Similarly, we propose a graph-to-relation mapping scheme, under which graph data are able to be transformed to relational data. We use LDBC [16], which is commonly used in GDBMSs, and extend it to evaluate RDBMSs. Second, to address the issue that RDBMSs and GDBMSs have different query languages, we transform all 22 SQL queries in TPC-H to graph queries and transform all 5 graph queries in LDBC into SQL queries. Third, consider that GDMBSs can adopt different back-end storage engines, which could potentially affect the performance of GDMBSs. We then evaluate the GDBSMs using different storage engines and report our findings. We select Postgresql, a popular open-source RDBMS, as the representative of RDBMSs and two popular GDBMSs Neo4j, ArangoDB as the representatives. We conduct extensive experimental evaluations to compare GDBMSs and RDBMSs over TPC-H and LDBC under the metrics, including query processing time, memory utilization ratio, and CPU utilization ratio.

In summary, our contributions are as follows:

- We extend a unified benchmark for both RDBMSs and GDBMSs to evaluate them under the same datasets as well as the same metrics.
- We propose a graph-to-relation inter-mapping scheme under which graph data and relational data are inter-transformed. We rewrite all SQL queries in TPC-H to graph queries and rewrite all graph queries in LDBC to SQL queries.
- We evaluate the performance of the GDBMSs using different back-end storage engines and report our findings.
- We conduct extensive experimental evaluations for existing popular RDBMSs and GDBMSs over both standard TPC-H and LDBC, and report our findings in detail.

The remainder of this paper is organized as follows: We review the related work in Sect. 2 and elaborate our unified benchmark in Sect. 3, following which we report experimental results and our findings in Sect. 4, before concluding the paper in Sect. 5.

2 Related Work

Our work is related to both RDBMSs benchmark and GDBMSs benchmark.

RDBMSs Benchmark As the mainstream commercial database systems, there exist rich RDBMSs benchmarks. Among them, the most well accepted are the TPC series. TPC-C [17] is an online transaction processing (OLTP) benchmark, which involves a mix of five concurrent transactions of different types and complexity. It models order management and extracts the workload. TPC-C is mainly used to test the capacity of transaction processing. TPC-H is a decision support benchmark which models business procurement, whose datasets consist of 8 tables representing general business procedure. The 22 queries and the data populating the database have been designed to evaluate the capacity of handling critical business questions. TPC-DS [18] is also a decision support benchmark which models several generally applicable aspects of a decision support system, including 24 tables, and 99 randomly replaceable SQL queries. It focuses on emerging technologies, such as big data systems, to execute the benchmark.

GDBMSs Benchmark Unlike RDBMSs benchmarks which are proposed by authoritative organization, GDBMSs benchmarks come from some database companies. NoSQL performance benchmark [19] is proposed by ArangoDB [7], with its prime target to compare the performance among MongoDB, PostgreSQL, OrientDB, ArangoDB, and Neo4j under the metrics including read/write performance test, memory utilization ratio. GDBMSs benchmark [20] is created by TigerGraph [21], mainly evaluating the data loading and query performance of TigerGraph, Neo4j, Amazon Neptune [22], JanusGraph [23], and ArangoDB. The LDBC graph analytics benchmark [16] is an industrial-grade benchmark for graph analysis platforms. It consists of several typical graph algorithms, standard datasets, data generators, and reference outputs, enabling the objective comparison of graph analysis platforms.

As pointed out in existing literature [10], both RDBMSs and GDBMSs are shown to be capable of managing relational and graph data. Nevertheless, the boundary of using RDBMSs and GDBMSs still remains unclear, i.e., for RDBMSs and GDBMSs, which category should be properly used under a given application scenario. To cope with this issue, in this paper, we then propose a unified benchmark for both RDBMSs and GDBMSs.

3 A Unified Benchmark for RDBMSs and GDBMSs

In this section, we present the unified benchmark that is applicable for both RDBMSs and GDBMSs. The main idea can be described as follows:

- We utilize the standard RDBMSs benchmark, TPC-H, and extend it to evaluate the performance for GDBMSs.
- Similarly, we utilize a widely used GDBMSs benchmark, LDBC, and extend it to evaluate the performance for RDBMSs.

By doing this, we can evaluate RDBMSs and GDBMSs on the same datasets with the same query workloads under the same metrics.

3.1 Data Generation

3.1.1 Data Generation Schemes

Since RDBMSs and GDBMSs have different data models, we need to develop an inter-transformation mechanism between relational data and graph data. To transform relational data to graph data, we propose a relation-to-graph mapping schema shown in Tables 1 and 2. In this schema, we generate one-to-one mapping between records of tables and vertices of the graph. In particular, records from the same table are mapped to the same class of vertices, i.e., vertices that correspond to the records of the same table are associated with the same label. Edges of the graph are generated according to the primary key and foreign key relationships. Specifically, the table containing the foreign key is called the child table, and the table containing the primary key is called the parent table. For a given record r from a child table, and its referenced record s (if any) from the parent table,

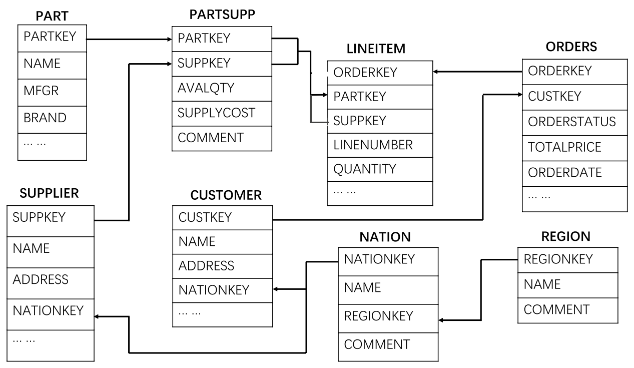


Fig. 1 The database schema for TPC-H benchmark

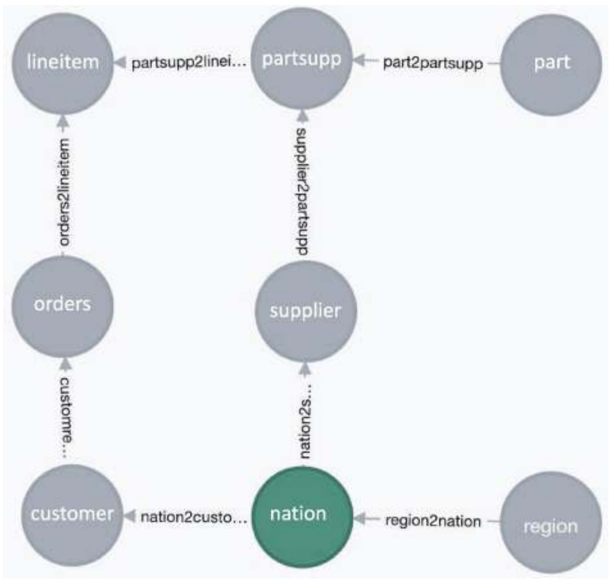


Fig. 2 The graph schema for TPC-H benchmark

Table 1 Table and vertex mapping

RDBMSs table	GDBMSs vertex
PART	Part
SUPPLIER	Supplier
PARTSUPP	Partsupp
LINEITEM	Lineitem
ORDERS	Orders
CUSTOMER	Customer
NATION	Nation
REGION	Region

Table 2 Table key and relationship mapping

From key	To key	Relationship
PART.PARTKEY	PARTSUPP.PARTKEY	Part2partsupp
SUPPLIER.SUPPKEY	PARTSUPP.SUPPKEY	Supplier2partsupp
PARTSUPP.PARTSUPPKEY	LINEITEM.PARTSUPPKEY	Partsupp2lineitem
ORDERS.ORDERKEY	LINEITEM.ORDERKEY	orders2lineitem
CUSTOMER.CUSTKEY	ORDERS.CUSTKEY	Customer2orders
NATION.NATIONKEY	SUPPLIER.NATIONKEY	Nation2supplier
NATION.NATIONKEY	CUSTOMER.NATIONKEY	Nation2customer
REGION.REGIONKEY	NATION.REGIONKEY	Region2nation

an edge is generated from r to s. By doing this, it is able to generate vertices and their edges based on the primary key and foreign key relationships. Note that in this paper, we use property graph model for GDBMSs to manage data. Thus, we generate one-to-one mapping between the attributes and the properties for each tuple from entity tables (or relationship tables) and its corresponding vertex (or edge), and set the property value to the attribute value accordingly. Then, we extract relational data from RDBMSs and transfer it to vertex files and edge files that follow the definition of the graph schema. By doing this, we can migrate relational data to GDBMSs via the built-in import tools in GDBMSs, such as *neo4j-import* in Neo4j and *arangoimp* in ArangoDB. Figure 1 shows the database schema for TPC-H benchmark, and Fig. 2 gives the transformed graph schema for the TPC-H benchmark in RDBMSs. We omit the details since Fig. 2 is self-explained.

Similarly, to transform the graph data to the relational data, we propose a graph-to-relation mapping scheme as well. In this scheme, we simply store the directed edges as triples, which are maintained in a relation with three attributes, namely *fromVertex, edgeLabel, toVertex*. As mentioned before, we utilize LDBC as the graph benchmark. This benchmark includes four datasets ranging from thousands of vertices and edges to millions of vertices and edges. The data sets cover four application domains, ranging from social network, citation network, Web graphs, to communication network. For ease of illustration, we label the dataset to small(S), medium(M), large(L), and extra (XL) according to its data size. Since there are no labels associated with the vertices, we do not create a separate vertex table for them.

3.1.2 Datasets to be Used

In TPC-H benchmark, the relational datasets, shown in Table 3, are generated using the TPC-H data generator with different sizes, ranging from 50 MB to 1 GB. Accordingly, we transform the relational datasets to the graph datasets based on the relation-to-graph mapping scheme, and the number of vertices and edges is shown in Table 3 as well, respectively. In LDBC benchmark, the graph datasets are shown in Table 4. Accordingly, we transform the graph datasets to relational datasets based on the graph-to-relation mapping scheme.

Table 3 TPC-H datasets

ID	Size	Vertices	Edges
tpch-0.05	50 MB	432,844	2,261,723
tpch-0.1	100 MB	866,602	4,530,029
tpch-0.5	500 MB	4,330,622	22,634,256
tpch-1	1 GB	8,661,245	45,268,530

Table 4 The real graph datasets

Graphs	Vertices	Edges	Size	Domain
Wiki-Vote	7115	103,689	S	Social
Cit-HepTh	27,770	352,807	M	Citation
Web-Stanford	281,903	2,312,497	L	Web graphs
Wiki-Talk	2,394,385	5,021,410	XL	Communication

3.2 Query Workload

The query workload is divided into three categories which are listed in Table 5. The first category is named as *atomic relational queries* consisting of four primitive operations, including *Projection*, *Aggregation*, *Join*, and *Order by*. We build this category of query workload to evaluate the performance of primitive relational operations implemented in GDBMSs. The second category is named as *TPC-H query workloads*. This category consists of 22 queries used in TPC-H. We target to evaluate the performance of GDMBSs under the case that the legacy RDBMSs are good at. The third category is named as *graph query workloads*, including 5 graph algorithms in LDBC Benchmark.

Table 5 Representatives of RDBMSs and GDBMSs to be compared

Category	Operations	# Of queries
Atomic relational queries	Project, Aggregation, Join, Order by	4
TPC-H workloads	All the TPC-H query workloads	22
Graph query workloads	BFS, CDLP, PR, LCC, WCC	5

We target to evaluate the performance of RDBMSs under the case that the GDBMSs are good at.

3.2.1 Atomic Relational Queries

Atomic relational queries emphasize on the evaluation of primitive relational operations, which are *Projection*, *Aggregation*, *Join*, and *Order by*. *Projection* queries are utilized for choosing which columns (or expressions) the queries should return. *Join* queries combine columns from one or more tables in a RDBMS. *Aggregation* queries are designed for grouping together the values of multiple rows. *Order by* queries are for sorting the rows of result set.

3.2.2 TPC-H Query Workloads

Typically, each GDBMS provides an SQL-like query language (e.g., *Cypher* for Neo4j and *AQL* for ArangoDB) to support data manipulation over graph data [24]. We transform all of the 22 queries of TPC-H into equivalent SQL-like graph query statements. Although among different GDBMSs, graph query languages could vary a lot, they basically belong to declarative languages, a user or a programmer merely specifies what is to be done rather than how to do in the query statements. For ease of illustration, we choose Neo4j as the representative of GDBMSs. For reference, as compared to the primitive relational operations in the RDBMSs, we list their counterparts in the GDBMSs and show them in Table 6. For the operations, *Projection*, *Aggregation*, and *Order by* in the RDBMSs, GDBMSs provide similar operations of the RDBMSs. However, GDBMSs implement the *Join* operation in quite a different way. Recall that a record in RDBMSs is mapped to one and single one vertex in the graph. Besides, for a given record *r* from a child table, and its referenced record *s* (if any) from the parent table, an edge in the graph is generated from *r* to *s*. For this reason, we transform the *Join* operation to a path query starting from *r* to *s*. To gain a better understanding, we further provide two examples of the transformations from primitive operations in RDBMSs to their counterpart in GDBMSs which are shown in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 AQL for TPC-H Query 1

```
FOR $line$ IN $lineitem$
FILTER
    DATE_ISO8601($line.L\_SHIPDATE$)<=DATE_ADD(
    DATE_ISO8601($'1998-12-01'$), $-90$, $"day"$)
COLLECT
    $RETURNFLAG = line.L\_RETURNFLAG$,
    $LINESTATUS = line.L\_LINESTATUS$,
    $sum\_qty$ =SUM(TO_NUMBER( $line.L\_QUANTITY$)),
    $and\ other\ aggregation\ operations$
SORT $RETURNFLAG, LINESTATUS$
RETURN
    $L\_RETURNFLAG : RETURNFLAG$,
    $L\_LINESTATUS : LINESTATUS$,
    $sum\_pty$,
    $and\ other\ elements\ retrieved\ by$ collect
```

Algorithm 2 Cypher for TPC-H Query 2

```
MATCH$(ps : Partsupp) - [\,] -> (s : Supplier) - [\,] -> (n : Nation) - [\,] -> (r : Region)$
WHERE
    $r.rName =' EUROPE'$
WITH $min(ps.psSupplycost)$ as $minvalue$
MATCH$(ps : Partsupp) - [\,] -> (p : Part), (ps : Partsupp) - [\,] -> (s : Supplier) - [\,] -> (n : Nation) - [\,] -> (r : Region)$
WHERE
    $p.pSize = 13$ and $p.pType = '.*SMALL.*'$ and $r.rName =' EUROPE'$ and $ps.psSupplycost = minvalue$
RETURN
    $s.sAcctbal$,
    $s.sName$,
    $and\ other\ elements$
ORDER BY
    $s.sAcctbal\ desc, n.nName, s.sName, p.pPartkey$
```

Table 6 RDBMSs' and GDBMSs' operations mapping

DB/Operations	Projection	Aggregation	Join	Order by
RDBMS	Select	Group by/sum/average ...	Join	Order by
GDBMS	Match	Group by/sum/average ...	Edges between vertexes	Order by

Algorithm 3 Bread-First Search in SQL

```
1: with RECURSIVE $BFS(toID, level, fromid, paths)$
2: as(
3: select $toID, 0, fromID, ARRAY[null, toID]$ from $R_{rel}$
4:                      where $toID = m$ and $fromID$ is $NULL$
5: union all
6: select $R_{rel}.toID, level + 1, BFS.toID, paths||R_{rel}.toID$
7:                      from $R_{rel}, BFS$
8:                      where $R_{rel}.fromID = BFS.toID$
9:                      and $level < n$
10: )
11: select $level, paths$ from $BFS$
```

3.2.3 Graph Query workload

We re-implement the five graph algorithms using SQL statements. For recursive algorithms, taking BFS (breadth-first search) [25] for example (shown in Algorithm 3), we use *with [recursive]* [26] clause to do the transformation by referring to *SQL'99* [27, 28]. Algorithm 3 shows the details on how to implement BFS in SQL. We first insert a record which *fromID* equals NULL and *toID* equals the start vertex's *ID* into relation R_{rel}. Then, we select the initial record into the temporary table *BFS*(*toID*, *level*, *fromid*, *paths*). Finally, iterative operations with the *with...unionall* clause continue to execute until *level* >= *n*, where *n* represents the depth of BFS.

For the following iterative algorithms, CDLP (community detection using label propagation) [29], PR (PageRank) [30], LCC (local clustering coefficient) [31], and WCC (weakly connected components) [32], none of them can be implemented in recursive SQL queries due to their *Group by* and *Aggregation* operations. So we use the *procedure* with *While* loop. Taking CDLP for example (shown in Algorithm 4), we first create a table *LP*(*ID*, *label*) to store *ID* and *label* of a vertex. Secondly, we figure out its neighbors of each vertex in *LP* and store the neighbors into R_{adj}. Thirdly, for each vertex *v*, we calculate the frequency of the labels that *v*'s neighbors have and set the label of *v* to the most frequent label that *v*'s neighbors have. In case that there might exist multiple labels to be the most frequent, we can simply choose one label among them with the minimum value, e.g., alphabetical order. Finally, loop operations with the *while* clause continue to execute until rec_times >= *times*, where *times* represents the loop times.

Algorithm 4 Community Detection Using Label Propagation in SQL

```
1: $LP(ID, label)$ as select $ID, ID$ from $R_{vertex}$
2: while $(rec\_times \leq times)$ do
3:     $R_{adj}(ID, label)$ as select $lp_1.ID, lp_2.label$ from $LP\ lp_1$
4:                      left join $R_{rel}\ rel$ on $lp_1.ID = R_{rel}.fromID$
5:                      left join $LP\ lp_2$ on $R_{rel}.toID = lp_2.ID$
6:     $R_{count}(ID, label, cnt)$ as select $ID, label, count(label)$ from $R_{adj}$
7:                      group by $ID, label$
8:     TRUNCATE $LP$;
9:     $LP(ID, label)$ as select distinct on$(vertexid) vertexid, label$ from $R_{count}$
10:                     group by $ID, label, cnt$
11:                     order by $ndoeid, cnt$ desc, $label$
12:    TRUNCATE $R_{adj}$
13:    TRUNCATE $R_{count}$
14:    $rec\_times := rec\_times + 1$
15: end while
```

3.3 Representatives of RDBMSs and GDBMSs to be Compared

We choose PostgreSQL, Oracle, and Microsoft SQL SERVER as the representatives of RDBMSs because they are widely used in the application domains. Note that our focus is to make a thorough study between RDBMSs and GDBMSs. Thus, we do not evaluate RDBMSs and GDBMSs separately. Namely, we choose PostgreSQL as the representative for RDBMSs in the relational operation evaluation. Similarly, we choose Neo4j and ArangoDB as the representatives of GDBMSs. For reference, versions of RDBMSs and GDBMSs are listed in Table 7.

Table 7 Testing object

DBMS	VERSION	CATEGORY
PostgreSQL	9.5	RDBMS
Oracle	11 g	RDBMS
MSSQL	2017	RDBMS
Neo4j	3.4.6	GDBMS
ArangoDB	3.3.19	GDBMS

3.4 Metrics

We measure the performance of RDMBSs and GDBMSs under the following metrics:

- Query processing time: the execution time of a graph query or an SQL query, which is returned and collected by RDBMSs and GDBMSs;
- Memory usage ratio: the peak usage ratio of memory during the execution of the whole workload;
- CPU usage ratio: the peak usage ratio of CPU during the execution of the whole workload;

We run each graph and SQL query for five times, and all the query processing time, memory usage ratio, and CPU usage ratio are computed on average.

4 Experiments

In this section, we first introduce the experimental setup. We then conduct extensive analysis of GDBMSs and RDBMSs over the same datasets, using the same query workload, under the same metrics. We finally summarize our findings.

4.1 Experimental Setup

Experimental Environment The experiments are conducted on a single vertex with a Intel(R) Xeon(R) Gold 6138 CPU @ 2.00 GHz processor, 256 G RAM, a 60T hard disk. We install Ubuntu 16.04 operating system, Java 1.8.0 with a 64-bit server VM. All the databases we examine are installed in this vertex. The hardware specifications of the machines are listed in Table 8.

Table 8 Hardware specifications

Component	Parameter
CPU	Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
Cores	40 (80 threads)
Memory	256 G
Disk	60 T
GPU	Titan V

Consider that the adoption of different storage engines could potentially affect the performance of GDBMSs. We first evaluate the effect of back-end storage engines of GDBMSs. We then compare GDBMSs and RDBMSs on the same datasets (i.e., TCP-H and LDBC), using the same query workload, and the same metrics. Details of the comparison setup are listed in the following.

- **Back-end storage engines of GDBMSs** The GDBMSs use a variety of back-end storage engines, which will bring different query performance for the same data model. Therefore, we first investigate the impact of different back-end storage engines on read and write performance under the same graph model.
- **Relational operations** We evaluate the performance of each database when processing general TPC-H queries and some extra evaluation queries. For TPC-H queries, we execute 22 queries on three databases: PostgreSQL as the representative for RDBMSs, Neo4j as the representative GDBMSs, and ArangoDB as a typical system for multi-model NoSQL database which includes graph data models. Twenty-two queries are executed on each of them, and meanwhile, the processing time, CPU usage, and main memory usage are recorded. Mean values are calculated for the processing results of the 22 queries to measure the general capacity of handling business-oriented ad hoc queries and concurrent data modifications for 3 databases. The evaluation for TPC-H queries only measures general capacity for business-oriented scenarios, yet every query in TPC-H consists of many atomic operations such as *Projection*, *Aggregation*, *Join* and so on. Four extra queries have been designed to measure the performance of processing 4 typical atomic operations: *Projection*, *Join*, *Aggregation*, and *Order by*.
- **Graph algorithms** For graph algorithms, we execute 5 graph algorithms we have introduced in Sect. 3 on four databases: Neo4j as the representative for GDBMSs; Oracle, PostgreSQL, and Microsoft SQL Server as 3 typical RDBMSs. Every graph algorithm is executed on 4 databases, and the processing time is recorded for each database.

4.2 Experimental Results and Analysis

In this section, we report the experimental results and analyze the results by highlighting our findings.

Table 9 Different storage engines

Back-end storage engine	Version	Category
RocksDB	5.8	Key-value storage engine
Hbase	2.2	Key-value storage engine
Cassandra	3.11	Key-value storage engine
Neo4j	3.4.6	Native graph storage engine
MySQL	5.7	Row storage engine

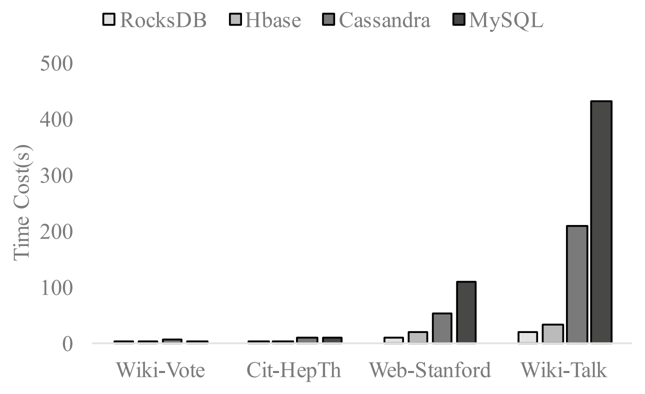


Fig. 3 The performance of data load

4.2.1 Back-end Storage Engines Evaluation

Note that the separation of compute and storage is architected in GDBMSs. For this reason, the GDBMSs can adopt different back-end storage engines. To make a clear explanation about the impact of the back-end storage engine on query performance, we conduct the following experiments to evaluate the performance of HugeGraph [33] with different back-end storage engines shown in Table 9.

Firstly, we test the performance of data loading. The results of four different back-end storage engines are shown in Fig. 3. It can be observed that key-value store engines have clear advantages over the row store engine, owing to its back-end structure based on LSM-tree, while row store engines like MySQL utilize B-tree [34] for persistent storage. For B-tree, small update operations may cause much random write operations. On the contrary, LSM-tree can deal with the key-value pairs in a batch to convert the random write operations to sequential write operations, which can magnificently improve the efficiencies of write operations.

Then, we compare the performance of three typical algorithms on different back-end storage engines. Figures 4, 5, and 6 illustrate the performances on shortest path algorithm, k-neighbor algorithm [35], and find-all-paths algorithm, respectively. We find that back-end storage engines with Hbase [36] and Cassandra [37] perform the worst in any algorithm test. MySQL performs similarly to RocksDB [38], and RocksDB performs even better in some tests. Just as discussed in the load performance above, key-value store engines based on the LSM-tree incur better performance in write operations, but it also costs much more for read operations, which is why back-end storage engine with MySQL performs better in these algorithms tests. Interestingly, we also find that RocksDB performs well and even better than MySQL. This benefit is from the read performance optimization strategy implemented by RocksDB. Specifically,

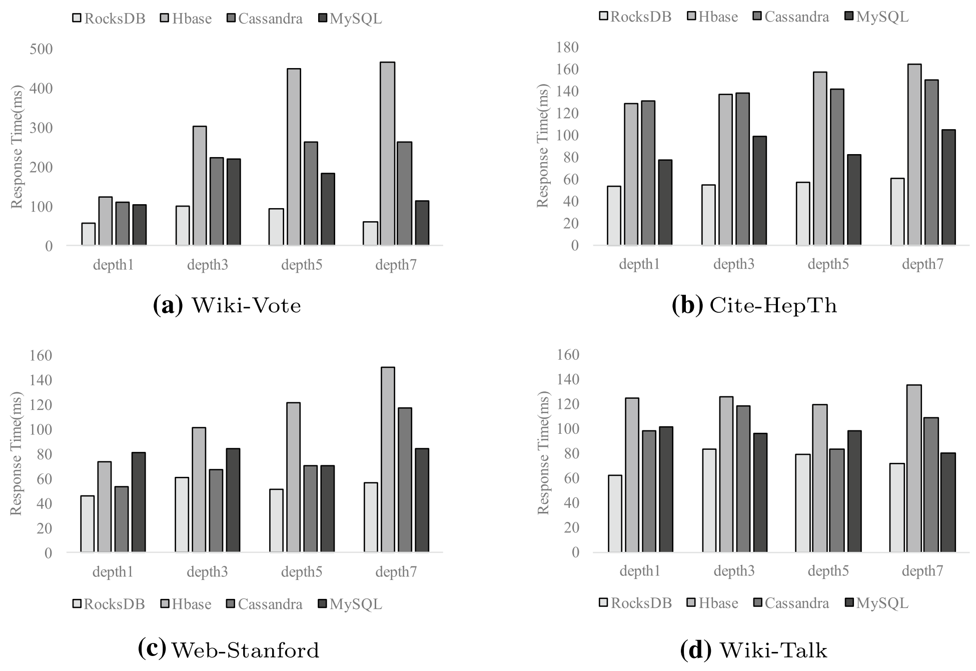


Fig. 4 Testing the shortest path algorithm over 4 back-end storage engines

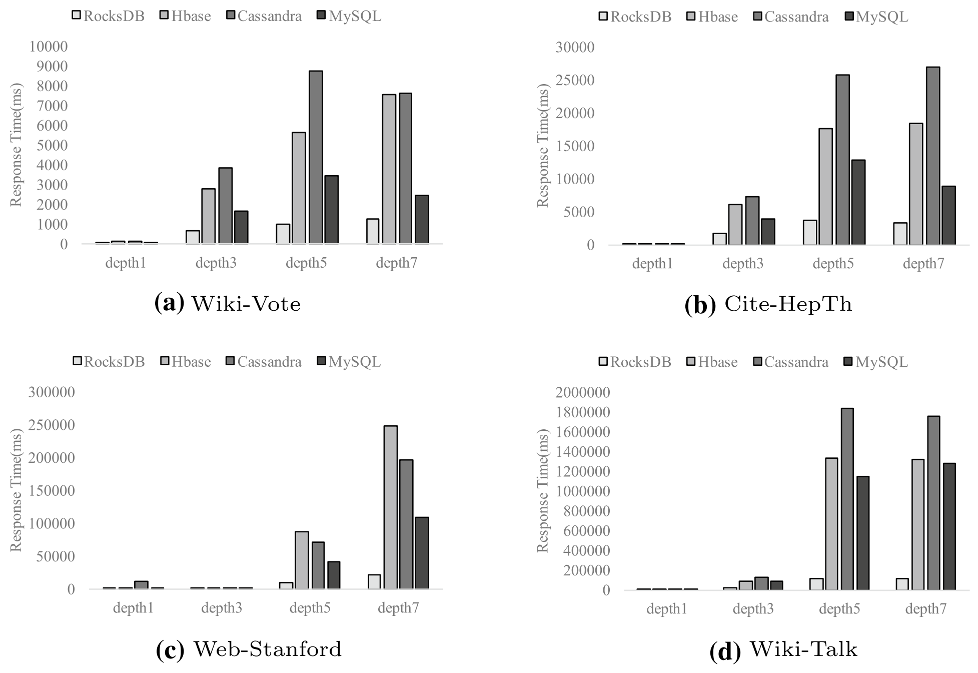


Fig. 5 Testing the K-neighbor algorithm over 4 back-end storage engines

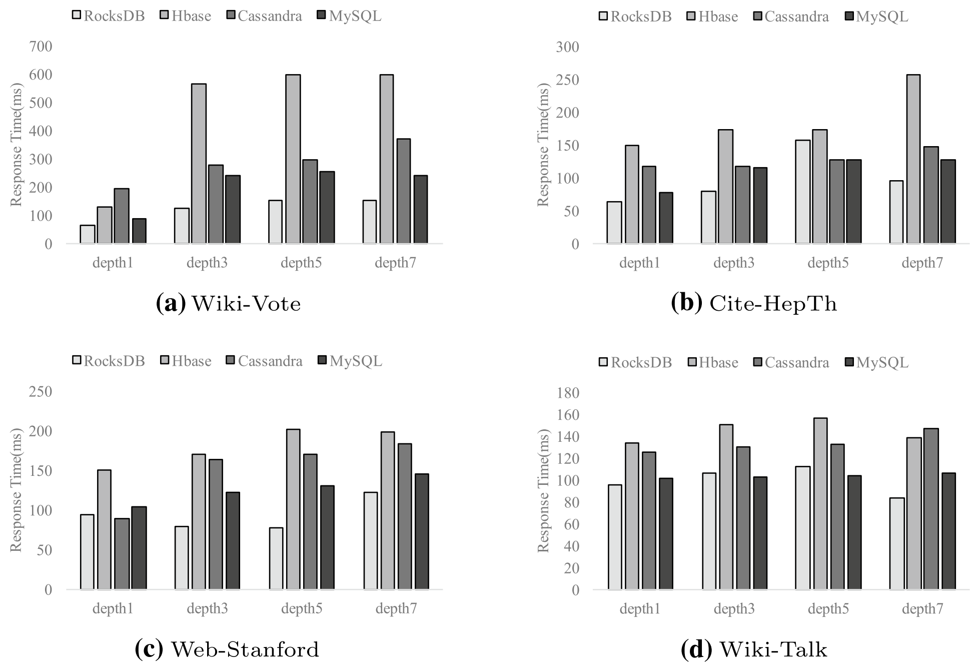
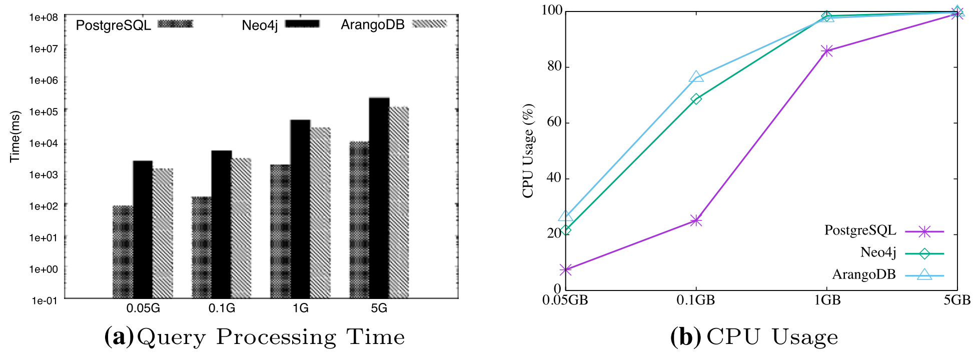


Fig. 6 Testing find-all-paths algorithm over 4 back-end storage engines

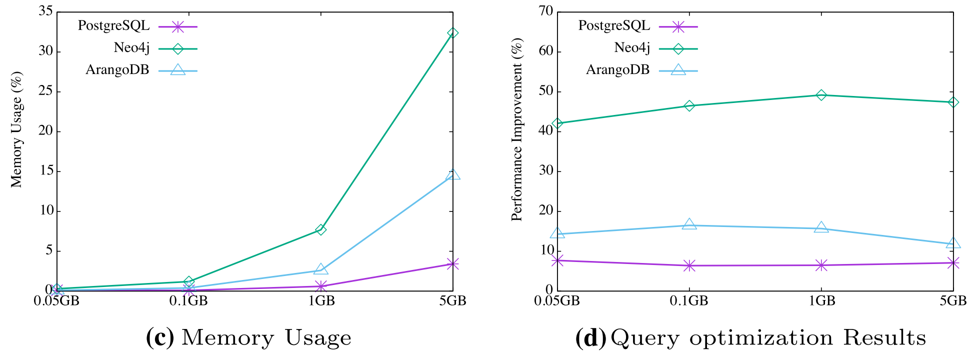
RocksDB implements two bloom filters [39]. One is utilized to filter blocks before reading blocks (the same as LevelDB [40]) without key. The other is dynamically generated to achieve key filtering in memory (before block read) when querying from memtables, improving the read performance tremendously. Combined with the official benchmark results [41] given by HugeGraph, Neo4j performs better with native graph storage than HugeGraph with Rocksdb.

From the above analysis, we can observe that the back-end storage engine plays an important role in query performance. The row-based storage engine used by RDBMSs do not perform well both for read and write operations. The key-value storage engines with the built-in LSM-tree index show their superiority in write operation, while the native storage of graphs outperforms the others in read operations.

Fig. 7 Relational operation test

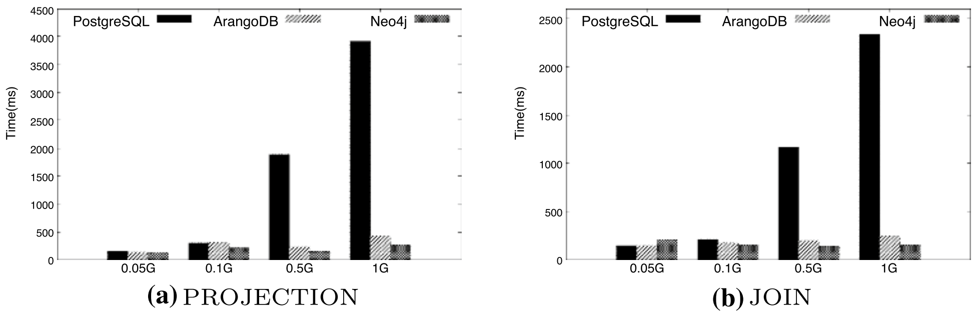


(a) Query Processing Time **(b)** CPU Usage

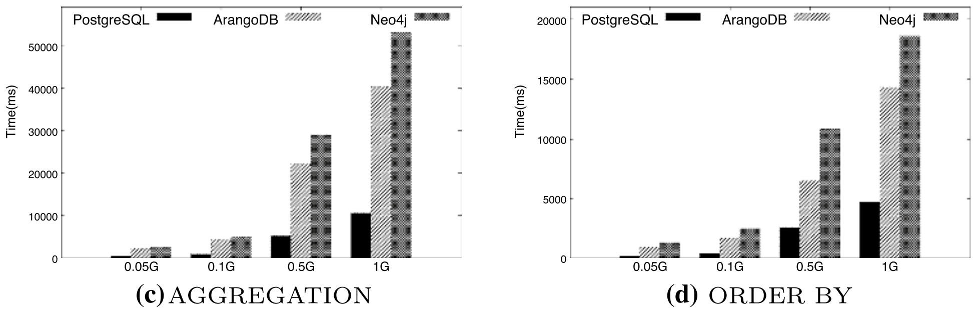


(c) Memory Usage **(d)** Query optimization Results

Fig. 8 Atomic operation test



(a) PROJECTION **(b)** JOIN



(c) AGGREGATION **(d)** ORDER BY

4.2.2 Relational Operation Evaluation

We carry out relational operation evaluation on three databases, namely PostgreSQL, Neo4j, and ArangoDB. Figure 7a illustrates the averaged processing time of the whole 22 TPC-H queries. From the figure, we can observe that the processing time in Neo4j and ArangoDB is significantly longer than that in PostgreSQL, although Neo4j is widely used by both academic researchers and industrial engineers. Figure 7b and c depicts the averaged CPU usage rate and the main memory usage rate, respectively. From the figures, we can observe that PostgreSQL also outperforms the other two databases in terms of CPU and main memory usages. Compared with Neo4j, ArangoDB consumes relatively less CPU but more main memory.

The above results indicate that compared with GDBMSs, RDBMSs have advantages in terms of query processing time and resource consumption. When the data size becomes larger, GDBMSs tend to be incompetent on the efficiency of query processing.

Besides, we also carry out an experiment to evaluate the processing time of 4 typical atomic operations (i.e., *Projection, Join, Aggregation, and Order by*) in RDBMSs. As Fig. 8 illustrates the results, we can observe that GDBMSs excel in *Projection* and *Join* operations but have disadvantages in *Aggregation*, and *Order by* operations. Note that with the increase in data scale, time consumption of PostgreSQL on *Projection* and *Join* raises sharply while Neo4j and ArangoDB increase smoothly. Meanwhile, for *Aggregation*, and *Order by* operations, there is much higher difference regarding the time consumption when data scale increases between RDBMSs and GDBMSs.

From the above experiments, we can find that both RDBMSs and GDBMSs have their own advantages on processing relational operations, but unexpectedly there is a high difference between them when handling TPC-H workloads. Therefore, we need to conduct an extra query optimization experiment to explore the reason.

The GDBMSs show their inefficiency when dealing with TPC-H datasets. Recall that GDBMSs are separately architected in their compute and storage modules. Moreover, in some GDBMSs, like Neo4J, properties of vertices and edges are physically linked in the storage system. This is good for traversing graph-structured data. However, this kind of storage structures requires plenty of time to query data when vertices have many properties without any index.

We then analyze the queries that consist of many complex operations over attributes (property of vertices in GDBMSs) such as the *Aggregation* operation and the *Order by* operation. Then, we create indices on these attributes/properties in 3 databases and measure the query processing time again. As shown in Fig. 7d, the improvement is apparent for GDBMSs, especially for Neo4j, whose improvement reaches over 40%, making its time consumption be close to PostgreSQL's time consumption. Meanwhile, we have also carried out evaluation on a graph dataset Wiki-Vote. The result indicates that the difference of time consumption for GDBMSs and RDBMSs tends to be rather slight, since there is no property for any vertex in this dataset.

From the experiments and analysis above, we can find that both RDBMSs and GDBMSs have their advantages of dealing with relational operations, whereas in business-oriented scenarios such as TPC-H workload, GDBMSs cannot have a satisfied performance due to their storage strategies. According to our extra experiments, the problem can be solved by creating indexes on properties properly and changing the schema for graph data transferred from RDBMSs, for schema of GDBMSs is flexible and adjustable. To be specific, by creating indexes on properties that constantly retrieved and avoid adding too many properties in vertices via extracting some properties and creating as vertex, GDBMSs can achieve much better performance for relational application scenarios than before, but still slightly worse than RDBMSs.

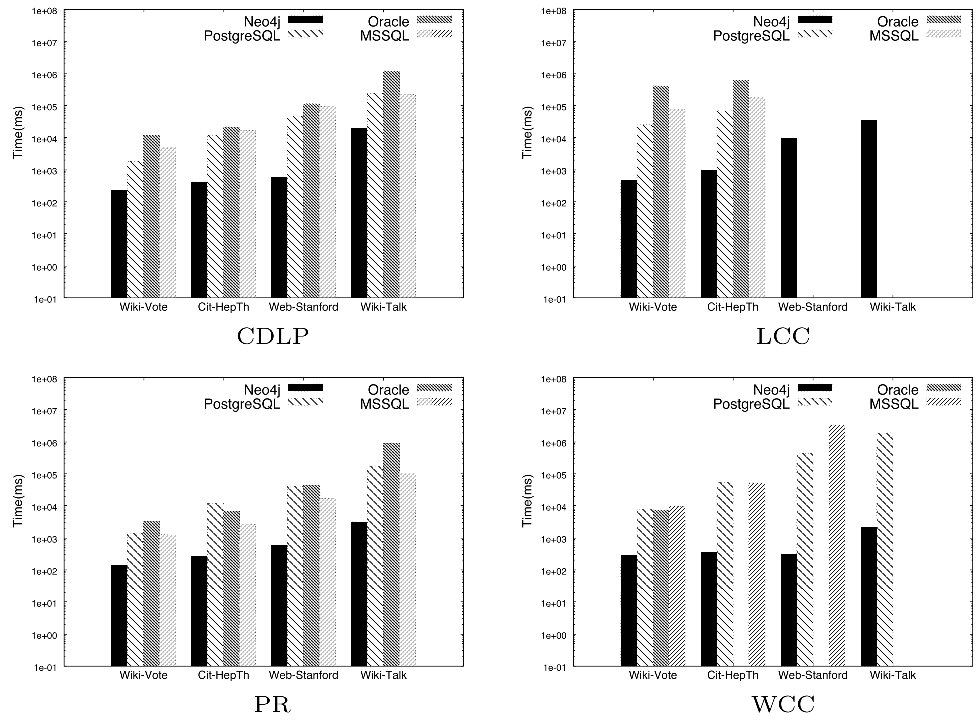


Fig. 9 Testing 4 graph algorithms over 4 datasets

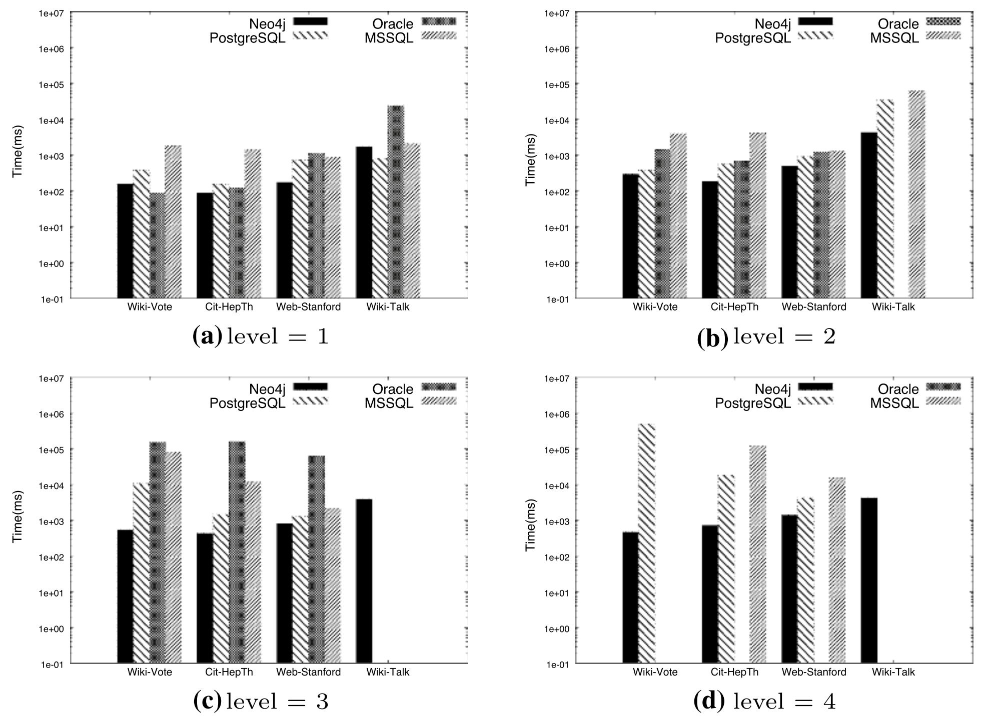


Fig. 10 Testing BFS algorithm over 4 datasets

4.2.3 Graph Algorithm Evaluation

We carry out graph algorithm evaluation over four real graphs. Figures 9 and 10 illustrate the performance of the five algorithms (mentioned in Sect. 3) over four directed graphs we have introduced in the part of datasets, namely Wiki-Vote, Cit-HepTh, Web-Stanford, and Wiki-Talk. We omitted the result for a graph algorithm if it fails to finish in 1 hour. In addition, the variable *level* represents the depth of recursive operations for BFS algorithm.

As shown in Fig. 9, we can observe that as expected, Neo4j has the best efficiency performance on these four real graphs.

Oracle and MSSQL fail to process LCC and WCC algorithms in large dataset in one hour, since LCC and WCC have massive sophisticated operations, including multi-levels recursive join operations which are time-consuming. Specifically, we find the intermediate results reach at a scale of ten billions for our *M* (see Table 4) size dataset when we execute LCC algorithm. WCC algorithm traverses all vertexes using *for ... in* loops, each of which involves several iterations to process *Join, Insert, Exists* operations. The number of iterations is an important factor in determining the performance. Departing from WCC algorithm which has multi-levels recursive join operation, CDLP and PR simply find adjacent vertexes for every vertex and loop several times according to a user-defined parameter. Moreover, as graph datasets scale from *S* to *XL*, the time consumption increases for PR and CDLP.

For BFS algorithm, we set the parameter *level* from 1 to 4; Fig. 10 shows that the time consumption of Neo4j is very close to RDBMSs on *level* 1, whereas, the performance differential is getting bigger as the *level* increases. When *level* reaches 4, RDBMSs cannot finish their processing in a certain time, namely stuck or terminated. We also conduct extensive tests by setting *level* to be larger than 4; only Neo4j processes the workload in a reasonable time. This result further proves that GDBMSs outperform RDBMSs in dealing with iterative and recursive operations. RDBMSs can just complete the operations having only a fewer depth of iterations and small scale of datasets in a reasonable time.

5 Conclusion

RDBMSs are ubiquitously used for storing, manipulating, and retrieving data over past decades. Meanwhile, the diversity and complexity of the applications led to challenges for RDBMSs to deal with sophisticated relationships between entities. For this reason, GDBMSs have lately received considerable attention which employs graph structures with vertices, edges, and properties to represent and store data, whereas both RDBMSs and GDBMSs are capable of managing relational data and graph data, making the boundaries of them unclear. To clearly answer the question, we have proposed a unified benchmark referring to existing benchmarks for RDBMSs and GDBMSs, which consist of relational workloads and

Table 10 Performance conclusion

Perspectives	Categories	Databases	Performance description
Data model	Relation model	RDBMSs	Outperformed in handling aggregate, group and sort large amounts of data, and algorithms with fewer iterations and less depth
	Graph model	GDBMSs	Outperformed in handling multi-table join and multilayer iterative calculation
Back-end storage Engine	Row storage	RDBMSs	Row-based storage engine used by RDBMSs did not stand out in either read or write operations
	Key-value storage	GDBMSs	Key-value storage engines with LSM -Tree outperforms in write operations
	Native graph storage	GDBMSs	Native storage of graphs performs well in read operations

Table 11 The comparisons between RDBMSs and GDBMSs

Database category	Performance summary
RDBMSs	RDBMSs outperform GDMBSs by a substantial margin under the workloads which mainly consist of group by, sort, and aggregation operations, and their combinations
GDBMSs	GDMBSs show their superiority under the workloads that mainly consist of multi-table join, pattern match, path identification, and their combinations

graph algorithms, to evaluate databases on the same datasets under the same metrics. We have implemented an inter-transfer between SQL and graph query languages for querying and importing data in RDBMSs and GDBMSs, respectively. We have conducted extensive experimental evaluations for existing popular RDBMSs and GDBMSs over both standard TPC-H and LDBC and reported our findings in detail. Meanwhile, we have conducted further comparative experiments to find out the effect of different data models and back-end storage engines on query performance for GDBMSs. We can conclude that, shown in Tables 10 and 11, RDBMSs outperform GDMBSs by a substantial margin under the workloads which mainly consist of group by, sort, and aggregation operations, and their combinations. GDMBSs show their superiority under the workloads that mainly consist of multi-table join, pattern match, path identification, and their combinations. For GDBMSs with different storage engines, key-value storage engines with LSM-tree outperform for write operations, while native graph storage engine has an advantage for read operations. Row storage engines do not stand out in either read or write operations. As a future work, we will study the optimization on both data models and back-end storage engines.

Acknowledgements This work was partially supported by National Key Research and Development Program of China under Grant 2018YFB10044401, National Natural Science Foundation of China under Grant 61732014, Beijing Municipal Science and Technology Project under Grant Z171100005117002.

References

1. Neumann T, Weikum G (2008) RDF-3x: A risc-style engine for RDF. Proc VLDB Endow 1(1):647–659. https://doi.org/10.14778/1453856.1453927
2. Weiss C, Karras P, Bernstein A (2008) Hexastore: sextuple indexing for semantic web data management. Proc VLDB Endow 1(1):1008–1019. https://doi.org/10.14778/1453856.1453965
3. Abadi DJ, Marcus A, Madden SR, Hollenbach K (2009) SW-store: a vertically partitioned DBMS for semantic web data management. VLDB J 18(2):385–406. https://doi.org/10.1007/s00778-008-0125-y
4. Angles R, Gutierrez C (2008) The expressive power of SPARQL. In: Proceedings of the 7th international conference on the semantic web, ser. ISWC '08. Springer, Berlin, pp 114–129 https://doi.org/10.1007/978-3-540-88564-1_8
5. Wylot M, Hauswirth M, Cudré-Mauroux P, Sakr S (2018) RDF data storage and query processing schemes: a survey. ACM Comput Survey 51(4):84:1–84:36. https://doi.org/10.1145/3177850
6. https://neo4j.com. Accessed 8 Nov 2019
7. https://www.arangodb.com. Accessed 8 Nov 2019
8. Zou L, Özsu MT, Chen L, Shen X, Huang R, Zhao D (2014) gStore: a graph-based SPARQL query engine. VLDB J 23(4):565–590. https://doi.org/10.1007/s00778-013-0337-7
9. Zou L, Mo J, Chen L, Özsu MT, Zhao D (2011) gStore: answering SPARQL queries via subgraph matching. Proc VLDB Endow 4(8):482–493. https://doi.org/10.14778/2002974.2002976
10. Zhao K, Yu JX (2017) All-in-one: graph processing in RDBMSS revisited. In: Proceedings of the 2017 ACM international conference on management of data, SIGMOD conference 2017,

Chicago, IL, USA, 14–19 May 2017, pp 1165–1180. https://doi.org/10.1145/3035918.3035943

11. Gao J, Jin R, Zhou J, Yu JX, Jiang X, Wang T (2012) Relational approach for shortest path discovery over large graphs, CoRR, arXiv:abs/1201.0232. Available: http://arxiv.org/abs/1201.0232
12. Gao J, Zhou J, Yu JX, Wang T (2014) Shortest path computing in relational DBMSS. IEEE Trans Knowl Data Eng 26(4):997–1011. https://doi.org/10.1109/TKDE.2013.43
13. De Leo D, Boncz P (2017) Extending SQL for computing shortest paths. In: Proceedings of the 5th international workshop on graph data-management experiences & systems, ser. GRADES'17. ACM, New York, NY, USA, pp. 10:1–10:8. Available: https://doi.org/10.1145/3078447.3078457
14. Qin L, Yu JX, Chang L, Tao Y (2009) Querying communities in relational databases. In: Proceedings of the 25th international conference on data engineering, ICDE 2009, 29 March 2009– 2 April 2009, Shanghai, China, pp 724–735 https://doi.org/10.1109/ICDE.2009.67
15. TPC-H (2012). https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf. Accessed 8 Nov 2019
16. Iosup A, Hegeman T, Ngai WL, Heldens S, Prat-Pérez A, Manhardto T, Chafio H, Capotă M, Sundaram N, Anderson M, Tănase IG, Xia Y, Nai L, Boncz P (2016) LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. Proc VLDB Endow 9(13):1317–1328. https://doi.org/10.14778/3007263.3007270
17. TPC-TPC-C (2010) https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. Accessed 8 Nov 2019
18. TPC-TPC-DS (2015) https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.1.0.pdf. Accessed 8 Nov 2019
19. https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-post-gresql-orientdb-neo4j-arangodb. Accessed 8 Nov 2019
20. https://info.tigergraph.com/benchmark. Accessed 8 Nov 2019
21. https://www.tigergraph.com. Accessed 8 Nov 2019
22. https://aws.amazon.com/neptune. Accessed 8 Nov 2019
23. Janusgraph distributed graph database 2017. http://janusgraph.org. Accessed 8 Nov 2019
24. Wood PT (2012) Query languages for graph databases. SIGMOD Record 41(1):50–60. https://doi.org/10.1145/2206869.2206879
25. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. The MIT Press, Cambridge
26. Ordonez C (2005) Optimizing recursive queries in SQL. In: Proceedings of the 2005 ACM SIGMOD international conference on management of data, ser. SIGMOD '05. ACM, New York, NY, USA, pp 834–839. https://doi.org/10.1145/1066157.1066260
27. Melton J, Simon A (2001) SQL:1999: understanding relational language components. Morgan Kaufmann Publishers Inc, San Francisco
28. Finkelstein IMSJ, Mattos N, Pirahesh H (1996) Expressing recursive queries in SQL, in ISO-IEC JTC1/SC21 WG3 DBL MCI, pp. X3H2–96–075
29. Raghavan UN, Albert R, Kumara S (2007) Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E 76:036106
30. Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University Press, New York
31. Krot A, Ostroumova Prokhorenkova L (2015) Local clustering coefficient in generalized preferential attachment models. In: Proceedings of the 12th international workshop on algorithms and models for the web graph, vol 9479, ser. WAW 2015. Springer, Heidelberg, pp 15–28. https://doi.org/10.1007/978-3-319-26784-5_2
32. Chitnis L, Das Sarma A, Machanavajjhala A, Rastogi V (2013) Finding connected components in map-reduce in logarithmic rounds. In: Proceedings of the 2013 IEEE international conference on data engineering (ICDE 2013), ser. ICDE '13. IEEE Computer Society, Washington, DC, USA, pp 50–61. https://doi.org/10.1109/ICDE.2013.6544813
33. https://github.com/hugegraph. Accessed 8 Nov 2019
34. Comer D (1979) Ubiquitous b-tree. ACM Comput Surveys (CSUR) 11(2):121–137
35. Sankaranarayanan J, Samet H, Varshney A (2006) A fast k-neighborhood algorithm for large point-clouds. In: SPBG, pp. 75–84
36. George L (2011) HBase: the definitive guide: random access to your planet-size data. O'Reilly Media Inc, Sebastopol
37. Cassandra A (2014) Apache cassandra, p 13. Website https://planetcassandra.org/what-is-apache-cassandra. Accessed 8 Nov 2019
38. Yang F, Dou K, Chen S, Hou M, Kang J-U, Cho S (2015) Optimizing NoSQL DB on flash: a case study of RocksDB. In: 2015 IEEE 12th international conference on ubiquitous intelligence and computing and 2015 IEEE 12th international conference on autonomic and trusted computing and 2015 IEEE 15th international conference on scalable computing and communications and its associated workshops (UIC-ATC-ScalCom). IEEE, pp 1062–1069
39. Hao F, Kodialam MS, Lakshman TV (2011) High accuracy bloom filter using partitioned hashing, US Patent 7,930,547
40. Dent A (2013) Getting started with LevelDB. Packt Publishing Ltd, Birmingham
41. https://hugegraph.github.io/hugegraph-doc/performance/hugegraph-benchmark-0.5.6.html. Accessed 8 Nov 2019