A Comparative Analysis of Knowledge Graph Query Performance

Masoud Salehpour and Joseph G. Davis
University of Sydney

Abstract. As Knowledge Graphs (KGs) continue to gain widespread momentum for use in different domains, storing the relevant KG content and efficiently executing queries over them are becoming increasingly important. A range of Data Management Systems (DMSs) have been employed to process KGs. This paper aims to provide an in-depth analysis of query performance across diverse DMSs and KG query types. Our aim is to provide a fine-grained, comparative analysis of four major DMS types, namely, row-, column-, graph-, and document-stores, against major query types, namely, subject-subject, subject-object, tree-like, and optional joins. In particular, we analyzed the performance of row-store Virtuoso, column-store Virtuoso, Blazegraph (i.e., graph-store), and MongoDB (i.e., document-store) using five well-known benchmarks, namely, BSBM, WatDiv, FishMark, BowllognaBench, and BioBench-Allie. Our results show that no single DMS displays superior query performance across the four query types. In particular, row- and column-store Virtuoso are a factor of 3-8 faster for tree-like joins, Blazegraph performs around one order of magnitude faster for subject-object joins, and MongoDB performs over one order of magnitude faster for high-selective queries.

Keywords: Knowledge Graph · query performance · SPARQL queries.

1 Introduction

The term Knowledge Graph (KG) was used by Google in 2012, referring to collecting information about real-world entities and their inter-relationships to facilitate the exploitation of semantics for searching the Web. From a broader perspective, any labeled directed graph-based representation of knowledge in a particular domain can be called a KG [14]. For example, the term KG has been used to refer to Semantic Web Linked Datasets such as DBpedia or YAGO. In the recent years, many organizations such as Amazon, Facebook, Microsoft, and Alibaba have created large KGs for different purposes ranging from semantic search, recommendations, reasoning, and data integration. However, unlocking KGs’ full potential in response to the growing deployment requires data frameworks to represent KG content and data platforms that can efficiently store the content and execute queries over them.

For the data framework, the World Wide Web Consortium (W3C) has recommended the Resource Description Framework (RDF) as a directed and labeled
graph-like structure for representation, integration, and exchange of the content of a KG using a large set of triples of the form <subject predicate object>. RDF offers a simple representation in which subjects and objects of triples are vertices of a graph that are connected by predicates as labeled edges. This simplicity can help provide an intuitive conceptualization of real-world entities and their inter-relationships. It can also represent diverse KG content ranging from structured to unstructured. However, this flexibility as well as the absence of an explicit schema and the heterogeneity of KG content pose a challenge to Data Management Systems (DMSs) for querying KGs efficiently since DMSs typically cannot make any priori assumptions about the structure of the KG content [8].

For the data platforms, DMS designers have employed a variety of design choices and architectures to tackle the above-mentioned challenges for querying KGs. For example, a variety of exhaustive indexing strategies, compression techniques, and dictionary encoding (i.e., to keep space requirements reasonable for excessive indexing) have been implemented by major native RDF-stores such as multiple bitmap indexes of Virtuoso or dictionary-based lexical values encoding of Blazegraph. A number of research prototypes have also been presented. For instance, [1] proposed a workload-adaptive and self-tuning RDF-store using physical clustering of the underlying data and [12] followed a RISC-style (reduced instruction set) architecture to leverage multiple query processing algorithms and optimization. However, the problem of storing and querying KGs efficiently continues to challenge DMS designers.

In addition to the design choices and architectures of DMSs, KG query performance is also affected by the diversity of SPARQL query types [2]. While the importance of these factors has been recognized, our understanding of the comparative performance of different types of queries across the major DMS types is somewhat limited. In this paper, we explore this problem, including the interactions between a DMS and query types. We provide a fine-grained, comparative analysis of four major DMS types, namely, row-, column-, graph-, and document-stores, against major types of KG queries, namely, subject-subject (aka, star-shape), subject-object (aka, chain-like or path), tree-like (aka, combined), and optional (aka, left-outer-join or OPT clauses) join queries. The performance of row-, column-, and graph-stores for executing queries has been studied in [2] based on their widespread use for processing RDF data. A widely accepted typology of KG queries is yet to emerge. At this stage query types such as subject-subject, subject-object, tree-like, and optional queries have been analyzed in previous research. Query types such as subject-subject, subject-object, and tree-like have been the focus of experiments in [19]. [3] has highlighted the importance of optional queries.

For our experiment, we selected row-store Virtuoso, column-store Virtuoso, Blazegraph, and MongoDB as representative DMSs for row-, column-, graph-, and document-store, respectively. We loaded five well-known benchmark datasets, namely, BSBM, WatDiv, FishMark, BioBench-Allie, and BowlognaBench into the DMSs separately. The benchmark queries were executed over each of the DMSs separately and query execution times computed to analyze the effects
of query types on the performance of different DMS types. Our contributions include:

- Comparative performance analysis and experimental evaluation of row-, column-, graph-, and document-stores in supporting the different SPARQL query types
- Providing explanations for the observed strengths and limitations of the different DMSs depending on the types of queries
- Communicating clear scientific and practical guidelines to researchers and practitioners through summarizing the lessons learned from our journey

The remainder of this paper is organized as follows. In Section 2 we provide some preliminary information about KG query types. Section 3 presents our experimental setup including the KG benchmark characteristics, computational environment, DMSs configuration, indexing, and data loading process. In Section 4 results of the query processing and related analyses are presented. We summarize the lessons learned from our research and discuss some of the limitations in Section 5. Section 6 highlights related work. We present our conclusions and future work in Section 7.

2 Query Types

In this section, we present some preliminary information about different query types using the “OperaHouse” KG example depicted in Fig. 1. The content of this KG can be represented by the following triples:

- OperaHouse located_in "Sydney"
- OperaHouse instance_of "landmark"
- OperaHouse instance_of "heritage site"
- OperaHouse instance_of "tourist attraction"

1 We use human-readable names in our examples in this paper.
An example of a query is then given below. It asks for the subject “OperaHouse’s” architectural style name from the KG in Fig. 1. “?styleName” is a variable to return the associated value (i.e., “expressionist”) as the result. Queries may contain a set of triple patterns such as “OperaHouse style ?styleName” in which the subject, predicate, and/or object can be a variable.

```
SELECT ?styleName
WHERE {
    OperaHouse style ?styleName .
}
```

Each triple pattern typically returns a subgraph. This resultant subgraph can be further joined with the results of other triple patterns to return the final resultset. In practice, there are four major types of join queries: (i) Subject-subject joins (aka, star-like), (ii) subject-object joins (aka, chain-like or a path), (iii) tree-like (i.e., combination of subject-subject and subject-object joins), and (iv) optional joins (aka, left-outer-join or OPT clauses).

**Subject-subject joins.** A subject-subject join is performed by a DMS when a KG query has at least two triple patterns such that the predicate and object of each triple pattern is a given value (or a variable), but the subjects of both triple patterns are replaced by the same variable. For example, the following query looks for all subjects of the KG in Fig. 1 that are located in “Sydney” and their style is “expressionist” (the result will be “OperaHouse”).

```
SELECT ?x
WHERE {
    ?x style "expressionist" .
    ?x located_in "Sydney" .
}
```

**Subject-object joins.** A subject-object join is performed by a DMS when a KG query has at least two triple patterns such that the subject of one of the triple patterns and the object of the other triple pattern are replaced by the same variable. For example, the following query looks for all subjects that are located within Australian cities (the result will be “OperaHouse”).

```
SELECT ?y
WHERE {
    ?x located_in "Australia" .
    ?y located_in ?x .
}
```
**Tree-like joins.** A tree-like joins consists of a combination of subject-subject and subject-object joins. For example, the following query requires a tree-like join to look for the opening date of “OperaHouse” (the result will be “20 Oct. 1973”).

```
SELECT ?y
WHERE {
  ?x opening_date ?y .
  ?x located_in ?z .
  ?z instance_of "capital" .
  ?z instance_of "metropolis" .
}
```

**Optional Joins.** Queries return resultsets only when the entire query pattern matches the content of the KG. However, optional joins allow KG queries to return a resultset even if the optional part of the query is not matched since completeness and adherence of KGs’ content to their formal ontology specification is not always enforced. This makes optional join a suitable tool for querying KGs. For example, the following query using optional join (in addition to a subject-subject join) to return “OperaHouse” as one of Sydney’s tourist attractions.

```
SELECT ?x
WHERE {
  ?x instance_of "tourist attraction" .
  ?x located_in "Sydney" .
  OPTIONAL {?x instance_of "zoo" .}
}
```

**Selectivity of Queries.** As above mentioned, each KG query contains a set of triple patterns where a triple pattern is a structure of three components which maybe concrete (i.e. bound) or variable (i.e. unbound). Sets of triple patterns specify the complexity of the access to the underlying data. When the number of stored triples satisfying sets of triple patterns’ conditions is large (i.e., as compared to the total number of stored triples), the corresponding query considered as low-selective [16]. In this regard, each query type can also be either high-selective or low-selective depending on the number of stored triples satisfying its triple patterns’ conditions.

3 Experimental Setup

In this section, we present our experimental setup including the KG benchmark characteristics, computational environment, DMSs’ configuration, indexing, and data loading process.
### Table 1: Statistics of the Benchmark datasets

| Benchmark       | Scale (nominal) | #Subjects | #Predicates | #Objects | #Triples    |
|-----------------|-----------------|-----------|-------------|----------|-------------|
| BSBM            | 10M             | 934,324   | 40          | 1,919,901| 10,190,687  |
|                 | 100M            | 9,197,305 | 40          | 15,207,734| 100,652,457 |
|                 | 1000M           | 91,647,129| 40          | 140,996,171| 1,004,406,629|
| WatDiv          | 10M             | 521,585   | 86          | 1,005,832 | 10,916,457  |
|                 | 100M            | 5,212,385 | 86          | 9,753,266 | 108,997,714 |
|                 | 1000M           | 52,120,385| 86          | 92,220,397| 1,092,155,948|
| BioBench-Allie  | 100M            | 19,227,252| 26          | 20,287,231| 94,420,988  |
| FishMark        | 10M             | 395,491   | 878         | 1,148,159 | 10,002,178  |

#### 3.1 Benchmarks

We used five well-known benchmarks in this research. These are publicly available datasets along with a collection of queries. These benchmarks are: mBerlin SPARQL Benchmark\(^2\) (BSBM)\(^3\), Waterloo SPARQL Diversity Test Suite\(^4\) (WatDiv)\(^5\), FishMark\(^6\), BowlognaBench\(^7\), and BioBench-Allie\(^8\). WatDiv and BSBM follow specific rules that allow us to scale the datasets to arbitrary sizes using their scale factors while other datasets are of fixed size. Table 1 shows the statistical information related to the above benchmarks. The RDF representations of these benchmarks are available in different formats such as N-Triples, Turtle, and XML. We used the RDF/N-Triples format. However, to load them into a document-store like MongoDB, we had to convert them to the JSON-LD\(^9\) format. We performed the conversion using a parser designed and developed as part of this project\(^10\).

We ran the benchmark queries against the corresponding datasets using the four DMSs. We selected twelve queries across the benchmarks that were representative of the major four query types. These queries provide varying degrees of selectivity and complexity\(^1\). The selected **subject-subject join** queries are: Query 5 from FishMark (**FishMark-Q5**), Query 7 from BowlognaBench (**BowlognaBench-Q7**), and Query 7 from WatDiv (**WatDiv-Q7**). The selected **subject-object join** queries are: Query 2 from BioBench-Allie (**BioBench-Allie-Q2**), Query 21 from WatDiv (**WatDiv-Q21**), and Query 22 from WatDiv (**WatDiv-Q22**) and the selected **tree-like join** queries are: Query 1 from BioBench-Allie (**BioBench-Allie-Q1**), Query 14 from BowlognaBench (**BowlognaBench-Q14**), and Query 19 from FishMark (**FishMark-Q19**). Finally, the selected **optional join** queries are: Query 2 from BSBM (**BSBM-Q2**), Query 4 from BSBM (**BSBM-Q4**), and Query 2 from FishMark (**FishMark-Q2**). For Wat-

---

\(^2\) [http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/](http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/)

\(^3\) [https://dsg.uwaterloo.ca/watdiv/](https://dsg.uwaterloo.ca/watdiv/)

\(^4\) [http://allie.dbcls.jp/](http://allie.dbcls.jp/)

\(^5\) [https://www.w3.org/2018/jsonld-cg-reports/json-ld/](https://www.w3.org/2018/jsonld-cg-reports/json-ld/)

\(^6\) The source code is available through [https://github.com/oursubmission/ESWC](https://github.com/oursubmission/ESWC)

\(^7\) These queries are available through [https://github.com/oursubmission/ESWC](https://github.com/oursubmission/ESWC)
Div and BSBM benchmarks, corresponding query execution times over KGs with 100M triples are reported in this paper while results for 10M and 1000M only available online due to space constraint.

3.2 System Settings

Computational Environment. Our benchmark system is a Virtual Machine (VM) instance with a 2.3GHz AMD Processor, running Ubuntu Linux (kernel version: 4.4.0-170-generic), with 48GB of main memory, 16 cores, 512K L2 cache, 5TB instance storage capacity. The VM cache read is roughly 2799.45MB/sec and the buffer read is roughly 35.85MB/sec (i.e., the output of the “hdparm -Tt” Linux command). The operating system is set with almost no “soft/hard” limit on the file size, CPU time, virtual memory, locked-in-memory size, open files, processes/threads, and memory size using Linux “ulimit” settings.

Data Management Systems (DMSs). We chose four different DMSs: (1) Row-store Virtuoso (Open Source Edition, version 06.01.3127), (2) Column-store Virtuoso (Open Source Edition, Version 07.20.3230–commit 4a668a5), (3) Blazegraph (Open Source Edition, version 2.1.5–commit 3122706), and (4) MongoDB (community edition, version: 4.0.9).

Configuration of row- and column-store Virtuoso. We configured both of them based on the vendor’s official recommendations. For example, we configured the Virtuoso process to use the main memory and the storage disk effectively by setting “NumberOfBuffers” to “4,000,000”, “MaxDirtyBuffers” to “3,000,000”, and “MaxCheckpointRemap” to “a quarter” of the database size as recommended. We also used the latest version of GNU packages that are necessary to build column-store Virtuoso (e.g. GNU gpref 3.0.4, libtool 2.4.6, flex 2.6.0, Bison 3.0.4, and Awk 4.1.3).

Configuration of Blazegraph. We configured it based on the vendor’s official recommendations as well. For example, we turned off all inference, truth maintenance, statement identifiers, and the free text index in our experiment since reasoning efficiency was not part of our research focus in this paper.

Configuration of MongoDB. We used its default settings. We set its level of profiling to “2” to log the data for all query-related operations for precise and detailed query execution time extraction.

Indexing of Virtuoso. We did not change the default indexing scheme of Virtuoso (both row- and column-store). As highlighted in the official website, “alternate indexing schemes are possible but will not be generally needed”. More specifically, Virtuoso’s data modeling is based on a relational table with three columns for S, P, and O (i.e., S: Subject, P: Predicate, and O: Object)

8 These queries are available through https://github.com/oursubmission/ESWC
9 Previously known as Bigdata DB.
10 http://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFPerformanceTuning
11 https://wiki.blazegraph.com/wiki/index.php/PerformanceOptimization
12 http://docs.openlinksw.com/virtuoso/rdfperfsrcmdline
13 In the case of loading named graphs, it adds another column for the context, called C.
and carrying multiple indexes over that table to provide a number of different access paths. Most recently, column-store Virtuoso added columnar projections to minimize the on-disk footprint associated with RDF data storage. Virtuoso (both row- and column-store) creates the following compound indexes by default for the loaded KG: PSO, PO, SP, and OP.

**Indexing of Blazegraph.** As recommended in its official website[^14], we did not change its default indexing schema. In Blazegraph, indexes are based on “B+Trees” data structure. Blazegraph typically uses the following three indexes for triples modes: SPO, POS, and OSP. For normal use cases, these indexes are laid out on variable sized pages. These index pages are read from the backing store and load in the main memory on demand (i.e., into the Java heap). However, Blazegraph takes advantage of a variety of data structures to execute queries when stored KG content is loaded in the main memory. For example, the underlying data structure is retained by a mixture of a ring buffers on the stack alongside a native memory cache for buffering writes to reduce write application effects.

**MongoDB Storage Layouts.** We did not change its default storage engine which is a key/value store, namely, WiredTiger, to store JSON documents.[^15] MongoDB usually assigns an arbitrary (and unique) identifier to each JSON document as a key and considers the document as a value. It uses B-Trees to create indexes on the contents of each JSON document.

**Indexing of MongoDB.** We created indexes on those name/value pairs of the JSON-LD that were representatives of subjects and predicates.

**Loading the benchmark KG.** We loaded the RDF/N-Triples format of benchmarks into Virtuoso (row- and column-store) by using its native bulk loader function (i.e., “ld_dir”). To load the KGs into Blazegraph, we used its native “DataLoader” utility[^16]. We loaded KGs into MongoDB using its native tool, called “mongoimport”.

**Shutdown store, clear caches, restart store.** We measured the query execution times in our evaluation. This is an end-to-end time computed from the time of query submission to the time when the result is outputted. After the execution of each query, we carefully checked to ensure that the output results are correct and exactly the same across different DMSs. The query times for both cold- and warm-run (aka, cold and warm cache) are reported. For cold-run we dropped the file systems caches using `/bin/sync`, `echo 3 > /proc/sys/vm/drop_caches`, and `swapoff -a & swapon -a` commands. For fairness, the warm-run query times reported for each DMS are averaged[^17] over 5 successive runs (with almost no delay in between).

[^14]: <https://wiki.blazegraph.com/wiki/index.php/PerformanceOptimization>
[^15]: MongoDB uses the binary equivalent of each JSON document (i.e., BSON) for storage, in which the structure of each document remained unchanged.
[^16]: <https://wiki.blazegraph.com/wiki/index.php/Bulk_Data_Load>
[^17]: Geometric mean is used.
4 Evaluation

Subject-subject Joins. The query execution times are shown in Fig. 2 in which X axis represents the DMSs and the Y axis shows the execution times of queries, namely, FishMark-Q5, BowlognaBench-Q7, and WatDiv-Q7 in milliseconds (using log scale). Fig. 2 shows that MongoDB runs this type of queries over one order of magnitude faster than the other DMSs when queries are high-selective. For example, MongoDB executed FishMark-Q5 in 2.19 milliseconds (warm-run) while Blazegraph, column-store Virtuoso, and row-store Virtuoso executed the same query in 394.24, 89543.81, and 29045.86 milliseconds, respectively. However, our results show that Blazegraph performs at least 2x faster than other DMSs when subject-subject join queries are low-selective (e.g., WatDiv-Q7). In Fig. 2, the differences between cold- and warm-run show that Virtuoso (row- and column-store) can take advantage of caching techniques more than other DMSs. For example, Virtuoso (row and column) executes BowlognaBench-Q7 in over 1500 milliseconds (cold-run) while its execution time is around 150 milliseconds in a warm-run.

Subject-object Joins. The query execution times for the selected subject-object join queries is shown in Fig. 3 in which X axis shows the DMSs and the Y axis shows the execution time of queries, namely, BioBench-Allie-Q2, WatDiv-Q21, and WatDiv-Q22 in milliseconds (using log scale). Although MongoDB executed BioBench-Allie2-Q2, which is a high-selective query, over 2 orders of magnitude faster than other DMSs, it could not finish the execution...
of WatDiv-Q21 and WatDiv-Q22 within the given time-out period of 50,000 milliseconds. The complexity and non-selectivity of these two queries may have contributed to the unsuccessful execution over MongoDB. However, Fig. 3 shows that other DMSs performed comparably. For instance, WatDiv-Q21 executed over Blazegraph in around 570 milliseconds (warm-run) where this execution time is equal to 118.38 and 374.6 milliseconds for column-store Virtuoso and row-store Virtuoso, respectively.

Tree-like Joins. The query execution time for the selected tree-like join queries is shown in Fig. 4. This figure shows that row- and column-store Virtuoso performed similarly for warm-run execution of BioBench-Allie-Q1 and FishMark-Q19 while Blazegraph is around 5x slower. MongoDB appeared to be the slowest for warm-run execution of BioBench-Allie-Q1 while its performance is comparable with Blazegraph for FishMark-Q19. BowlognaBench-Q14 executed around 2 orders of magnitude faster using MongoDB, probably because it is a high-selective tree-like join query. The comparison between cold- and warm-run execution of FishMark-Q19 can also give rise to the importance of the role that caching techniques play in query performance where MongoDB is the fastest in cold-run, but in warm-run, it is almost the slowest (after Blazegraph).

Optional Joins. The query execution time for the selected optional join queries is shown in Fig. 5 in which MongoDB executed them faster than other DMSs. High-selectivity of the selected queries may have been an important factor in MongoDB’s performance advantage. Row-store Virtuoso was the slowest across others while column-store Virtuoso performed over 3x faster than Blazegraph to run these queries (warm-run). However, in the cold-run, aside from the
A Comparative Analysis of Knowledge Graph Query Performance

Fig. 4: Impacts of tree-like join queries on the DMSs (cold- and warm-run).

4.2 Analysis

Subject-subject Join. Our results showed that MongoDB can execute this query type over one order of magnitude faster, especially for queries with high selectivity. Virtuoso (both row- and column-store) and Blazegraph typically execute subject-subject join by scanning indexes for each triple pattern separately. The retrieved result of each triple pattern is kept in the main memory as an intermediary result. These DMSs join different intermediary results to return the final result. Virtuoso (both row- and column-store) and Blazegraph typically use a hash join algorithm for executing subject-subject joins over the intermediary results. However, in MongoDB, all triples with the same subject have appeared in a single JSON document and the joining of triples with the same subject is equivalent to an index-based look-up querying of a given subject. Therefore, we observed better performance from MongoDB for high-selective subject-subject join queries.

Subject-object Joins. We observed that Blazegraph offers a significant performance improvement on the cold-run execution of subject-object join queries. Merge join is known to be an efficient algorithm to be implemented by DMSs for running subject-object join over intermediary results, after scanning indexes, to return the final result. To the best of our knowledge, none of the DMSs have implemented the merge join as a part of their query processing engines. As

Please note that [10] did not use the exact term “subject-object join”, instead it refers to this query type by its definition.
a result, these DMSs use Index Nested Loop join algorithm to support subject-object queries. In our experiments, the faster query execution time of Blazegraph (cold-run) for this query type may stem from the use of $B^+$-tree-based index nested loop join which is more read-optimal as compared to bitmap index-based of both row- and column-store Virtuoso. In addition, Blazegraph uses cardinality estimation to predict the size of the intermediary results of queries to find a good join ordering. This estimation requires dynamic programming techniques and the building of statistical summaries such as histograms. Such cardinality estimation has a significant performance effect on the execution time of low-selective subject-object join queries. Therefore, we observed better performance from Blazegraph as compared to row- and column-store Virtuoso for cold-run execution of subject-object join queries.

**Tree-like Joins.** Our results showed that Virtuoso (specially column-store) executed tree-like join queries faster. Technically, a tree-like join can be considered as a combination of subject-subject and subject-object joins. The performance of tree-like query types may vary depending on the complexity of such a combination and the efficiency of DMSs’ query optimizer. In our experiments, Virtuoso (row- and column-store) showed better performance for executing this query types with lower selectivity. We speculate that Virtuoso’s vectorized query execution model and its secondary indexing strategies (aka, compound indexes) along with its well-engineered query optimization engine may explain the better performance as compared to others. In addition, column-store Virtuoso usually stores indexes more compactly. Therefore, it can store and index short, fixed-length identifiers rather than string literals of subject, predicate, and object values. This compactness typically contributes to faster index selection in query planning and has a positive performance impact on tree-like join queries.
Optional Joins. There is no evidence that any of the four DMSs has implemented specialized optimizations for optional joins. As a result, although MongoDB showed better performance for running high-selective queries, we do not observe significant differences in query performance among the DMSs for low-selective queries. We note that both row- and column-store Virtuoso come with a compression strategy for storing KG datasets. Furthermore, bitmap indexing provides row- and column-store Virtuoso with better space utilization as compared to $B^+$-tree of Blazegraph (or MongoDB). In this regard, we speculate Virtuoso is likely to aggressively prune intermediate results and perform faster than others for optional join query processing, especially for low-selective optional join queries.

Scale effects. FishMark, BioBench-Allie, and BowlognaBench are fixed-size datasets that cannot be scaled. In contrast, WatDiv and BSBM are scalable. In this paper, we reported the corresponding query execution times of these two datasets with 100M triples. However, corresponding results for datasets with 10M and 1000M are computed and are available online. Our results showed that in most cases, selectivity and query type along with query optimization and caching techniques are probably more significant contributory factors to the performance differences across employed DMSs as compared to the size of the underlying dataset (i.e., the scale factor).

5 Discussion

5.1 Limitations

Our results indicate that no single DMS displays superior query performance across different query types. These results are likely to be generalizable. However, more experimentation is warranted before we can arrive at any firm conclusions. In our experiments, we had four archetypal query types. However, there may be other query types that we need to consider in the future.

Currently, the maximum size of each JSON document in MongoDB is 16MB. It rejects JSON documents when its size exceeds this value. Technically, the maximum document size in document-stores helps ensure that a single document cannot use an excessive amount of memory, but the JSON-LD representation of KGs might be affected negatively by this. In our experiments, there were no cases in which the document size exceeded the maximum value. However, in principle, the size of JSON documents may exceed the maximum document size depending on the KG content.

Another issue that remains to be addressed is the automatic conversion of SPARQL to JavaScript-like (i.e., for MongoDB) queries. In our experiments, we converted the benchmark queries manually and after the execution of each query, we carefully checked to ensure that the output results are correct and exactly the same across different DMSs and representations.

19 https://github.com/oursubmission/ESWC
5.2 Lessons Learned

Intermediary Result. We note that the performance of different query types tends to be negatively affected by the sizes of the query’s output and more often its intermediary results. When a query type contains more than a triple pattern, DMSs usually have to scan large parts of indexes for each triple pattern and then join the result of these scans. These index scans would produce large intermediary results. We observed that even when the query itself is very selective with small output, the size of the intermediary results can be still very large. The size of the intermediary result challenges DMSs. Currently, DMSs usually use either of two techniques data compression or Sideways Information Passing (SIP) to decrease the size of intermediary results. It appears that employing these techniques to decrease intermediary results may increase the computation need of the query evaluation process for the uncompression or additional filtering (for SIP) requirement.

Locality. Column-store Virtuoso and MongoDB are designed to increase data locality while storing KGs’ content more than others. In the column-store Virtuoso storage model, each column of a table or index is stored contiguously to provide physical adjacency. Therefore, when queries (e.g., tree-like joins) need to access a subset of columns from one table, only those columns actually being accessed need to be read from disk which can be culminated to better use of I/O throughput and memory. This locality has this potential to reduce the traffic between CPU cache and main memory and provide a better CPU utilization. MongoDB similarly takes advantage of data locality since all the triples related to one resource (i.e., a subject in the JSON-LD) are physically located together. We speculate that such locality leads to denser data layout, more CPU cache (i.e., L2 cache) locality and more RAM locality and therefore increased overall performance on high-selective KG queries.

Cache Efficiency. DMSs usually utilize their internal and the underlying filesystem cache memory. When enough free memory available and allocated to DMSs, efficient utilization of this memory for caching purposes can typically contribute to faster warm-run query execution. Comparing the results of different queries across the DMSs in cold- and warm-run query execution suggested that column-store Virtuoso provides better cache management. In applications with ad-hoc queries, the cache management may not impact the performance significantly, but for cases in which a number of queries are repeated periodically, employing suitable cache techniques can positively contribute to query performance.

6 Related Work

Early approaches employed relational database systems to store the Semantic Web datasets. In addition, several approaches have exploited NoSQL databases for building DMSs as discussed in [19]. We can classify these studies into three categories: Triple-based Indexing. HexaStore [17] is a well-known DMS based on indexing. This created indexes on all permutations of the triple pattern. The
effectiveness of triple-based indexing solutions can be limited since querying KGs typically requires touching a large amount of data and complex filtering. **Infrastructure Configuring.** JenaHBase [11] and H2RDF [13] are well-known DMSs that focused mainly on the configurations of underlying infrastructure such as cluster segmentation, communication overhead, and distributed storage layouts. **Graph Processing.** Graph-based stores usually model Semantic Web data as a labeled and directed multi-edge graph by using a disk-based adjacency list table and executes queries by mapping them to a sub-graph matching task over the graph.

In addition to the design of the DMSs, analysis of available DMSs using benchmark datasets has been a core topic of Semantic Web data management research. For example, some studies such as [5,2,9] presented new benchmark datasets. Some other studies such as [6] did not propose any new dataset, but tried to use available benchmarks and DMSs for the same purpose such as reporting key advantages and drawbacks of each DMS. There are also studies such as [15] which comprehensively surveyed and analyzed available datasets in terms of different metrics such as the number of projection variables, the number of BGPs, etc. However, to the best of our knowledge, our paper is one of the first that investigated the comparative KG query performance by mapping archetypal SPARQL query types with different DMS types.

7 Conclusion

We have focused on the mapping of different types of KG queries onto different types of DMS. We analyzed the performance of row-store Virtuoso, column-store Virtuoso, Blazegraph (i.e., graph-store), and MongoDB (i.e., document-store) using five well-known benchmarks, namely, BSBM, WatDiv, FishMark, BowlognaBench, and BioBench-Allie. A summary of our findings is as follows:

- There are significant interaction effects between different types of DMSs and query types.
- Our results showed that the simplicity of the underlying storage layout, increasing data locality, and suitable caching techniques in Virtuoso (specially column-store) lead to a performance advantage for tree-like join queries by generating smaller intermediary results.
- We also found that suitable cardinality estimation as well as efficient query optimization of Blazegraph offers a significant performance improvement on the cold-run execution of subject-object join queries.
- Taking advantage of data locality and employing efficient data structures such as B-trees for implementing indexes in MongoDB can contribute to over one order of magnitude better performance for executing subject-subject join queries, especially for queries with high selectivity.

The results presented in this paper can assist in the benchmarking of the emerging type of DMSs. However, more experimentation is warranted before we can arrive at any firm conclusions. In addition, our experience while performing
a comparative analysis of KG query performance raised several new and interesting questions and research directions that need to be addressed in the future. These include replication of this research using more datasets and DMSs and automatic rewriting of SPARQL queries to other declarative query languages such as MongoDB’s query language.

References

1. G. Aluç, M. T. Özu, and K. Daudjee. Building self-clustering rdf databases using tunable-lsh. *The VLDB Journal*, 28(2):173–195, 2019.
2. G. Aluç et. al. Diversified stress testing of RDF data management systems. In *Proc. of the Int. Semantic Web Conf. (ISWC)*, pages 197–212, 2014.
3. M. Atre. Left bit right: For SPARQL join queries with OPTIONAL patterns (left-outer-joins). In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*, pages 1793–1808, 2015.
4. S. Bail et. al. FishMark: A linked data application benchmark. In *Proc. of the Int. Workshop on Scalable and High-Performance Semantic Web Systems (SSWS)*, pages 1–15, 2012.
5. C. Bizer et. al. The Berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5:1–24, 2009.
6. P. Cudré-Mauroux et. al. NoSQL databases for RDF: An empirical evaluation. In *Proc. of the Int. Semantic Web Conf. (ISWC)*, pages 310–325, 2013.
7. G. Demartini et. al. Bowlognabench—benchmarking rdf analytics. In *Proc. of the Int. Symp. on Data-Driven Process Discovery and Analysis (SIMPDA)*, pages 82–102, 2012.
8. S. Duan et. al. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*, pages 145–156, 2011.
9. O. Erling et. al. The LDBC social network benchmark: Interactive workload. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*, pages 619–630, 2015.
10. S. Groppe. *Data management and query processing in Semantic Web databases*. Springer Science & Business Media, 2011.
11. V. Khadilkar et. al. Jenia-hbase: A distributed, scalable and efficient RDF triple store. In *Proc. of the Int. Semantic Web Conf. (ISWC)*, pages 85–88, 2012.
12. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *Proc. VLDB Endow.*, 19(1):91–113, 2010.
13. N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: Adaptive query processing on RDF data in the cloud. In *Proc. of the Int. Conf. on World Wide Web (WWW)*, pages 397–400, 2012.
14. H. Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8(3):489–508, 2017.
15. M. Saleem et. al. How representative is a SPARQL benchmark? an analysis of RDF triplestore benchmarks. In *Proc. of the Int. Conf. on World Wide Web (WWW)*, pages 1623–1633, 2019.
16. M. Stocker et. al. Sparql basic graph pattern optimization using selectivity estimation. In *Proc. of the Int. Conf. on World Wide Web (WWW)*, pages 595–604, 2008.
17. C. Weiss et. al. Hexastore: Sextuple indexing for Semantic Web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, 2008.
18. H. Wu et. al. BioBenchmark Toyama 2012: an evaluation of the performance of triple stores on biological data. *Journal of Biomedical Semantics*, 5(1):32–43, 2014.
19. M. Wylot et. al. RDF data storage and query processing schemes: A survey. *ACM Comput. Surv.*, 51(4):84:1–84:36, 2018.