Relational Schemata for Distributed SPARQL Query Processing

Victor Anthony Arrascue A.
University of Freiburg
arrascue@cs.uni-freiburg.de

Polina Koleva
University of Freiburg
polina.n.koleva@gmail.com

Anas Alzogbi
University of Freiburg
alzogbi@cs.uni-freiburg.de

Matteo Cossu[*]
University of Freiburg
elcossu@gmail.com

Michael Färber
University of Freiburg
michael.faerber@cs.uni-freiburg.de

Patrick Philipp[†]
University of Freiburg
philipp@cs.uni-freiburg.de

Guilherme Schievelbein
University of Freiburg
schieveg@cs.uni-freiburg.de

Io Taxidou
University of Freiburg
taxidou@cs.uni-freiburg.de

Georg Lausen
University of Freiburg
lausen@cs.uni-freiburg.de

ABSTRACT

To benefit from mature database technology RDF stores are built on top of relational databases and SPARQL queries are mapped into SQL. Using a shared-nothing computer cluster is a way to achieve scalability by carrying out query processing on top of large RDF datasets in a distributed fashion. Aiming to this the current paper elaborates on the impact of relational schema design when queries are mapped into Apache Spark SQL. A single triple table, a set of tables resulting from partitioning by predicate, a single wide table covering all properties, and a set of tables based on the application model specification called domain-dependent-schema, are the considered designs. For each of the mentioned approaches, the rows of the corresponding tables are stored in the distributed file system HDFS using the columnar-store Parquet. Experiments using standard benchmarks demonstrate that the single wide property table approach, despite its simplicity, is superior to other approaches. Further experiments demonstrate that this single table approach continues to be attractive even when repartitioning by key (RDF subject) is applied before executing queries.

CCS CONCEPTS

• Information systems → Relational parallel and distributed DBMSs; **Resource Description Framework (RDF)**; **• Computer systems organization → Cloud computing**.

KEYWORDS

Relational Schema, RDF, SPARQL, Spark SQL, Parquet

[*]current affiliation: Ultra Tendency GmbH, Niels-Bohr-Str. 10c, 39106 Magdeburg, Germany. Email: matteo.cossu@ultratendency.com
[†]current affiliation: FZI Forschungszentrum Informatik am KIT, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany. Email: philipp@fzi.de

SBD'19, July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6766-0/19/07…$15.00
https://doi.org/10.1145/3323878.3325804

ACM Reference Format:
Victor Anthony Arrascue A., Polina Koleva, Anas Alzogbi, Matteo Cossu, Michael Färber, Patrick Philipp, Guilherme Schievelbein, Io Taxidou, and Georg Lausen. 2019. Relational Schemata for Distributed SPARQL Query Processing . In *The International Workshop on Semantic Big Data (SBD'19), July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3323878.3325804

1 INTRODUCTION

RDF data is a popular framework for publishing data [3] and building knowledge graphs [9]. Therefore, during the last years much work has been done to achieve efficient processing of RDF stores using SPARQL. Early efforts to build SPARQL endpoints, most notably Jena [27], and Sesame [7] and later RDF-3X [16], YARS2 [12], and DB2RDF [6], are using single-node machine environments empowered by sophisticated storage structures and index support. To achieve better scalability when handling very large graphs, systems designed for distributed environments have been proposed and investigated. In this context, partitioning the data among different nodes of a computer cluster is a major critical task. Some distributed systems implement vertical partitioning, i.e. using a predicate-based partitioning approach [10, 14, 15, 22, 25], while others apply subject-based partitioning for handling a single triple table [2]. Other designs use a single wide property table covering all properties [24], or a combination of vertical partitioning and wide property table [8]. Finally, some systems suggest partitioning based on graph-covers [11, 21]. For a more comprehensive review of the relevant literature, we refer the reader to [13, 28].

We are interested in mapping SPARQL queries into SQL to be able to reuse mature robust relational database technology. The choice of relational table design plays a crucial role for efficiency. Different designs not only imply different kinds of physical data partitions, but also different translations to SQL operations. Both aspects are among the most crucial factors for distributed processing efficiency. Investigations of various schema designs for RDF data have been started by [1, 18]. Recently, for a single-node system, the analysis provided in [19] gives arguments for using an *emergent schema*, which corresponds to a domain-dependent-schema in our current work. However, in that paper and in [5], when discussing Virtuoso Cluster[1], it is left unanswered, whether an emergent schema is also beneficial for the distributed case.

[1]http://docs.openlinksw.com/virtuoso/clusteroperation/

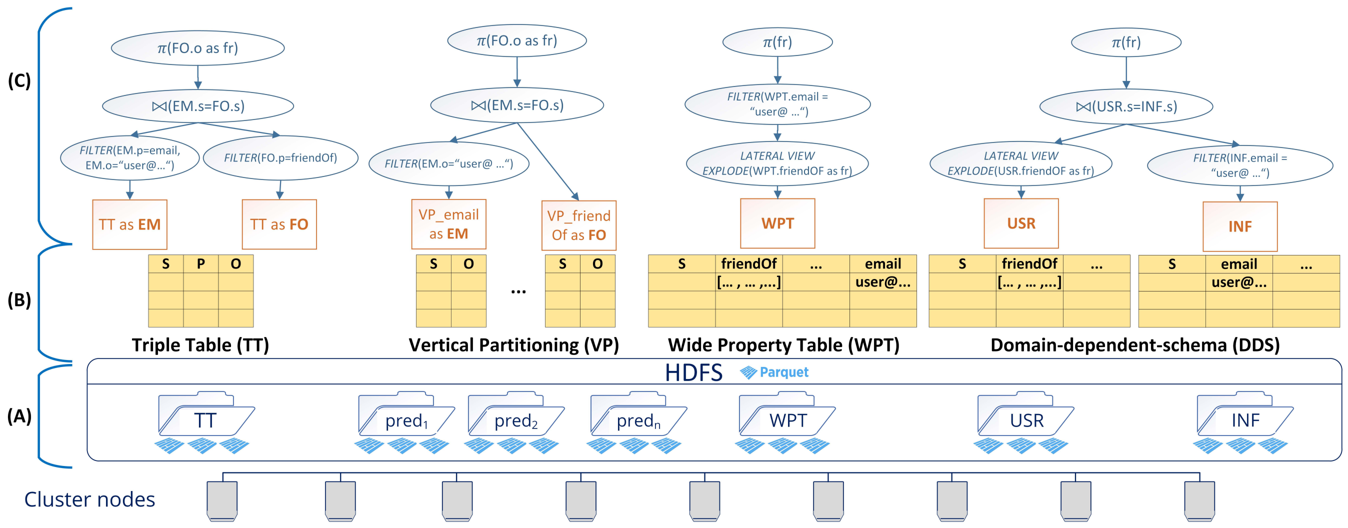

Figure 1: (A) Tables storage in HDFS at the physical layer (B) Logical tables and their corresponding relational schemata. (C) Algebra Tree for Query Q1 for the different schemata.

We are approaching the analysis of the relational schemata using Apache Hadoop as the distributed processing platform and mapping SPARQL queries into Spark SQL. Differently from our previous research on distributed SPARQL query processing [8, 24, 25], in the current paper we systematically analyze the different designs. Our experiments take into account several aspects, such as the impact of broadcast-joins and partitioning based on RDF subject. Not only we conduct experiments on a synthetic benchmark, WatDiv, but also on a real RDF-graph, Yago.

The main contribution of the current paper is twofold. We demonstrate experimentally, that using a single wide property table (WPT) for relational table design provides a level of efficiency superior to triple table (TT), vertical partitioning (VP) and domain-dependent-schema (DDS). Further experiments demonstrate that the single wide property table approach persists to be attractive even when a repartitioning by key (RDF subject) is applied before executing queries. While the gain in efficiency of WPT in these experiments is moderate compared to other schemata, WPT, in contrast to DDS or an emergent schema, does not require complex additional design considerations and remains nearly stable during RDF-graph evolution. In addition, our evaluation exhibits, that reducing the number of joins in the execution plan and having a balanced distribution of the data among the cluster nodes are more important than the size of the various tables accessed.

The paper is structured as follows. In Section 2 we elaborate on the technical background of our experimental work. Section 3 presents our approach for SQL query construction. Section 4 discusses and summarizes the evaluation results, and finally, Section 5 concludes the paper.

2 BASICS

Spark Input. Given an RDF graph, the data is transformed in a distributed relational-table form, so that we can apply queries over these tables using Spark SQL. The tables are stored within a Hive database so that they can be consistently accessed by different applications. Figure 1 shows the kind of relational schemata we consider in this work, how the tables appear at the logical level in Hive (B), and how the data is placed into such database (A). Each Hive table is stored in one HDFS folder and the data files in there are distributed among the nodes. The data itself is compressed and stored using the Parquet format, a columnar storage. These technologies allow us to: (1) have an efficient representation of sparse tables (Parquet ignores *null* values during the encoding), and (2) being able to store multiple values in some columns if required. In contrast to classic relational systems, this can be easily achieved in Spark, Hive and Parquet thanks to their support for complex data types, such as arrays or maps.

The distribution of data within a Hive table depends on how the data was built, i.e. in our case it reflects the distribution of the result of a Spark SQL query, which was persisted as the table. Consequently, the HDFS-blocks containing the data are not necessarily full. Therefore, when a Spark *job* is created to evaluate a query, by default, the data of the involved tables is eventually reorganized into *partitions* aiming to fill the HDFS-block size[2].

Query Execution Workflow. The SQL operations involved in the distributed execution of a Spark SQL query are of two kinds: *transformations* and *actions*. The first ones are *lazily* evaluated, i.e. they are computed only when an action requires them to be executed. For instance, a join operation (⋈) is a transformation, whereas counting the number of rows (*count*()) is an action. Let $(t_1 \bowtie t_2).count()$ be a Spark SQL expression. The execution of the join is triggered only when the counting action is invoked. In the system an action triggers the launch of a Spark job. This starts with the creation

[2]For instance some data files could be placed together in a single partition. The actual procedure depends on the data format. Since we have Parquet files, the specific Parquet reader is responsible for building these partitions.

of an execution plan for carrying out a job. For this, Spark examines the lineage of operations on which that action depends[3]. For the above-mentioned example the operations are *scan*(...), *join*(...), and *count*(). These operations have dependencies on each other. Clearly, *count*() depends on the computation of *join*(...), and at the same time this depends on reading the corresponding tables. The simplicity of this action should not suggest that this is always the case. As a matter of fact, an action can have a very complex graph lineage which depends on many transformations.

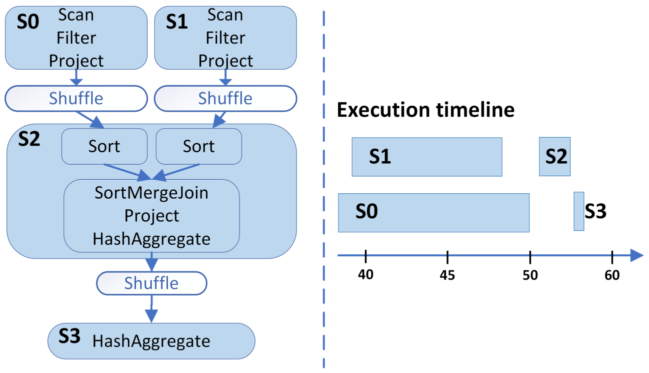

Figure 2: Stage dependencies and their execution timeline.

The dependencies are hence used to assemble the job's transformations into *stages*. Each stage consists of a sequence of transformations which do not require *shuffling* data, i.e. transferring data from one node to another one. Shuffling occurs then in between stages. Therefore, stages without dependencies might be executed in parallel. Figure 2 illustrates the stages dependencies and an execution timeline for the above considered example, $(t_1 \bowtie t_2).count()$, obtained from a real cluster. As one can notice, the two *scan*() operations (one for each table) can be carried out as two parallel stages (S0 and S1) because they do not depend on each other. This is followed by a shuffle carried out for the next stage, the join, which requires all partitions with the same key to be in the same data node (S2). Finally, a count is executed as a separate stage (S3) to aggregate the counters sent by each node.

Going deeper into the hierarchy, a stage corresponds to a collection of *tasks*, each of which executes the same code on a different partition. To carry out a task, an *executor*, a sort of tasks monitor, is initialized in each partition's node[4]. An executor can run multiple tasks over its lifetime and multiple tasks concurrently. The number of concurrent task-slots is *numExecutors* $*$ *ExecutorCores*. For example, with 10 running executors and 6 cores in each machine, 60 concurrent tasks can run in parallel. Thus, a task represents the fundamental unit of work on a partition of a distributed dataset.

This is why the number of partitions, together with the interdependencies between stages, play such an important role in the parallelization degree in the SQL query evaluation. Overall, the ideal job distribution occurs when: (1) Data is evenly distributed among all nodes. This is achieved by the HDFS block-based distribution and data balancing; (2) All cluster nodes are involved in the computation, i.e. all cores are busy running tasks in parallel; (3) The shuffling is minimal, i.e. each task is carried out locally[5]; (4) Queries are able to reuse relevant in-memory or *cached* data.

3 SQL QUERY CONSTRUCTION

SPARQL to Spark SQL. The translation process starts with the construction of a SPARQL algebra tree for each corresponding schemata. We consider a single triple table (TT), a set of tables resulting from partitioning by predicate (VP), a single wide property table covering all properties (WPT) and a domain-dependent-schema (DDS) based on the corresponding data specifications. Further details about our DDS design are provided in the next Section. Consider query Q1 whose purpose is to find a list of friends `?fr` of a user identified by her email address. The tokens in red, which denote constants are important to eventually reduce the number of intermediate results.

```
SELECT DISTINCT ?fr
WHERE {
   ?usr watdiv:friendOf ?fr .
   ?usr sorg:email "user@email.com"
}
```

Query Q1: Finding friends of user

Figure 1(C) illustrates the built algebra trees generated for each of the considered schemata. The algebra trees are then translated by our system to the corresponding Spark SQL queries, which are equivalent to those shown in the following tables[6]:

```
TT:SELECT fr FROM
     (SELECT s as usr FROM TT
       WHERE p='<.../email>' AND o='user@email.com') EM
   JOIN
     (SELECT s as usr, o as fr FROM TT
       WHERE p='<.../friendOf>') FO
   ON EM.usr = FO.usr
```

```
VP:SELECT fr FROM
     (SELECT s as usr FROM VP_email
       WHERE o='user@email.com') EM
   JOIN
     (SELECT s as usr, o as fr FROM VP_friendOf) FO
   ON EM.usr = FO.usr
```

```
WPT:SELECT fr FROM
      (SELECT s as usr, EXP_fo as fr, COL_email as em
        FROM WPT
        lateral view explode(COL_friendOf) FO as EXP_fo
        WHERE COL_email IS NOT NULL AND COL_email='user@email.com')
```

```
DDS:SELECT fr FROM
     (SELECT s as usr, EXP_fo as fr FROM USR
       lateral view explode(COL_friendOf) FO as EXP_fo) USR
   JOIN
     (SELECT s as usr, COL_email as em FROM INF
       WHERE COL_email IS NOT NULL AND COL_email='user@email.com') INF
   ON USR.usr = INF.usr
```

Translations to Spark SQL

As one can notice, the queries are different. While TT accesses a single distributed table, requires a self-join, and three filters, VP requires a join between two distributed tables and applies a single filter. In contrast, the algebra trees of WPT and DDS have the same

[3]This is also known as the lineage graph, a graph of the transformations which have to be executed after an action is been called.

[4]To determine which nodes are assigned to run the *executors*, there is a previous resource negotiation with the resources manager, in our case Yarn, which finds the available hosts. This allocation is not fixed: Spark's dynamic allocation feature allows it to dynamically scale the number of executors based on the workload.

[5]This obviously depends on the task. For instance, searching for a specific value can be done locally. However, some operations such as a join might require moving partitions around the network, because it requires join partners to be in the same location. Shuffles have overall the most negative impact on the time performance.

[6]In practice our application uses the Spark SQL Java API: https://spark.apache.org/docs/2.2.0/sql-programming-guide.html.

operations, which are however applied either to a single table or to multiple tables, respectively. In the example's DDS, the table USR contains the list of friends, while INF the email, an attribute which applies not only to users but to other entities such as retailers. The translation for DDS requires, therefore, an additional join.

Not only the queries play a role in efficiency but also the characteristics of each schemata. For instance, VP tables are of smaller size. While one might think this could be an advantage in terms of query execution, the VP tables could, due to their size, result in a low number of partitions thus limiting the parallelization potential. In contrast, TT and WPT consist of a single large table. In particular, WPT can process all triple patterns with the same variable on the subject without employing joins. DDS tables are also large and for queries accessing a single DDS table, the algebra-tree would be the same as for WPT, being the size of the table the only difference. Moreover, WPT and DDS can have columns with complex data types such as lists, which are exploded with *lateral view explode*. The advantage of this operator is that this can be applied to multiple complex columns. In this case, a cross-product between the exploded columns is automatically executed.

Thus, the question arises to which extent the resulting Spark SQL translation, its execution plan, the number and sizes of tables, and complex operations such as *lateral view explode* or *contains()* affect the performance of the query execution. Our experiments are designed to provide insights into these aspects.

Query construction. To achieve a fair comparability of the different schemata, we consistently generate the algebra tree according to some basic rules. First, operations which access constants in the triple patterns are pushed to the bottom of the tree to reduce intermediate results. Next, a basic join ordering based on selectivity is carried out. The selectivity values are obtained in the loading phase of the RDF-graph by considering triples in isolation[7]. Finally, the algebra tree is visited in a depth-first way to build the Spark SQL expression which is submitted to Spark SQL's optimizer, Catalyst.

Catalyst performs different logical and physical optimizations to the query plans based on heuristics and statistics information, e.g. it chooses a specific join strategy. Spark SQL supports shuffle-hash-joins, sort-merge joins, and broadcast-joins. In a join, matching tuples typically reside on different nodes. When one of the join operands, i.e. a table, is small, this might be entirely sent to all nodes (broadcast join). Otherwise, a hash-based partitioner can be applied on the join columns of the respective tables, which shuffles the tuples, thus placing potential join partners in the same nodes. As we analyze relational schemata formed by tables of different sizes, broadcast joins might be an option for VP and DDS, while for TT and WPT this is rather unlikely. However, a broadcast join implies an overhead for reading the table statistics (size), collecting all tuples of a table and transmitting it to all nodes. This might not pay when the other join operand is also small, because only a small number of executors will be used thus resulting in a small degree of parallelism. Therefore, joining larger tables using a hash or sort-merge join after shuffling both tables might be beneficial as well, because of a higher degree of parallelism.

[7]Better selectivity measures can be obtained based on *characteristic sets* [17]. However as we are not interested in absolute execution times our approach is assumed to be sufficient for a relative comparison of the different schemata.

4 EVALUATION RESULTS

We perform our tests on a small cluster of 10 machines connected via Gigabit Ethernet connection. Each machine is equipped with 32GB of memory, 4TB of disk space and with a 6 Core Intel Xeon E5-2420 processor. The cluster runs Cloudera CDH 5.10.0 with Spark 2.2 on Ubuntu 14 and consists of one master and nine workers. Yarn is the scheduler, which in total uses 198GB and 108 virtual cores. A Spark partition size is equal to the size of an HDFS-block (128MB).

For evaluation we use the datasets WatDiv [4] and Yago [23]. The dataset provided by the Waterloo SPARQL Diversity Test Suite 7 has approx. 109 Million triples and 86 predicates. Our set of 88 queries is derived from 20 templates given by the *WatDiv basic query set*. The number of triple patterns in the queries ranges from 2 to 10. These queries are of varying shape and selectivity in order to model different scenarios. They are grouped into the subsets C (complex-shaped), F (snowflake-shaped), L (linear-shaped), and S (star-shaped) queries. Our second dataset, Yago (version 2s 2.5.3), is a semantic knowledge base derived from Wikipedia, WordNet and GeoNames, i.e. a real-world dataset with approx. 220 Million triples and 104 predicates. Our Yago query set of 15 queries is the same used in S2RDF [25], where the number of triple patterns ranges from 3 to 13. Our schema designs for each dataset are TT, VP, WPT, and DDS. For each of the different designs the memory required by their corresponding tables in HDFS is as follows:

	TT	**VP**	**WPT**	**DDS**
WatDiv	2.04GB	38.8KB-679MB	1.04GB	4.12MB-637MB
Yago	7.12GB	20.9KB-1.3GB	3.61GB	48.5KB-1.8GB

It is important to notice that TT and WPT consist of a single large table, which means that they cannot take advantage of broadcast-joins when Spark's default threshold of 10 MB is used. However, this threshold can be adjusted. While for TT, VP and WPT the design of the tables is independent from the meaning of the RDF data, our design of the DDS is inspired by [20] and based on the corresponding data model specifications[8]. Thus, for WatDiv we derived 9 tables with numbers of columns ranging from 2 to 37 and for Yago 11 tables with numbers of columns ranging from 2 to 25. The following table shows the sparsity of WPT compared to that of our DDS schema. Let sparsity (S) be the ratio between the *null* values and the table size ($|Rows| \times |Cols|$).

WPT	#Tabl	S	**DDS**	#Tabl	AVG(S)	STD(S)
Watdiv	1	0.937		9	0.427	0.222
Yago	1	0.975		11	0.554	0.267

As expected the WPT which has all properties as columns has a very high sparsity (the maximum is 1). Our DDS design achieves a more compact representation. However, we shall not forget that Parquet is used as data format, thus it efficiently encodes *null* values. In our experiments we conduct the evaluation on our relational schemata taking into account the following aspects.

(1) First, we consider the performance of the benchmark queries when one query at a time is submitted and executed in a single

[8]Watdiv: https://dsg.uwaterloo.ca/watdiv/#dataset.
Yago: https://github.com/yago-naga/yago3/blob/master/schema/relations.ttl.

Spark application, thus representing a cold-start scenario in which memory is free. We refer to this as to the *cold-one-by-one* mode. However, executing a set of queries in the same Spark application is a more realistic usage of the platform and closer to real SPARQL endpoints. In this case one query can then take advantage of in-memory data left by a previously executed query. For this reason we consider also a *warm-random* case. The numbers reported for each query group and relational schema are for the cold-one-by-one case the average of 5 execution times. For the warm-random mode, each query set is evaluated 50 times in random orders. We prune execution times outliers using the interquartile range (IQR) to set an upper fence (3rd quartile + IQR * 1.5).

(2) Secondly, as another dimension for our analysis, we distinguish between experiments with broadcast-join disabled and enabled. As some VP and DDS potentially contain small relations in contrast to for example TP and WPT, broadcast-joins might considerably change their execution times. This aspect will allow us to better understand the influence of the table sizes.

(3) As a third aspect, we also investigate the impact of different physical partitioning strategies offered by Spark. By default Spark builds partitions by organizing the tables' data in HDFS without knowing its actual content. Partitioning based on a key is supported by Spark. Using the RDF subject as that key could intuitively be a means to avoid shuffling of partitions for certain queries.

Broadcast-Join disabled. In our first group of experiments we disable broadcast-join and run the benchmark queries for the cold-one-by-one and warm-random modes. The results are presented in Figure 3[9]. The first observation is that the performance of WPT compared to that of other schemata is better for the majority of query types. In the *cold-one-by-one* case, WPT shows the best performance except for the Watdiv L and S query types, although it is not significantly worse. For *warm-random*, which is the realistic case, it shows the best performance for all query types. The same trend is showed for Yago queries. There is a reason for this. WPT has the advantage of having all properties for each subject in a single row and each row is entirely located in one partition. This is particularly beneficial for joins on the subject. Hence for WPT, finding all information related to a subject is as simple as projecting the corresponding columns. This is exploited even more in our experiments because the benchmarks are biased towards star-shaped queries. As a matter of fact, 88% of the triple patterns in Watdiv are involved in star-shaped joins. On average there are 1.58 *on-subject* joins per query. In the Yago query set this is a bit less extreme: 77% involved triple patterns and an average of 1.8 subject-based joins per query. Therefore, in both cases this kind of join dominates. Interestingly, TT which is also a single-distributed table, performs the worst for all cases. In TT, rows with the same subject might instead be distributed in several nodes. Therefore, shuffling is required to bring them together. Looking at the other side of the spectrum in terms of table size we have the VP schemata. This approach has the downside of having to read as many tables as there are predicates in a SPARQL query. Therefore, a query translated for VP most probably requires a larger number of Spark SQL joins than a WPT which can subsume all triple patterns with the same subject variable. A larger number of joins leads to a larger number of stages, which at the same time introduces more dependencies between them and reduces the parallelization potential. Figure 3 shows that performance of VP comes closer to that of WPT when a group of queries contains fewer predicate constants. Queries of the group C have 8 triple patterns on average, the F queries 6.6, L 2.33 and S 3.33. This explains why for L and S queries VP achieves a better performance than WPT in the *cold-one-by-one* case. Since WPT can be kept longer in-memory (all queries access the same table), this is no longer the case for the *warm-random* case.

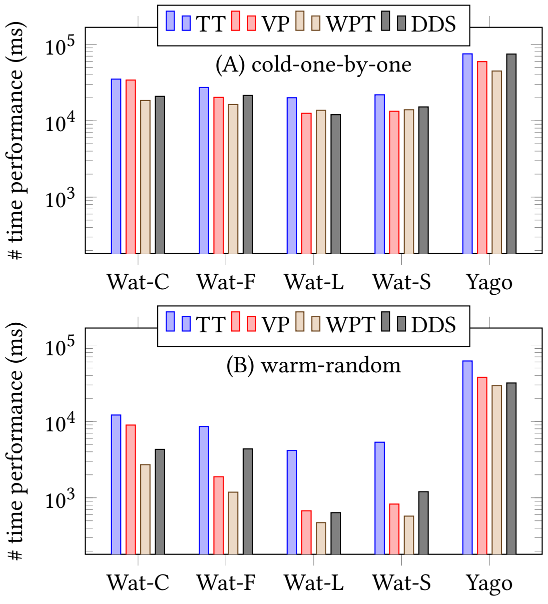

Figure 3: Performance evaluation for Watdiv and Yago when broadcast-join is disabled. (A) *cold-one-by-one* mode. (B) *warm-random* mode.

The performance of the DDS schemata lies somewhere in between. However, we can notice that for Watdiv-L queries DDS shows a competitive performance. These queries have three triple patterns two of which can be found in the same table (`USER`). In DDS this can be achieved with a single join between the `USER` and `PRODUCTS` tables. This results in fewer stages and faster execution than VP, which requires accessing three tables and two joins.

Broadcast-Join enabled. WPT shows a surprisingly good performance when broadcast-joins are disabled, but schemata with smaller tables could not benefit from broadcast-joins. Therefore, we carried out experiments with this feature enabled and using two threshold values, 10MB (default) and 1MB, to assess whether this brings a benefit for VP and DDS. The results can be found in Figure 4. As the results show broadcast-joins are not bringing significant benefits to VP, nor DDS. Not only is the default threshold worsening the performance in nearly all cases, but also the threshold set at 1MB is only closely behind VP_{no}[10]. Looking at our results more closely, we realized that even queries which didn't trigger broadcast-joins required longer execution times when the feature

[9]The vertical axes of all plots are showed on a semi-logarithmic scale.

[10]For DDS no results are showed for the 1MB, because its smallest table has 4.12MB.

was enabled than when it was disabled. This is due to the overhead required by the query engine to obtain the table size information from the Hive statistics.

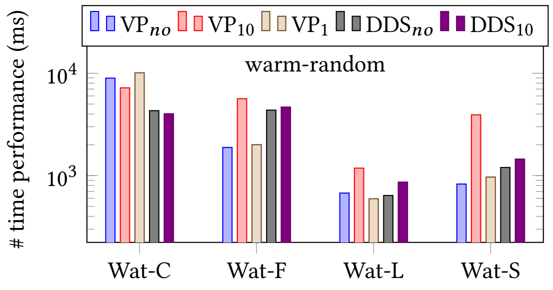

Figure 4: Performance evaluation for Watdiv when broadcast-join is enabled with default threshold of 10MB and 1MB. VP$_{no}$, DDS$_{no}$ represent times with this feature disabled. *Warm-random* mode.

Hence, if broadcast-joins are not bringing a benefit, the single wide property table continues to show the best performance.

Impact of Spark's partition by subject. Since our queries show a bias towards star-shaped queries (joins on subjects), an interesting aspect to investigate is whether a partitioning scheme based on the (RDF) subject speeds up the execution. This is typically done in Spark to reduce the overhead of shuffling. Thus, in this experiment, we first force *repartition(subject)* to create the partitions and use the resulting DataFrame(s) as the input for the query engine. Since repartition also requires shuffling, the number of partitions generated is equal to *spark.sql.shuffle.partitions* which is set at a default value of 200. The following table shows the impact with respect to the default partitioning scheme (showed in Figure 3(B)). The repartitioning time is excluded. In this table negative values (in red) denote a negative impact in the performance. The results show that partitioning by subject brings a significant benefit only for DDS, whose performance is only slightly worse than WPT with the default partitioning scheme. While having a partition for each subject helps to speed up joins on the subject column, other kinds of joins are strongly penalized and forced to deal with a much larger number of partitions of much smaller size (200).

	TT	**VP**	**WPT**	**DDS**
Wat-C	-9.1%	+0.8%	-1.8%	+35.2%
Wat-F	-8.2%	-7.6%	-9.3%	+32.6%
Wat-L	+2.8%	-18.1%	-7.4%	+62%
Wat-S	-1.6%	-15.2%	+4.6%	+18.1%
Yago	-0.2%	-5.9%	-6.7%	+1.1%

Figure 5: Impact of partition by subject and default broadcast threshold

5 CONCLUSION

In this paper, we evaluate relational schemata for SPARQL query evaluation. While this has already been extensively discussed for single machines [1, 26], a general discussion for computer clusters was still missing. We provide such an analysis for distributed in-memory approaches using Spark SQL. The overall benchmark evaluation in the current paper suggests that using a simple standard one table approach, i.e. WPT, seems to be competitive to more elaborate designs as, for example, the domain-dependent-schema DDS. This might become an interesting important issue when highly unstructured RDF graphs must be processed for which deriving a DDS or emergent schema becomes difficult and costly. Moreover, at the schema design level, WPT is nearly independent from the evolution of the respective underlying RDF graph; for example, only adding a new column is required to accommodate new RDF triples with previously unseen properties. Based on the analysis in the current paper we can attribute the surprisingly good performance of WPT to the following. First, as WPT collects all the data in one table, we have a guarantee to execute a number of joins which is as small as possible. This reduces number of stages and their dependencies. In addition, the relative huge size of WPT implies a large enough number of partitions to provide work for as many nodes of the cluster as possible. Both features are among the prerequisites for high parallelism during distributed query evaluation.

REFERENCES

[1] D. J. Abadi et al. Scalable semantic web data management using vertical partitioning. In *Proc. VLDB*, 2007.
[2] I. Abdelaziz et al. Combining vertex-centric graph processing with sparql for large-scale rdf data analytics. *IEEE TPDS*, 2017.
[3] A. Abele et al. Linking open data cloud diagram 2017. http://lod-cloud.net/, 2017.
[4] G. Aluç et al. Diversified stress testing of rdf data management systems. In *Proc. ISWC*, 2014.
[5] P. A. Boncz et al. Advances in large-scale RDF data management. In *Proc. Linked Open Data - Creating Knowledge Out of Interlinked Data - Results of the LOD2 Project*. 2014.
[6] M. A. Bornea et al. Building an efficient RDF store over a relational database. In *Proc. SIGMOD*, 2013.
[7] J. Broekstra et al. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proc. ISWC*, 2002.
[8] M. Cossu et al. Prost: Distributed execution of sparql queries using mixed partitioning strategies. In *Proc. EDBT*, 2018.
[9] M. Färber et al. Linked Data Quality of DBpedia, Freebase, OpenCyc, Wikidata, and YAGO. *Semantic Web Journal*, 2018.
[10] D. Graux et al. Sparqlgx: Efficient distributed evaluation of sparql with apache spark. In *Proc. ISWC*, 2016.
[11] S. Gurajada et al. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proc. SIGMOD*, 2014.
[12] A. Harth et al. Yars2: A federated repository for querying graph structured data from the web. In *The Semantic Web*. 2007.
[13] Z. Kaoudi and I. Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1), 2015.
[14] A. Madkour et al. Sparti: Scalable rdf data management using query-centric semantic partitioning. In *Proc. SBD*, 2018.
[15] A. Madkour et al. WORQ: workload-driven RDF query processing. In *Proc. ISWC*, 2018.
[16] T. Neumann et al. Rdf-3x: a risc-style engine for rdf. *Proc. VLDB*, 2008.
[17] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Proc. ICDE*, 2011.
[18] Z. Pan and J. Heflin. DLDB: Extending relational databases to support semantic web queries. In *Proc. PSSS1 - Practical and Scalable Semantic Systems*, 2003.
[19] M. Pham and P. A. Boncz. Exploiting emergent schemas to make RDF systems more efficient. In *Proc. ISWC*, 2016.
[20] M.-D. Pham et al. Deriving an emergent relational schema from rdf data. In *Proc. WWW*, 2015.
[21] A. Potter et al. Distributed RDF query answering with dynamic data exchange. In *Proc. of ISWC*, 2016.
[22] R. Punnoose et al. Rya: a scalable rdf triple store for the clouds. In *Proc. IWCI*, 2012.
[23] T. Rebele et al. YAGO: A multilingual knowledge base from wikipedia, wordnet, and geonames.
[24] A. Schätzle et al. Sempala: interactive sparql query processing on hadoop. In *Proc. ISWC*, 2014.
[25] A. Schätzle et al. S2rdf: Rdf querying with sparql on spark. *Proc. VLDB*, 2016.
[26] L. Sidirourgos et al. Column-store support for rdf data management: Not all swans are white. *Proc. VLDB Endow.*, 2008.
[27] K. Wilkinson. Jena property table implementation. In *Proc. SSWKBS*, 2006.
[28] M. Wylot et al. RDF data storage and query processing schemes: A survey. *ACM Comput. Surv.*, 51(4), 2018.