K.-U. Sattler et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2021),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 237

Flexible data partitioning schemes for parallel merge joins in semantic web queries

Benjamin Warnke [1], Muhammad Waqas Rehan [2], Stefan Fischer [2], Sven Groppe [1]

BTW-2021
RESULTS
REPRODUCED

Abstract:

Semantic Web technologies are enabling large amounts of data to be preprocessed and stored on the web to be queried efficiently later. The key technology in this topic is the triple store storing all information in the form of triples (subject, predicate and object). Depending on the triple patterns used within the queries, varying graph structures can be observed in the datasets. Currently, such properties are only exploited implicitly during join optimization in the form of histograms or similar technologies. Towards a new paradigm for explicitly exploiting graph structures in the datasets, this paper proposes a new flexible partitioning scheme at runtime. To do so, we experimented with partitioning schemes, that can be selected depending on the actual data access within a given query in order to improve query performance. The experimental results show that the proposed flexible data partitioning schemes are faster, up to a factor of 12.65 in comparison to no partitioning.

Keywords: Triple store; Partitioning; Parallel Join

1 Motivation

The Semantic Web makes huge datasets [HHK19, TWS20] available that may be queried for information processing and gathering afterwards. These datasets are continuously growing in size, either by users adding more information, or by automated data sources continuously providing new data. According to a survey paper about Semantic Web query languages [Ba05], SPARQL [SH13] is the most important RDF query language.

Some of these datasets contain more than a billion triples [HHK19]. Such big datasets are often created, maintained and used by many users. Nevertheless the users of such a Semantic Web database system desire a fast response time. Both aspects alone are challenging for a database system. Together the pressure to create fast and resource friendly query execution plans is increasing even more.

[1] Universität zu Lübeck, Institute of Information Systems, {warnke,groppe}@ifis.uni-luebeck.de
[2] Universität zu Lübeck, Institute of Telematics, {rehan,fischer}@itm.uni-luebeck.de

doi:10.18420/btw2021-12

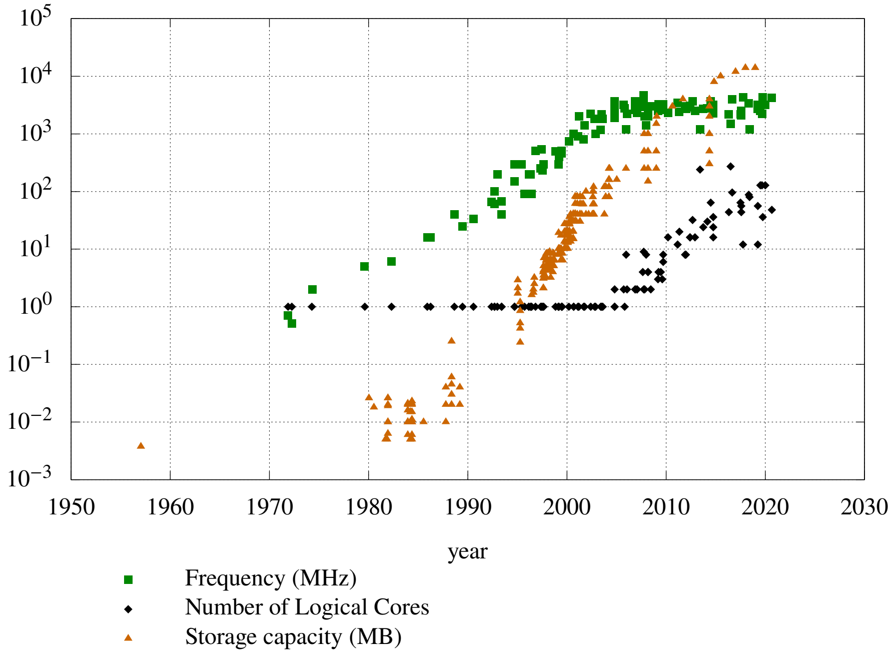


Fig. 1: CPU performance and storage size over the last 70 years. CPU-data modified from github [Ru20]. Storage-data taken from Wikipedia [Ha18]. This figure is not exhaustive and contains only data, which is available to the public.

To answer many queries on big datasets, the database system may take most out of the available hardware. The chart in Fig. 1 makes it obvious that the speed of a single processor core stagnates over the last 10 years. For further performance boosts, information technology companies have been developing CPUs with an increasing number of cores. The multi-core systems require massive parallelization for efficiently utilizing the hardware capacity and to satisfy the increasing demand for higher data processing capabilities. At the same time, the available persistent storage is increasing, too. The additional storage can be used to support massive parallelization by providing multiple variations of the original data. It opens up new challenges and opportunities for research in the context of parallel query processing.

Previous research [Ar11] has shown that various features of SPARQL such as ”projection”, ”basic triple pattern”, ”join”, ”optional join” and ”filter” are used frequently. All of these operators can be evaluated in parallel. This is advantageous because they can be evaluated even without communication between the threads, if the data is partitioned according to suitable columns. Since evaluating ”projection” and ”filter” in parallel is trivial, the remainder of this paper will focus on the join operator.

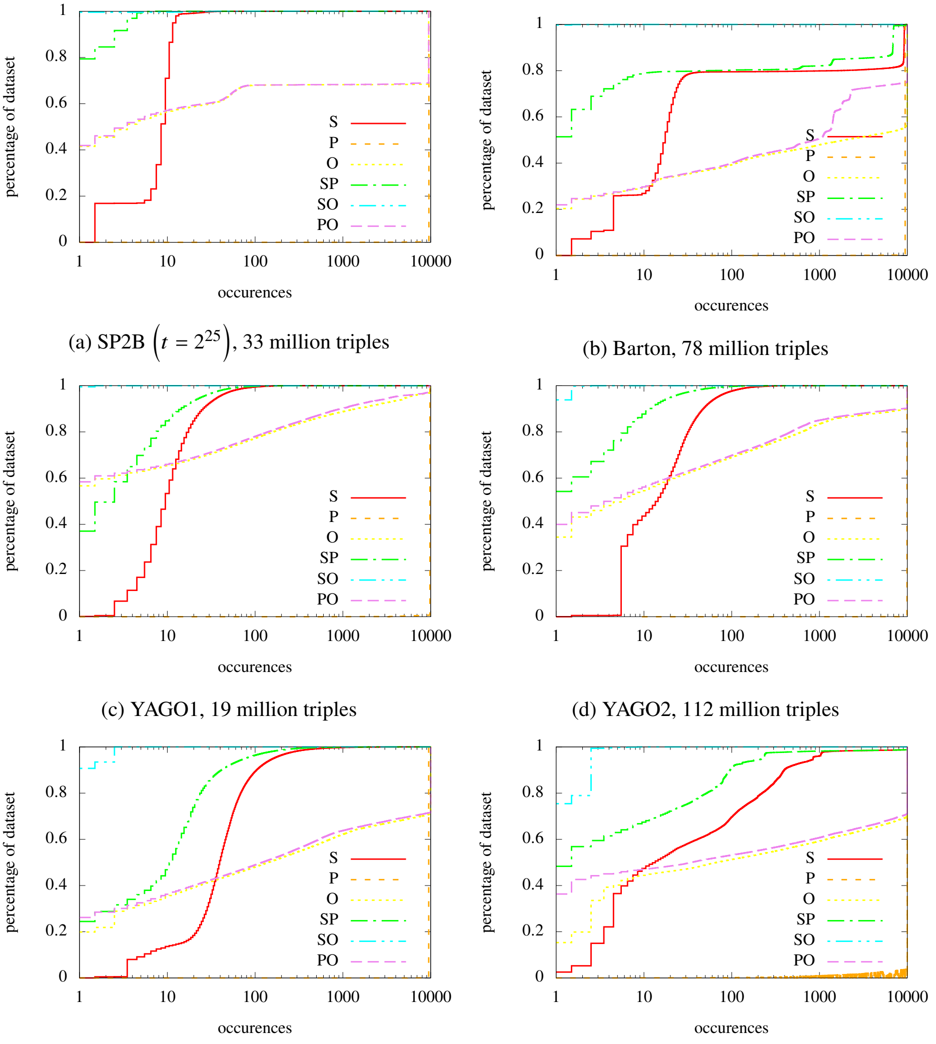
For the most part, join operators are executed using merge and hash join implementations.

Because merge joins may achieve a much higher performance than other implementations, whenever data is already sorted according to the join columns based on some previous operations or by accessing appropriate indices like B^{+}-trees. Therefore we will focus on merge joins.

To improve data parallelism, we have analyzed the fundamental structure of data in the triple stores and the methodology of accessing it. In SPARQL, access to the stored triples is formulated in terms of triple patterns. In each triple pattern, all three components of a triple need to be specified either as a constant specifying constraints on matched triples or as a named variable for storing queried triple components.

We assume that an entirely different number of result rows can be expected based on the triple pattern being processed. On the one hand, when only the predicate is a constant, then we assume that the number of result rows are very high. In other words, a small number of predicates is required to retrieve the whole data. On the other hand, when only the predicate is a variable, then we assume that there are almost always only a few results. Similar assumptions are reasonable for other combinations of variable and constant occurrences in the triple patterns. Therefore we analyzed a synthetic dataset from the SP2B benchmark [Sc09] as well as some real world datasets , i.e. BTC2019 [HHK19], Barton[Ab07], YAGO1 [SKW07], YAGO2 [Ho13] and YAGO2s [BKS13], as shown in Fig. 2.

This is interesting, because data parallelism works best if there is a lot of data. That means, if only the predicate is a constant, then data partitioning may gain huge improvements. But, when only the predicate is a variable, then the improvement - if any - is small. If we consider all possible triple patterns, we come to the conclusion, that a constant in the subject position leads to few results, so that the partitioning may hardly bring any advantages. Every other triple pattern yields more results such that a benefit due to data parallelism may be likely.



(a) SP2B $\left(t = 2^{25}\right)$, 33 million triples

(b) Barton, 78 million triples

(c) YAGO1, 19 million triples

(d) YAGO2, 112 million triples

(e) YAGO2s, 171 million triples

(f) BTC2019, 256 million triples

Fig. 2: The figure shows the cumulative distribution function $f(X < x)$. The X axis shows the number of triples which share the same value at the columns specified by legend entry. The Y axis represents the percentage of the triples in the whole triple store which share their value with at most X triples. The names of the graphs consist of the constant values of the triple pattern. For example the graph P shows the relation for triple patterns of type *?s <p> ?o*, where the predicate is a constant.

The main contributions of this article are:

- Introducing a flexible partitioning scheme of the data, which allows to select the number of partitions at runtime. Therefore we store multiple different partitioning schemes at runtime.
- Analyzing the number of partitions yielding the best performance for parallel merge joins depending on the number of triple patterns and the amount of stored data.
- We propose a function to predict the optimal number of partitions depending on the data in the store and the query.
- Comparing exhaustively different parallelization strategies.
- Procuring a performance improvement up to a factor of 12.65 in comparison to no partitioning.

2 Related work

There are several ways for improving the performance of joins in the context of triple store access. The most important among them are discussed below.

2.1 Additional indices

The indices RDF3X [NW08, NW10] and Hexastore [WKB08] are often used in triple store implementations. Both index variants use a dictionary for mapping the actual values to internal numeric ids.

RDF3X [NW08, NW10] uses 6 indices, which consist of all possible collation orders of S, P and O, which are SPO, SOP, PSO, POS, OSP and OPS. The SPO index is the short form for the collation order, where the triples are first ordered by their subject, then by the predicate and finally by the object. The other collation orders only differ in which triple component is ordered first, second and last. Due to performance considerations, the ordering is applied based on the integer ids instead of the values. Additionally the ordering allows to use very efficient compression. The indices themselves are stored as B^+-trees. Because the data is ordered, each triple pattern in the query can be translated to a range scan in the B^+-tree.

Hexastore [WKB08] uses 6 indices with all the collation orders. Instead of a B^+-tree, Hexastore [WKB08] uses a multi layer linking structure. Taking the SPO-index as an example, each subject points to a list of predicates, which in turn points to a list of objects. Such a list of objects can be shared with the PSO index, because they are exactly the same. Each of these lists is ordered. After the first triple is found, both indices (RDF3X and

Hexastore) allow a simple iteration over an ordered list. Both triple stores support aggressive usage of the merge join on already sorted data retrieved from its indices [NW10, WKB08].

In contrast to the above indices, which are both indexing triples, another contribution [NC20] proposes to index sub-graphs. The Triag-index [NC20] searches for triangular patterns in the graph, and stores them in a separate index. It allows a query optimizer to replace multiple consecutive joins by a range scan in their index structure. Especially in the context of ontologies it may yield high performance.

2.2 Parallel SPARQL processing

In this category, one approach [BK20] partitions the triples vertically, so that all predicates are stored in separate virtual tables. Within these tables the subjects and objects are stored in the form of independent arrays. The connection between these tables and arrays is established by vectors of pointers. The authors signify a higher storage efficiency of their approach - especially in a distributed context where it is important to find the node containing the required data.

Merge joins have a huge performance advantage over hash joins. Therefore, another approach [AKN12] partitions and orders the data on demand, so that massively parallel merge joins can be used everywhere. Even if this paper [AKN12] is about database systems in general, its results apply to Semantic Web database systems as well. Due to the sort operations at runtime, this approach requires a lot of available memory. Additionally this sorting step requires more computation time, compared to our approach, which can directly read ordered and partitioned data from the triple store.

In another paper [GG11], partitioning threads are used to horizontally partition the input data of the join operator. In this way, any intermediate result can be partitioned into any number of partitions at runtime, but at expense of the runtime overhead for partitioning. If the input for merge joins comes directly from the triple store, then partitions can be directly accessed based on the ranges in B+-trees. As a range is determined according to the histogram of the corresponding triple pattern, the partition sizes of the triple pattern to be joined may be unbalanced. This article additionally evaluates the performance gain using the pipeline parallelism. The main disadvantages of operator-based parallelism are the queues between the operators, which on the one hand require storage space and on the other hand entail thread safety by locking.

Our approach is novel in this context because it avoids the usage of partitioning threads and queues completely by employing multiple materialized balanced partitions which are flexibly chosen at query optimization time.

2.3 Distributed SPARQL processing

Our contribution focuses on local database systems. Nevertheless there exist several strategies in the distributed context, which provide different approaches to partition triple data. To the best of our knowledge, there is no distributed implementation with the same functionality as proposed in our contribution.

On top of the parallel SPARQL processing, which parallelizes on a single node, data can be distributed to multiple nodes for achieving a higher degree of parallelization. A prerequisite is obviously the presence of several nodes, as well as a fast connection between them for maintaining high performance. Similar to node local partitioning, the triples need to be assigned to a node. Typically such algorithms assign a node to each triple using a deterministic algorithm. The key task of all these algorithms is to yield a uniform distribution among all nodes.

In another paper [Ha16], the triples are distributed by a hash function on the subject. It may allow to process many joins locally on the corresponding nodes, which may reduce the network communication cost. The independent joins are similar to partitioning in a node local context. If a join can not be evaluated locally on a node, then hash joins may be heavily used for exploiting the advantage of the hash based distribution.

There are also approaches [Ha07], which allow to query from multiple data sources simultaneously. Therefore, they attach a context - the original data source - to each triple such that their database system effectively stores quads. Similar to other hash based approaches [Ha16], the hash function only uses one component of the triple for its partitioning. Hash based distribution can also be used for map-reduce based database systems [Pa13]. All of the above distribution strategies use only one component of a triple for assigning a node.

However, the following strategies use all components of a triple to calculate the assigned node. In the approach [JSL20], the connectivity between triples is calculated, and close connected triples - called ”molecules” - are assigned to the same node. Apart from the huge overhead during initialization, it may allow to calculate many joins independently and locally at a node. Another approach [Ze13] uses a completely different encoding. Similar to a previous strategy [JSL20], data is distributed such that related data is stored close to each other. In this case, the data is stored in the form of of adjacency-lists. The advantage is, that each node knows, which other nodes have related data. Therefore, distributed joins can explicitly access the relevant nodes, which reduces the communication overhead.

The key idea of all data distribution algorithms is to reduce or even remove the communication overhead as much as possible. As long as the data can fit in the memory of a single node, evaluating local queries is often much faster. In the remainder of this document, only node-local improvements in query evaluation are considered.

3 Our approach - flexible data partitioning

To use data parallelism, the data must be distributed over an arbitrary number of partitions. The key problem is to optimize the number of partitions. If we use too many partitions, then the overhead is larger than the benefit. If we use too few partitions, then both a fair data distribution and resource utilization (in terms of CPU core usage) is not possible.

If only merge joins are used, then the actual computation is so fast that the overhead caused by live partitioning can not be effectively compensated [GG11]. Hence, we propose to materialize different partitions already in the indices as parallel inputs to our merge join threads without introducing any computational overhead.

Fig. 2 shows that each triple pattern yields drastically different numbers of triples. Therefore, it is not possible to pick just a single number and use it for establishing the number of partitions. As a solution, we propose to use several different partitioning schemes - for each index - at the same time. This allows very flexible data storage depending on the expected data properties. Additionally, we are able to choose the used number of partitions during query optimization time. It enables a much more fine grained control over the effective parallelism. Fig. 3 shows a structural example of our proposed triple store implementation. It is important that each partitioning scheme can choose both: a different hash function and another number of partitions. We want to explicitly store partitions according to these different partitioning schemes to avoid the partitioning overhead at query runtime.

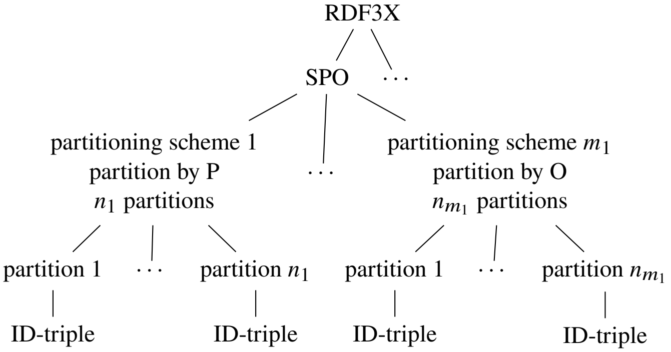


Fig. 3: Structure of database system implementation

Each partitioning scheme occupies persistent storage space. The more schemes are defined, the more memory is required. Therefore, we propose to store only a few of the most effective schemes. Unfortunately, it may introduce another type of problem requiring different numbers of partitions to join with each other.

To enable efficient implementations, we restrict ourselves to the numbers of partitions with a power of two. The efficiency comes from the properties of the modulo operator. If the number of partitions is halved, then exactly two partitions need to merge to a single one -

without touching any other partition. It allows to use much less locking, because less threads share common locks, and therefore increases the speed. In Sect. 4 we evaluate where to use which partitioning scheme.

4 Evaluation

To gain insight into how the number or presence of an additional partitioning layer affects the database system performance, there are many performance aspects to consider. In this paper, we focus on parallel main-memory query evaluation of merge joins.

4.1 Experimental setup

We use two different Benchmark Systems to verify the validity of our findings independently of the hardware architecture. That enables us to compare an server CPU with a recently introduced desktop CPU of the current generation.

The first machine (M1) is a dual socket machine using Intel Xeon E5-2620 v3 CPUs with a clock rate of 2.4GHz. In our experiments, hyper threading is enabled such that there are 24 hardware supported threads. Each socket is assigned with 16GB memory, such that a total of 32GB memory is available. The database systems either use gcc 5.4 or Java 1.8.265 in the server-edition.

The other machine (M2) is a single socket machine using an Intel i9-10900K CPU with a clock rate of 4.9 GHz. On this machine, hyper threading is enabled as well, such that there are 20 hardware supported threads. This machine has 64GB of RAM installed. It uses Java 14.0.2 in the server-edition.

4.2 Implementation details

Our database system LUPOSDATE3000[3] is a rewrite of LUPOSDATE [Gr11]. The old LUPOSDATE [Gr11] is implemented in Java. The new LUPOSDATE3000 database system is implemented in Kotlin programming language in order to support different targets like the JVM, JavaScript and native binaries for desktop, server, web and mobile environments. Currently, Kotlin-JVM-target is the fastest, therefore all benchmarks are evaluated using Java runtime. LUPOSDATE3000 uses a dictionary to map all values to integer IDs. These IDs are then stored in an index similar to RDF3X using all 6 collation orders. During query evaluation, LUPOSDATE3000 uses both column and row iterators, preferring the column iterators where applicable. In our experiments, the hash function used for partitioning only

[3] https://github.com/luposdate3000/luposdate3000.git

performs the modulo operator to the integer IDs in the store. We yield uniform partition sizes with these hash functions.

We choose Apache Jena[4], blaze-graph[5] and virtuoso[6] as competitive database systems, because they are the most used open source RDF database systems according to an RDF database ranking [DB20].

Jena is an RDF store written in Java. Since we use Kotlin-JVM-target, it allows to compare two different database systems using the same Java runtime environment. The triples are stored in B^+-trees.

Blaze-graph is written in Java. The indexes are stored in B^+-trees which are influenced by Google's BigTable system.

Virtuoso is written in c++. In our tests, we only use the RDF interface of virtuoso. It allows the comparison to a compiled database system, and verifies our expectation that garbage collected languages are not inherently slow.

4.3 Datasets and queries

In order to facilitate clear indications about how exactly partitioning influences the execution times, we use simple queries as shown in Fig. 4. The query Q1 is used as a template for the benchmarks which use up to 16 consecutive merge joins. Q2 is the only query, which enforces a hash join, all other queries can be evaluated using only merge joins.

```
PREFIX b: <http://benchmark.com/>
SELECT *
WHERE {
  ?s b:p0 ?o0 .
  ?s b:p1 ?o1 .
  ?s b:p2 ?o2 .
}
```

(a) Query Q1

```
PREFIX b: <http://benchmark.com/>
SELECT *
WHERE {
  ?s b:p0 ?o0 .
  ?s b:p1 ?o1 .
  ?o1 b:p2 ?o2 .
}
```

(b) Query Q2

Fig. 4: SPARQL queries used for the benchmarks.

We create synthetic data such that it matches our requirements of selectivity and output size. Because our proposed changes are not related to join order optimization, we do not want any side effects of the applied optimizer. Therefore our generated triple structure is the same for every triple pattern in our query.

[4] Version 3.14.0

[5] Version 2.1.6

[6] git://github.com/openlink/virtuoso-opensource.git, Revision 840b468fc400a254eab0eb20f1afde6ca3c2220d

We label all our graphs with result rows instead of the usually used number of triples. In this way, we can enforce a uniform workload across all the used threads. Furthermore, a low selectivity combined with a low number of input triples would yield too few result rows without notice. A few results would not be uniformly distributed to the threads, which in turn decreases the benefit achieved from multiple threads.

4.4 Query optimizer

Due to the new partitioning scheme, there are lots of possibilities how to execute a simple query. The query Q1 shown in Fig. 4a can be evaluated using two merge joins. We want to compare the performance of 1, 2, 4, 8 and 16 partitions for each of the join operators as well as for each of the triple store iterators. We remove the options, where a merge join requires to change the partitioning of both of its inputs during the runtime, therefore we get $2^2 \cdot 4^3 = 256$ different operator graphs.

Using the same number of partitions everywhere in the operator graphs yields the best results. We have verified it using the computer configurations of M1 and M2. Additionally in our experiments, we have considered different synthetic datasets, too, with either uniform or non uniform data distributions. For the non uniform synthetic datasets, we increased the differences such that one triple pattern yields up to 128 times more input than the others. This has changed the total query evaluation time, but not the optimal partitioning ranking in the operator graph.

Depending on where exactly the number of partitions changes in the operator graph, the evaluation speed is not that much lower compared to operator graphs using only one partition count. It means that our approach does not need to assign the same number of partitions to every collation order.

The query Q2 shown in Fig. 4b joins on two different variables, which means, that we can not use two merge joins. It yields to some more options for the optimizer especially which number of partitions on which variable are used for the second join. One option is, to merge the partitions after the first join, and then change the partitioning of that result just in time for the second join. The other option is to pass through the partitioning as it is, which means, that the second join is not partitioned by its join variable. In this case the second join must read in the whole not partitioned or united input from the other side, to still produce valid results. Our measurements show, that in this case it is the fastest to use the second option, which is passing through data which is not partitioned by a join variable.

4.5 Benchmarks

This section is divided into two parts. The first part deals with the evaluation of a micro-benchmark with an explicit focus on the performance of the merge join operator. Therefore, several database system functions are disabled or bypassed. The changes made are:

- The operator graphs, including the partitions to use, are hard coded, in order to investigate their effects on query performance.
- The benchmark code is included in the LUPOSDATE3000 binary to avoid HTTP-interface overhead.
- The result is only calculated as a row of integer IDs. The required dictionary lookups for converting those IDs to Strings are not included in the benchmarks.

This benchmark is intended to show the effect on query evaluation, in case both data structure and data size are changed. Since, it is not easy to apply the above changes to other existing database systems, this part is only evaluated within LUPOSDATE3000.

In the second part, the same queries are evaluated on multiple database system implementations. Here in this part, all database system features are enabled, and the HTTP-SPARQL endpoints are used.

4.5.1 Micro benchmark

To focus on the performance effects of partitioning during query processing of join operator chains, a synthetic micro-benchmark is used so that the input data has the desired input patterns. During the tests, the following properties have an impact on the query runtime:

- Number of input rows and number of result rows: Larger numbers of rows yield higher speedup because the sequential query initialization phase needs less time compared to the total computation time.
- Selectivity of the joins: When more rows are filtered away, the next join operator has less work to do, resulting in faster query processing.
- Number of CPU-cores: Due to the in-memory benchmark-setup, all experiments are both memory and CPU bounded. As long as the data is evenly distributed, it does not make any sense to employ more partitions than CPU cores.
- Number or partitions: Currently LUPOSDATE3000 parallelizes its query evaluation based on partitions only. As a result, maximum speedup corresponds to the number of partitions. Depending on the other parameters the maximum speedup may not be reached.

- Number of consecutive joins: More consecutive joins on the same join columns increase the throughput because there is no need to serialize or cache intermediate results.

Since we are interested in comparing the effects of different selectivities within the join operators, we have generated multiple synthetic datasets. For generation of the datasets we have used the following strategies:

For selectivities lower than 1, we choose a fixed n which specifies how many triples to skip after we find a triple participating in a join. Then, we repeat this procedure during data generation for every basic triple pattern. It yields a selectivity of $\frac{1}{1+n}$ in each join operator. In the following we restrict n to be a power of two.

Additionally, we have generated datasets where data volume is increasing within the join operators. To create such datasets, we emit blocks of m triples, which are then joined with each other. In the following we choose m to be a power of two. In the experiments we call this ”selectivity”, too, because it still specifies the factor by which the number of rows changes within the joins.

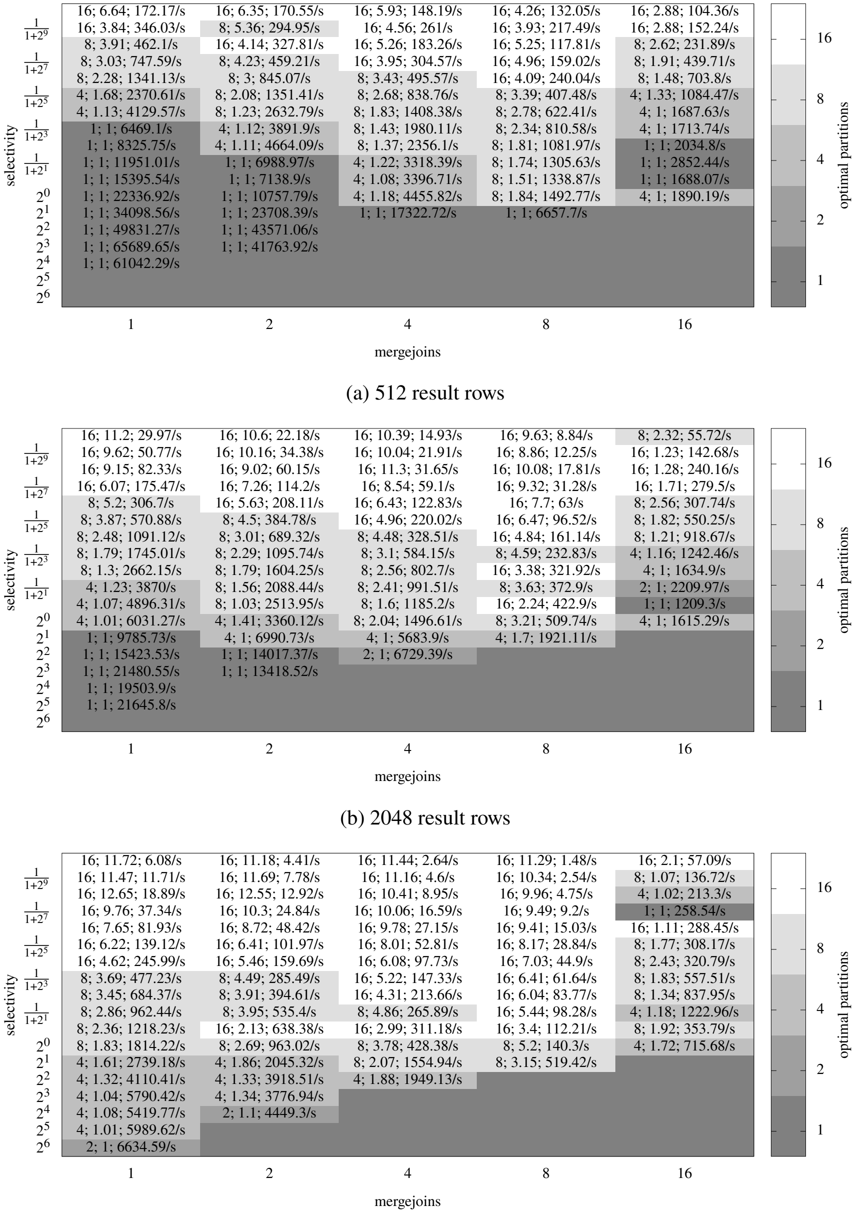
To generate our queries, we use the SPARQL template as shown in Fig. 4a. Afterwards, we change the number of triple patterns according to the desired number of joins.

In Fig. 5 we can see, that all of the three properties (output-rows, selectivity and number of joins) affect the optimal number of partitions for evaluating a query.

In the bottom right of each figure, there are missing experiments, because we can not create the target number of output rows. This is due to a massive increase of rows within the join operator chain, which is higher than the targeted output row count. Low optimal partition numbers in the bottom left part of each figure are caused by the same reason. Due to the very small number of triples in the store, it does not make sense to apply partitioning.

The optimal number of partitions is proportional to the theoretical workload, which we had expected. The more triples are stripped away due to a low selectivity, the more triples must be available in the store in the first place. The same holds for the number of merge joins. The more distinct triple patterns we want to join, the more input triples need to be defined. The last property, increasing the number of output rows obviously requires an increased number of input rows too. Vice versa, if we would have fixed the number of input rows, we would yield a similar result. In that case, the optimal number of partitions would be proportional to the number of output rows.

We used the results of this benchmark to predict the fastest partitioning within the query optimizer. As a base function we choose the polynomial $a \cdot x + b \cdot y + c \cdot z + d \cdot x^2 + e \cdot y^2 + f \cdot z^2 + g$ with ”x” as the number of result rows, ”y” as the number of joins and ”z” as the expected selectivity. We choose this function, because it promises good results for predicting the fastest partitioning and the function can be evaluated very fast during the query optimization



(a) 512 result rows

(b) 2048 result rows

(c) 8192 result rows

Fig. 5: Optimal number of partitions depending on the number and the selectivity of merge joins. The labels follow the form ”a;b;c”, where ”a” is the optimal number of partitions, ”b” the speedup compared to no partitions and ”c” the queries per second when no partitions are used. These experiments are run on computer configuration M2, because the additional RAM allows more experiments to be evaluated. Evaluating the queries on M1 achieve similar results.

phase. For our machine M2 we calculated the constants ”a” to ”g” such that we yield the complete function as seen in Fig. 6. We use the helper function $h(z)$ to convert the selectivity values to a similar range of numbers as all of the other variables. In a final step, we round the value to a discrete number of partitions. The prediction function $p(x, y, z)$ calculates most numbers of partitions optimally. Nevertheless we calculated the mean squared error between Fig. 6 and Fig. 5 to be 4.3030, which is quite small in comparison the huge number of known value pairs to fit.

$$\begin{aligned} h(z) &= -log_2(z) \\ f(x, y, z) &= 0.0025 \cdot x + 1.4827 \cdot y + 1.1277 \cdot h(z) + 0.0906 \cdot y^2 + 0.0279 \cdot h(z)^2 - 3.3696 \\ p(x, y, z) &= 2^{\lfloor \log_2(f(x,y,z)) \rfloor} \end{aligned}$$

Fig. 6: Prediction function for the number of partitions to use

Even if the above benchmark is evaluated on various synthetic datasets with different queries, we see the same effects in real world data, too. Every time a query uses a different constant (for example as a predicate), a completely different subgraph is accessed. Each subgraph in a real world dataset may contain a different number of triples as we have analyzed in Fig. 2. When we pick two different subgraphs, and join them with each other, we get very different selectivities within the join operators as well as different numbers of output rows, by only changing the query. This approves our assumption, that by just changing the query, the optimal partitioning scheme changes.

4.5.2 Macro benchmarks

This section presents a macro benchmark for comparing the overall speed of our LUPOSDATE3000 database system in a macro benchmark to other database systems in this section.

The database system LUPOSDATE was configured to use either its in-memory storage or a disk based RDF3X storage layout. Consequently, the results from both in-memory storage and disk based RDF3X storage layout are presented because the internal implementation of the triple stores is completely independent. All other database systems are using their default parameters as suggested by their documentation.

The performance measurements of the blaze-graph database system are suffering from very large measurement inaccuracies. All other database systems have a very low variation in the required time for the same query. To counter these inaccuracies, we repeated all experiments 10 times - and use the average measurements in our graphs.

For the experimental comparison, we have executed 10 merge joins on several different datasets. These datasets are generated in a manner to yield the target selectivity within each join operator. Fig. 7 shows the results of this comparison.

We have chosen to fix two different result sizes, and plot the graphs over a changing join selectivity. Both figures highlight different effects. The experiments of the Jena database system as well as the in-memory LUPOSDATE in the 128 result rows setting are dominated by the effect, that a fixed output size with a decreasing selectivity requires an increasing amount of input data.

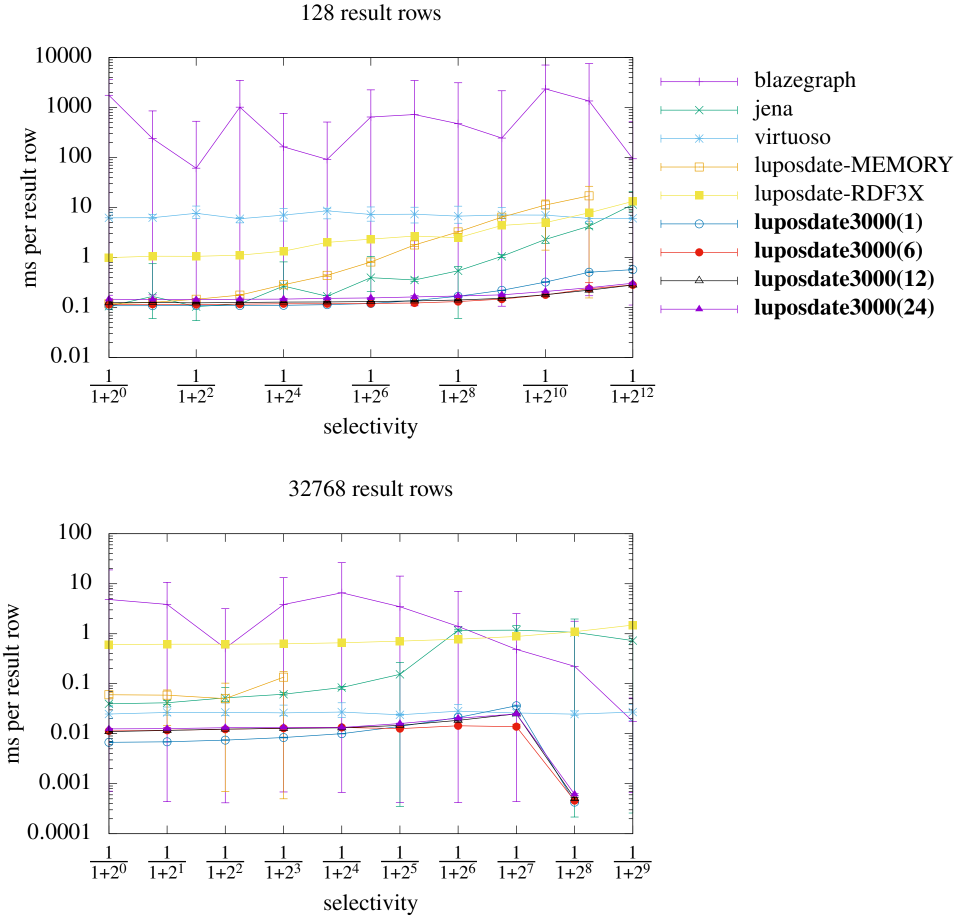


Fig. 7: Performance of Q1 with 10 merge joins on different database systems. These benchmarks are evaluated on M1 only, because we have multiple instances available, to perform more experiments. The numbers in the brackets, for example LUPOSDATE3000(6), show the number of used partitions.

Starting with a selectivity of $\frac{1}{1+2^9}$ the sequential performance of LUPOSDATE3000 decreases, while the time requirement of the partitioned evaluation remains the same. Although for the macro benchmarks the execution times for LUPOSDATE3000 include overhead for the endpoint communication and materialization of the string representations of the values, we still achieve speedups up to a factor of 1.81 compared to no partitioning, which shows that the correct number of partitions is significant for query evaluation.

All other configurations i.e. virtuoso, LUPOSDATE using RDF3x and partitioned LUPOSDATE3000 require the same evaluation time completely independent of the selectivity of the join operators, because very small data causes the database system to spend most of its time in static initialization.

When 32768 result rows are used, Fig. 7 shows a completely different performance characteristic, even if the only difference in the benchmark setup is the higher number of result rows. virtuoso as well as LUPOSDATE with RDF3X are still unaffected by changing the selectivity. This indicates that both of these database systems are limited by their capability to output their finished results. Contrary to before the required time per row decreases with decreasing selectivity for all LUPOSDATE3000 configurations as well as blaze-graph. This effect is based on the fact, that the static initialization time is getting smaller compared to the total evaluation time. Therefore, the overall speed per result row increases. The in memory LUPOSDATE variant suffers from bad memory management because e.g. no dictionary is used to map string representations to integer identifiers. Dictionaries are decreasing the memory footprint in all other database systems such that out-of-memory-errors are avoided during the triple load phase.

Even if the micro benchmark shows, that for huge numbers of result rows more partitions are better, the macro benchmark in the 32768 result rows setting is faster, if less partitions are used. At the same time both benchmarks show, that the sequential execution is slower. We believe, that this is caused by the different benchmark setup. Especially the sequential text output requires synchronization between the threads, which is not needed in the micro benchmark.

5 Summary and future work

In this paper we have investigated the factors which impact the performance of parallel SPARQL query processing. We have focused on the performance of data parallelism. According to our experiments, the performance depends on the amount of data in the store, the available hardware, and the structure of the query to process. In order to avoid overhead introduced by additional partitioning phases, we propose that the triple store materializes multiple independent partitioning schemes and choose the best among them on the fly. We present an experimental analysis and a concept of how a database system may optimize its query processing in this multi partitioning scheme context.

In the future we will implement and evaluate this partitioning strategy in a distributed database context. Due to our choice of Kotlin as our implementation-language, we will be able to run our database implementation directly on various operating systems. We plan to use these possibilities to evaluate our approach in a multi-operating-system environment like the Internet of Things with extremely heterogeneous hardware components. We expect that our approach will have huge advantages in those environments.

Acknowledgements

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 422053062

Funded by
DFG Deutsche Forschungsgemeinschaft
German Research Foundation

Bibliography

[Ab07] Abadi, Daniel J.; Marcus, Adam; Madden, Samuel R.; Hollenbach, Kate: Using The Barton Libraries Dataset As An RDF Benchmark. Technical Report MIT-CSAIL-TR-2007-036, MIT, 2007.

[AKN12] Albutiu, Martina-Cezara; Kemper, Alfons; Neumann, Thomas: Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. Proc. VLDB Endow., 5(10):1064–1075, June 2012.

[Ar11] Arias, Mario; Fernández, Javier D; Martínez-Prieto, Miguel A; de la Fuente, Pablo: An empirical study of real-world SPARQL queries. arXiv preprint arXiv:1103.5043, 2011.

[Ba05] Bailey, James; Bry, François; Furche, Tim; Schaffert, Sebastian: Web and Semantic Web Query Languages: A Survey. In: Reasoning Web, pp. 35–133. Springer Berlin Heidelberg, 2005.

[BK20] Bilidas, Dimitris; Koubarakis, Manolis: In-memory parallelization of join queries over large ontological hierarchies. Distributed and Parallel Databases, June 2020.

[BKS13] Biega, Joanna; Kuzey, Erdal; Suchanek, Fabian M: Inside YAGO2s: a transparent information extraction architecture. In: Proceedings of the 22nd International Conference on World Wide Web. pp. 325–328, 2013.

[DB20] DB-Engines Ranking of RDF Stores. https://db-engines.com/en/ranking/rdf+store, 2020. Accessed: 2020-09-16.

[GG11] Groppe, Jinghua; Groppe, Sven: Parallelizing join computations of SPARQL queries for large semantic web databases. In: Proceedings of the 2011 ACM Symposium on Applied Computing. pp. 1681–1686, 2011.

[Gr11] Groppe, Sven: Data Management and Query Processing in Semantic Web Databases. Springer, 2011.

[Ha07] Harth, Andreas; Umbrich, Jürgen; Hogan, Aidan; Decker, Stefan: YARS2: A federated repository for querying graph structured data from the web. In: The Semantic Web, pp. 211–224. Springer, 2007.

[Ha16] Harbi, Razen; Abdelaziz, Ibrahim; Kalnis, Panos; Mamoulis, Nikos; Ebrahim, Yasser; Sahli, Majed: Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. The VLDB Journal, 25(3):355–380, 2016.

[Ha18] Hankwang: , Hard drive capacity over time. https://de.wikipedia.org/wiki/Datei:Hard_drive_capacity_over_time.svg, 12 2018. Accessed: 2020-09-16.

[HHK19] Herrera, José-Miguel; Hogan, Aidan; Käfer, Tobias: BTC-2019: The 2019 Billion Triple Challenge Dataset. In: International Semantic Web Conference, Auckland, New Zealand. Springer, pp. 163–180, 2019.

[Ho13] Hoffart, Johannes; Suchanek, Fabian M; Berberich, Klaus; Weikum, Gerhard: YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. Artificial Intelligence, 194:28–61, 2013.

[JSL20] Janke, Daniel; Staab, Steffen; Leinberger, Martin: Data placement strategies that speed-up distributed graph query processing. In: Proceedings of The International Workshop on Semantic Big Data. pp. 1–6, 2020.

[NC20] Naacke, Hubert; Curé, Olivier: On Distributed SPARQL Query Processing Using Triangles of RDF Triples. Open Journal of Semantic Web (OJSW), 7(1):17–32, 2020.

[NW08] Neumann, Thomas; Weikum, Gerhard: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment, 1(1):647–659, 2008.

[NW10] Neumann, Thomas; Weikum, Gerhard: The RDF-3X engine for scalable management of RDF data. The VLDB Journal, 19(1):91–113, 2010.

[Pa13] Papailiou, Nikolaos; Konstantinou, Ioannis; Tsoumakos, Dimitrios; Karras, Panagiotis; Koziris, Nectarios: H 2 RDF+: High-performance distributed joins over large-scale RDF graphs. In: 2013 IEEE International Conference on Big Data. IEEE, pp. 255–263, 2013.

[Ru20] Rupp, Karl: , microprocessor-trend-data. https://github.com/karlrupp/microprocessor-trend-data, 7 2020.

[Sc09] Schmidt, Michael; Hornung, Thomas; Lausen, Georg; Pinkel, Christoph: SPˆ 2Bench: a SPARQL performance benchmark. In: 2009 IEEE 25th International Conference on Data Engineering. IEEE, pp. 222–233, 2009.

[SH13] Seaborne, Andy; Harris, Steven: SPARQL 1.1 Query Language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[SKW07] Suchanek, Fabian M; Kasneci, Gjergji; Weikum, Gerhard: Yago: a core of semantic knowledge. In: Proceedings of the 16th international conference on World Wide Web. pp. 697–706, 2007.

[TWS20] Tanon, Thomas Pellissier; Weikum, Gerhard; Suchanek, Fabian: YAGO 4: A Reason-able Knowledge Base. In: European Semantic Web Conference. Springer, pp. 583–596, 2020.

[WKB08] Weiss, Cathrin; Karras, Panagiotis; Bernstein, Abraham: Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment, 1(1):1008–1019, 2008.

[Ze13] Zeng, Kai; Yang, Jiacheng; Wang, Haixun; Shao, Bin; Wang, Zhongyuan: A distributed graph engine for web scale RDF data. Proceedings of the VLDB Endowment, 6(4):265–276, 2013.