In-Memory Parallelization of Join Queries Over Large Ontological Hierarchies

This is a pre-print of an article
published in Distributed and Parallel Databases. The final authenticated
version is available online at: https://doi.org/10.1007/s10619-020-07305-y

Noname manuscript No.
(will be inserted by the editor)

In-Memory Parallelization of Join Queries Over Large Ontological Hierarchies

Dimitris Bilidas · Manolis Koubarakis

Received: date / Accepted: date

Abstract The Resource Description Framework (RDF) data model enables the construction of knowledge graphs over various domains, using ontologies in order to encode information about the domain, and simple statements in the form of subject-predicate-object triples for data representation, facilitating the interlinking and exchange of Web data. However, this simplicity comes with the cost of having to execute a large number of joins in order to get the desirable query results, while at the same time large ontological hierarchies complicate the query answering process even more, for systems that provide complete answers with respect to such ontological axioms. In this work we present PARJ, an in-memory RDF store which takes into consideration ontological hierarchies during join processing with very low performance overhead, avoiding expensive preprocessing and materialization of implications, and is also amenable to straightforward parallelization. Specifically, we present a join implementation that allows to achieve any desired degree of parallelism on arbitrary join queries and RDF graphs stored in memory using compact vertical partitioning. We use an adaptive join processing approach, such that we take advantage of complete or even partial ordering of RDF data, which is compactly stored in order to increase spatial locality and keep memory consumption low, coupled with an ID-to-Position vector index used when ordering does not allow for efficient scanning of the input relation. Finally, we experimentally show the efficiency and scalability of our proposal.

Keywords RDF · SPARQL · OWL · Join Processing

D. Bilidas
National and Kapodistrian University of Athens, Greece
E-mail: d.bilidas@di.uoa.gr

M. Koubarakis
National and Kapodistrian University of Athens, Greece
E-mail: koubarak@di.uoa.gr

1 Introduction

The *Resource Description Framework* (RDF) [1] is a data model recommended by the W3C for semantic data integration, sharing and linking across different organizations and applications on the Web. RDF provides flexible modeling of data coming from heterogeneous domains in the form of triples forming subject-predicate-object statements, facilitating the construction of Knowledge Graphs. Every component of such a triple is a resource uniquely identified by an IRI or a data value in the form of a literal. The latter can only be present in the object position. A set of such statements can be considered an RDF graph, where subjects and objects are nodes and there exists an arc labeled with the property name, connecting corresponding subject and object for each statement. Several organizations publish data in the RDF model, leading to interlinking information from different sources and automatic processing using software agents. As a result, as of 2019 the *Linked Open Data* (LOD) cloud [49] contains more than 1200 datasets and 60 billion triple statements, with DBpedia [8], a dataset that contains semantic information extracted from Wikipedia, taking up a central position with 3 billion triples and around 50 million links to other datasets.

The SPARQL query language is the W3C recommendation for querying RDF graphs. The basic building block of SPARQL queries are triple patterns. A triple pattern is similar to an RDF statement, with the exception that each component (subject, predicate or object) can be either a resource or a variable. The evaluation of a single triple pattern over an RDF graph consists of finding matches of the pattern on the graph such that variables are substituted by RDF resources. A *Basic Graph Pattern* (BGP) is a set of triple patterns. During evaluation of a BGP all triple patterns are matched to an RDF statement and common variables between triple patterns are substituted by the same resource. If we consider RDF storage on a single relational triples table, a BGP with n triple patterns corresponds to $n-1$ self joins of the triples table.

On top of that, RDF can be extended with the ability to encode ontological knowledge which is useful when describing the domain of a knowledge graph. RDF Schema (RDFS) as well as more expressive ontological languages like OWL-2 QL define ontological constraints on top of RDF graphs, such that SPARQL query answering must be extended by taking into consideration the corresponding semantics in order to provide the user with the complete answers. Deep and wide class and property hierarchies pose a serious performance issue for all systems that perform query answering over RDF data with respect to entailment regimes that allow the definition of such hierarchies. Materializing all implied assertions with respect to these hierarchies, as it is the case in RDFS reasoning with forward chaining, is an expensive preprocessing step and it may lead to data size many times larger than the original, something that may not be viable especially for an in-memory system. On the other hand, using RDFS reasoning with backward chaining may lead to complicated queries. As we discuss in detail in Section 2.2, many approaches exist that aim to treat these problems, mainly focusing on disk based storage.

[1] https://www.w3.org/TR/rdf11-concepts/

Since the adoption of the RDF data model numerous systems and research prototypes have been developed aiming at efficient SPARQL query evaluation, focusing mainly on the evaluation of BGPs which proved to be extremely demanding. Centralized systems explored different physical storage options and query execution techniques. Main storage schemas include a single triples table, denormalized property tables, vertical partitioning, graph-based storage and storage based on bit arrays. Details and references to such systems are presented in the next section. As scalability became an issue with the continuously increasing size of several datasets, distributed approaches came into play, assisted by cloud technologies such as the MapReduce framework, its implementation Apache Hadoop and several Big Data processing systems built on top of it. Most of these systems use optimizations in order to minimize the execution cycles, which correspond to Hadoop jobs and involve data transfer between the workers. This is due to the synchronous nature of the MapReduce paradigm. As a result, depending on data partitioning and replication one can achieve evaluation completely in parallel for some queries, but for queries that require communication the overhead is important due to the synchronization step.

A number of in-memory distributed systems were later proposed such that their communication is based on custom asynchronous methods, mostly on the Message Passing Interface (MPI) standard. Trinity.RDF [58] is based on graph exploration and it was the first system to follow this design. TriAD [18] and the extension of the centralized main memory RDF store RDFox with a dynamic data exchange operator [39] also use an asynchronous execution model (In what follows we will refer to the system described in [39] as the dynamic exchange operator approach), but unlike Trinity.RDF they use relational-style joins, increasing the level of parallelism for large intermediate results over the graph-based approach. In order to do so, both of these systems use expensive graph partitioning before data loading. AdPart [3] tries to overcome this problem by using simple subject-based hash partitioning and then adaptively, based on the query load, replicates specific data fragments to the workers. As a result of the initial subject-based partitioning, expensive broadcast of intermediate result occurs in case of joins on objects.

In this work we present *PARJ*, an in-memory query processing system able to parallelize multi-join queries over large RDF graphs. Its name stands for Parallel Adaptive RDF Joins. Our query processing approach is inspired by the asynchronous execution model of main-memory distributed RDF stores, mainly of TriAD and the dynamic exchange operator approach. Both these approaches use expensive preprocessing in the form of graph partitioning in order to minimize communication between servers during query execution. Also, extra effort is needed in order to track the server that contains each resource. Most importantly, even in a centralized parallel environment these systems would require some form of inter-process or inter-thread communication and as a result some form of synchronization. For example, in case of rehashing, each worker of TriAD has to wait in order to receive and rehash all intermediate results from all other workers. Same kind of overheads occur in the dynamic exchange operator where each worker must hold a queue for each query atom, where incoming messages are put. This may lead to blocking execution until some other worker process results for a subsequent query atom. Also, in the dynamic exchange operator approach detecting termination is not trivial and requires a round of mes-

sage exchanging. Our method ensures parallel execution of arbitrary multi-join BGPs without any form of communication or synchronization between the workers (in our case threads) while at the same time avoiding expensive preprocessing like graph partitioning. Furthermore, we adaptively decide to scan the corresponding partitions when it is preferable, instead of always using index-based nested loops as done by the dynamic exchange operator approach. This adaptive cache-friendly method can take advantage of existing (even partial) sorting of RDF triples, that further improves our join implementation. An auxiliary bit vector index can be used to avoid binary search and improve efficiency.

Regarding the physical data storage, our approach is inspired by *column-store* systems such as MonetDB [21] and C-Store [52], as we first use vertical partitioning [1] to create a separate table for each property, and then keep subjects and objects for each property in separate arrays so that each tuple can be reconstructed by relating entities at the same positions in these arrays, reminiscent of the virtual IDs of column stores. This physical design compactly stores RDF data in memory, in order to increase spatial locality during join processing. For example, for scale 10240 of the LUBM dataset with about 1.4 billion triples, excluding dictionary, the storage requirements are only 22 GB (50GB if we include the dictionary). Also, we allocate a single array position for each distinct subject or object as a simple form of column specific compression (reminiscent of the POS and PSO indexes used by Hexastore [54]) and we keep two replicas of each two-column table in different sort orders.

Finally, PARJ is able to perform scalable query answering with respect to large class and property hierarchies by providing *virtually* complete data over these hierarchies. Specifically, during join processing, we incorporate on-the-fly union computations over our physical data storage without impairment of the pipelined execution model. Our architecture is based on [43], modified in order to use our in-memory system as a triple-store, instead of a relational database, in order to perform query answering over OWL2-QL ontologies. Our approach does not require expensive preprocessing in the form of materializing ontology inferences via forward chaining, and at the same time it only has an 10-20% overhead in query execution time in comparison with complete materialization of ontological hierarchies. Our experimental evaluation provides the fastest execution times over the $LUBM_\exists$ [28] OWL benchmark, outperforming state of the art systems based on materialization or query rewriting.

This paper is an extension of publication [9], where an initial version of PARJ had been presented, by adding the following novel contributions:

- An experimental evaluation with larger (LUBM 20480) and real-world (YAGO2) datasets that confirms the scalability and applicability of our approach
- A method for incorporating information about ontological type and property hierarchies during the join processing without any additions to the physical data storage layout and without affecting the scalability and effective parallelization of execution
- An implementation that couples PARJ with Ontop[13], a state of the art tool for Ontology-Based Data Access (OBDA), enabling PARJ to act as an efficient RDF store, answering queries over OWL2-QL ontologies. We also perform an experi-

mental evaluation and comparison of our implementation with other state of the art systems for OWL2-QL query answering.

We proceed by first presenting related work. Then in Section 3 we present details of the physical data storage and execution model and in Section 4 we present details of the adaptive join method that allows for incorporating parallelism into processing. Query answering approach over ontological hierarchies for OWL 2 QL ontologies is described in Section 5. We present implementation details and experimental evaluation in Section 6 and we finally conclude and discuss future work.

2 Related Work

In this section we present related work with respect to i) RDF and SPARQL processing and ii) Ontology-Based Data Access. The first subsection provides the relevant background regarding our contribution in creating a scalable and parallelizable method for executing multi-join queries over large RDF graphs, whereas the second subsection provides background related to query answering with respect to ontologies, focusing on the OWL2-QL language.

2.1 RDF and SPARQL

RDF storage and processing can be distinguished in three perspectives: i) a relational perspective, (ii) an entity perspective and (iii) a graph-based perspective [27]. Our work is mainly following the first perspective, as using relational technology for RDF processing has been a subject of research since the proposal of the RDF data model with prominent results. BGP evaluation using a single triples table that contains the whole RDF graph involves expensive self joins over this large table. As a solution, some systems like Jena [55] proposed the usage of "flattened" property tables, which contain a larger number of columns, in an effort to simulate a relational schema and avoid joins as much as possible. Nevertheless, this design has some drawbacks, like for example a lot of NULL values for wide tables, the need for UNION during a single BGP processing and difficulty to handle multi-values attributes. [5] aims at efficient evaluation using an object-relational DBMS including a two-column representation for properties. Vertical partitioning[1] uses this representation in order to treat the drawbacks of the property tables. In this approach a separate two-column table is created for every property of the RDF graph. In this case, the number of joins is not reduced in comparison to the single triples table, but each join is between smaller tables and also tables not relevant to the query do not need to be accessed at all. Column stores are ideal candidates for RDF processing using vertical partitioning, as they provide compact storage and compression over each column. Our physical design is based on vertical partitioning as in [1], combined with techniques from column stores adapted for efficient in-memory RDF data storage.

Hexastore enhances the vertical partitioning by replicating the data through six different indexes, corresponding to all possible permutations of subject, predicate and object [54]. RDF-3X [35] also uses extensive indexing such that an index is created

not only for all possible permutations but also for aggregated values, resulting in 15 indexes stored as clustered B+ trees. This schema along with several optimizations, such as skipping large parts of irrelevant data during merge joins using a form of *sideways information passing*, made RDF-3X one of the most efficient disk-based RDF stores, despite conceptually using the single triples-table approach. As in Hexastore, we keep two different replicas for each property in different sort orders (corresponding to POS and PSO indexes) with respect to our vertical partitioning data storage, and we also compactly store only distinct subjects and objects. Also, our adaptive join optimization (Section 4.1) can be considered a way of skipping irrelevant data as in RDF-3X.

Regarding SPARQL query processing using cloud technologies, [22] provides an overview and classification of systems and approaches in different categories regarding several characteristics. Here we briefly mention the most relevant research. An initial approach using the MapReduce framework is presented in [44,45]. In this work, the authors describe the query evaluation of Basic Graph Patterns of SPARQL using an iterative algorithm, such that every join in the query requires a separate MapReduce job. The RDF data is stored in plain files in the distributed file system. A similar iterative approach is also used in [32], but here the authors note that more than one triple patterns that share a variable can be joined together in the same MapReduce job. They use a greedy selection algorithm that chooses in every step the variable that appears in more triple patterns and they employ reduce-side joins to get the results. In [15] predicate-based hash partitioning is employed. The query is decomposed to subqueries using the same partitioning and in every node a local Sesame RDF store is used to evaluate each subquery. Instead of hash partitioning, in [20] the authors use a graph partitioning algorithm to assign triples to nodes and also they employ data replication for triples that are on the boundaries of each partition, in order to maximize the number of subqueries that can be executed without communication between the nodes. They stress the usefulness of a heuristic that finds the minimal number of subqueries because this corresponds to a minimal number of Hadoop jobs, and they split each query to a number of such subqueries using a brute-force method, which is suitable only for queries with few triple patterns.

A number of approaches store the RDF data into an existing system that has its own declarative language and then they transform the SPARQL queries into that language. For example, [47] uses Pig Latin[36] and performs some well known optimizations to the SPARQL query, like the early execution of filters and some selectivity estimations based on variable counting. During the translation to Pig Latin, [47] just uses multi joins when consecutive joins on the same variable are found, as this is an option that Pig Latin offers. RAPID+, a system which is also based on Pig Latin, is presented in [42]. Here the authors propose an intermediate algebra which is called Nested TripleGroup Algebra, in order to facilitate the grouping of join operators during the translation of the query to the execution plan in Pig Latin. The result is that each star join involving two or more triple patterns can be executed in one Map-Reduce job, using vertical partitioning.

H_2RDF+ [37] uses HBase[2] to store the RDF data. It takes advantage of the HBase key ordering for each table and it uses six tables, each one corresponding to an RDF triple permutation. In this way there is replication of the data, so that the system can perform fast merge joins when all triples are part of the initial RDF data. When some data is result of an intermediate step, the system first performs a sorting on this intermediate data. Another key feature of the system is that during the query planning it examines the option that the query will be executed in a centralized system. The rationale behind this is that if the query is simple, its evaluation in a centralized system can be preferable, because one can avoid the overhead of the MapReduce jobs and network communication. The system uses a greedy planner to decide about the order of the joins, based on a cost model and some index statistics that it has. In a similar manner, the system named Rya[40] uses Accumulo[3], to store indexes for permutations of subject, predicate and object in the row ID field of each corresponding table, but it only uses three indexes instead of all the possible ones. Rya supports range queries and regular expressions, multi-threaded join execution and also provides some limited inferencing capabilities. S2RDF [48] uses the in-memory system Spark to store the RDF data using vertical partitioning combined with semi-join materialization and then translates the SPARQL query to Spark SQL [7].

Regarding in memory join processing, a lot of research has been concentrated on cache friendly methods, such as the radix hash join [29], and also into taking advantage of hardware features such as the SIMD vectorized instructions for efficient parallel sort-merge joins [4,25]. These works consider the setting of relational data with arbitrary number of columns, where a single join has to be performed on previously non indexed columns and sorting or hashing is a serious overhead that has to be performed in parallel. Instead, our work is tailored for RDF graphs, as it exploits initial ordering of both subject and object RDF columns and partial ordering of subsequent joins for pipelining multiway joins, such that it completely avoids hashing or sorting during query execution. Exploiting partial ordering of values in a column has been used by main memory systems in the form of zone-maps [50,41] where additional statistics about each such zone have to be maintained in order to skip scanning certain areas. On the contrary, our join processing does not need to rely on extra statistics as in zone-maps. Adaptivity during run-time regarding the decision of scanning a base relation or use a secondary index has been studied in [10,11] for disk-based systems.

Regarding centralized parallel in memory RDF processing, to the best of our knowledge there is no work concentrating on query processing. RDFox [33] and Inferray [53] are both systems that aim at parallel in memory computation and materialization of RDF inferences. This can be thought of as a preprocessing step prior to querying. Although RDFox offers query evaluation, it seems that is not the focal point of the system and for such queries there is no support for intra-query parallelism, that is each query is evaluated in a single thread. In [16] several variations of the disk based RDF-3X are presented, such that they allow parallel join evaluation. From the experimental results it is shown that depending on the query, there is no clear variation that has better performance, whereas for some queries the original version is

[2] http://hbase.apache.org/

[3] http://accumulo.apache.org/

better, as parallel evaluation prohibits the usage of the sideways information passing optimization in RDF-3X. Also, their approach works by parallelizing each join separately and demands communication and synchronization costs.

2.2 OBDA

Ontologies can be used on top of RDF graphs in order to enrich the semantic information by providing a vocabulary that facilitates the conceptual modeling of a specific domain. For example, one can use an ontological statement to declare that if someone teaches a university course, then he is a professor. Then, for a given individual such that the data contain the information that he teaches a course, a query answering system under the appropriate entailment regime can deduce that this individual is a professor, even if this knowledge is not explicitly stated in the data. A system can follow two main approaches to provide complete answers under such entailment regimes. The first one is query rewriting, which is similar in spirit with backward chaining of datalog evaluation. Under this approach the original query is rewritten in order to provide complete answers when posed over the incomplete data. This method has the advantage that it does not need data preprocessing, but on the other hand it produces more complicates queries. On the other hand, the second approach that is similar with forward chaining, uses materialization during data loading in order to add all the implicit knowledge to the data. This usually achieves better query performance, but it involves expensive data preprocessing, leading to increased database size and making things complicated during data updates or when the ontology changes.

There are several ontology languages aiming to different trade-offs between expressivity and efficiency of reasoning tasks, but regarding conjunctive query answering over RDF graphs, W3C recommends the usage of a specific profile of the OWL language which is called OWL2-QL. Under this entailment regime, it is guaranteed that it always exists a first-order query rewriting of a conjunctive query such that, when this rewriting is executed over the incomplete data, it provides the full answers implied by the OWL2-QL ontology. Indeed, several methods in order to obtain such a rewriting have been proposed, assuming that data are stored in an external, usually relational, database. In many cases, this rewriting has the form of a union of conjunctive queries, as for example in the PerfectRef [38] which was the first such proposed rewriting. It was soon observed that the result union of conjunctive queries was in many cases prohibitively large in order to be evaluated efficiently (for example it could contain thousands conjunctive queries), and as a result, optimized methods that produce queries with fewer unions were proposed, based on query containment [31, 14]. Nevertheless, the result could still be extremely large. In order to solve this problem, some rewriting methods were proposed aiming to produce more compact rewritings in the form of nonrecursive datalog instead of union of conjunctive queries [46, 24], but efficient evaluation of such rewritings in existing database engines is still an issue. In [12] a cost-based comparison of different reformulations is carried out and in general, the final rewriting will be a join of unions of CQs. Semantic Index [43] contains an arithmetic encoding of class and property hierarchies and stores RDF data into relational back-ends using the appropriate B-tree indexes, such that class

and property membership can be determined by range queries over these indexes, avoiding a large number of union subqueries.

On the other hand, a complete materialization of all the implicit knowledge defined by an OWL2-QL ontology in the general case may not be even possible, as the canonical model of the ontology and data may be infinite. As a result, regarding methods based on forward chaining, there have been proposed a combined approach [26] relying on materialization with respect to certain axioms combined with query rewriting, and an extension of that which uses finite materialization coupled with a filtering procedure in order to discard spurious answers[28].

Commercial RDF stores that support query answering over OWL2 QL ontologies include Stardog [4] that is based on query rewriting, and GraphDB [5] that relies mostly on materialization. Our work, following the architecture of [43], is also based on query rewriting, using PARJ as the execution engine, instead of an external *relational database management system* (RDBMS), as we describe in Section 5.

3 Physical Data Storage and Execution Model

In this section we present our physical data storage and give an overview of the join method that allows incorporation of parallelism. First, following the common practice used by many systems, we use dictionary encoding, by assigning an integer value to each value encountered in the RDF data. We use common numbering for values appearing in the subject and object positions and a different numbering for values appearing in the property position, but for ease of presentation here we assume common numbering for all values. Thus, after parsing of an RDF dataset that contains N distinct values, our dictionary will contain integer IDs from 1 to N. Then, we apply vertical partitioning [1] to create a separate two-column table for each property defined in the data. We keep two replicas of each two-column table, the first sorted on subject and then on object, and the second sorted first on object and then on subject. Given that a property P is assigned to integer i from our dictionary encoding, we will refer to the first replica of two-column table for P as $prop_i$ and to the second replica as $prop_{i^-}$ and we will call the tables first sorted on subject *S-O tables* and tables first sorted in object *O-S tables*.

Consider for example the following RDF data (IRIs are omitted):

```
ProfessorA teaches Mathematics
ProfessorB teaches Chemistry
ProfessorC teaches Literature
ProfessorA teaches Physics
ProfessorA worksFor University1
ProfessorB worksFor University2
ProfessorC worksFor University2
```

[4] https://www.stardog.com
[5] http://graphdb.ontotext.com/

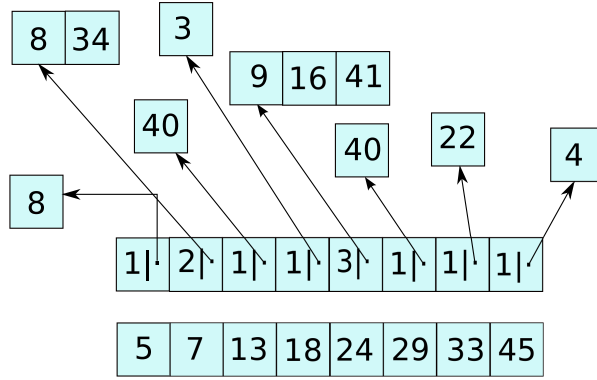

Fig. 1: Example of Physical Data Storage for a Property Partition

The dictionary encoding of the data is given in Table 1. Using this encoding, the two-column tables $prop_2$ and $prop_9$ that correspond to properties $teaches$ and $worksFor$ will be created.

Table 1: Example of Dictionary Encoding

Integer	**Value**
1	ProfessorA
2	teaches
3	Mathematics
4	ProfessorB
5	Chemistry
6	ProfessorC
7	Literature
8	Physics
9	worksFor
10	University1
11	University2

For each table, we store a sorted integer array with the distinct subjects (for S-O tables) or distinct objects (for O-S tables). We also store a second array of same length with the first. Each position of this second array contains a pointer to a sorted integer array and an integer denoting the length of this array. This is a pointer to the objects (for S-O tables) or subjects (for O-S tables) that correspond to the subject (respectively object) located at the same position of the first array. The reason that we keep two separate arrays has simply to do with compactly storing the integers of the first array and improving spatial locality during the join processing. We also keep track of the length of the first array, using an array of length $2 * (number\ of\ properties)$ that contains this information for all properties. Getting this information involves a simple lookup at a specific position, for example, to get the number of subjects for $prop_7$, we should look at position $2 * 7$, whereas to get the number of objects for $prop_{7^-}$ we should look at position $(2 * 7) + 1$.

Figure 1 contains an example of physical storage for a property table. Assuming that the specific table has been created for property $prop_3$, then it contains the following triples: 5 $prop_3$ 8, 7 $prop_3$ 8, 7 $prop_3$ 34, 13 $prop_3$ 40, 18 $prop_3$ 3, 24 $prop_3$ 9, 24 $prop_3$ 16, 24 $prop_3$ 41, 29 $prop_3$ 40, 33 $prop_3$ 22, 45 $prop_3$ 4. Note that in order to avoid memory fragmentation, the different object arrays of this example can be allocated to a continuous memory area. In this case, instead of having different pointers for each position of the second array, we can keep a single pointer to the start of this memory area and only keep offsets in each position of the second array.

Our execution models targets multi-threaded environments, where each thread operates on the common data without any form of inter-thread communication. To achieve this, our join method resembles an index-based nested loops join (or merge join when possible - this will be discussed later), such that each thread is assigned a different shard of the first (leftmost) table in the join, and runs in parallel, by probing the next table to be joined for each tuple. Given a number of available threads, the first table of a join is virtually partitioned in an equal number of shards, such that every shard contains about the same number of tuples. In this way our method operates on left-deep query join trees as shown in Example 1.

Example 1 Consider a SPARQL query:

```
SELECT ?x ?y ?z
WHERE {
?x teaches ?z .
?x worksFor ?y .  }
```

Also suppose that the join order chosen by the optimizer (see Section 5.3) is the same with the order of the triples in the text of the query. This will be translated to a join $prop_2 \bowtie_{subject=subject} prop_9$. If there are two available threads, our algorithm will start concurrently scanning two different shards of $prop_2$. For each tuple encountered during this process, it will probe, using binary search, table $prop_9$. This process can be decomposed into completely independent tasks that start from different shards and operate on read-only common data, and thus it straightforward to be implemented using threads (as it is our current execution model and implementation) or separate processes with shared memory. It is even straightforward to be implemented on different machines using complete data replication and parallelize the query across machines without any communication.

Note that for the given query, the degree of parallelism depends on the number of different shards of the first table. For more selective queries a different strategy may be needed as shown in Example 2.

Example 2 Consider the following query, that contains an extra filter:

```
SELECT ?x ?z
WHERE {
?x teaches ?z.
?x worksFor University1 .   }
```

In this case, suppose that the optimizer chooses the inverse join order, as it is reasonable that the filter will limit the results of the second triple pattern. In this case, table $prop_9$ should be scanned first. One first observation is that instead of scanning the whole table, we can search for tuples where object is equal to 10. To do so it is better to use the replica that is first ordered by object. After we search $prop_{9^-}$ for $object = 10$, we obtain the vector of subjects that correspond to $object = 10$ (in our case it is only value 1). Then we start scanning this vector and probing table $prop_2$ using these values. In this way we do not obtain any level of parallelism for this query, as we start from a specific value of the first table. It is easy though to recover the parallelism, if we start scanning concurrently different shards of the vector that corresponds to $object = 10$. If the query contains a triple pattern with variable in the predicate position, then a union over all properties will be needed, but this is rarely encountered in real world queries[1]. In any case, if the number of distinct predicates encountered in the dataset is very large, an ID-Predicate index similar to the one use in [57] can be useful. Also note that the exact number of threads that will be used is independent of our physical data storage and can be decided on a per query basis after data loading in memory. In our current implementation (Section 6) we choose to execute each query with the same number of threads (optimally this should be equal to the number of available processing cores or greater in case hyper-threading is supported as shown in Section 6.2.3), but an extension such that very simple and selective queries could be executed with fewer resources is possible.

4 Query Processing

The approach followed by RDF stores like RDF-3X and TriAD, is to take advantage of initial sorting of RDF triples, and perform merge joins when possible. Hash join is preferred when inputs are not sorted on the join key. On the other hand, the dynamic exchange operator approach always uses index-based nested loops aiming at low memory consumption and avoiding blocking operators. Our system uses a combination of these two approaches, by taking into consideration the following points:

- When both inputs are already sorted on the join key, merge join is preferable over hash join.
- For main memory systems, index-based nested loops (in our case in the form of binary searches over the inner table stored as an array) does not exploit data locality and also it is not amenable to efficient data prefetching due to conditional branching. Nevertheless, for very selective joins, it may still be faster than merge join.
- For RDF data processing, where the initial triples are sorted in all three subject, predicate and object columns, even if the whole input is not sorted on the join key of a subsequent join, large portions of the input can still be sorted as it is demonstrated in the following example.

Example 3 Consider the following SPARQL query:

```
SELECT ?x ?y
```

```
WHERE {
?x prop1 ?y .
?x prop2 ?z .
?z prop3 ?w .    }
```

If the selected join order is as shown in text of the query, S-O tables will be used for all properties. As shards of prop1 are scanned, for each thread of execution, $prop2$ will be probed for values sorted on $?x$, but for the second join, probing $prop3$ will not in general be sorted on $?z$. Nevertheless, for each distinct $?x$, $prop3$ probing will still be sorted on $?z$ and if each subject of $prop3$ is connected to many objects, it may be more efficient to avoid binary search on $prop3$ and switch to scanning for each distinct $?x$.

A single join operator has been implemented in our system, that adaptively during run-time, for each search key, decides if it will switch to binary search (a behavior similar to index-based nested loops) or keep scanning the input in the form of sequential search, continuing from the position that the cursor has been left from a previous search (a behavior similar to merge join).

4.1 Adaptive Join Processing

Given a left-deep join tree produced from the optimizer, each worker starts scanning a shard of the first relation, or a specific shard of an object/subject vector of the first S-O/O-S relation in case a filter exists, and searching the subsequent relations for each produced tuple. The search procedure is presented in Algorithm 1. The algorithm takes as input a pointer to current cursor position ($cursor_position$), which corresponds to the position of the last accessed element for the array, and decides if it will use binary or sequential search. The $cursor_position$ is updated each time for both successful and unsuccessful searches inside the functions $Sequential_Search$ and $Binary_Search$.

Obtaining an exact cost-model in order to take the correct decision is an involved process that needs to take into consideration factors such as the exact cache hierarchy, the size and bandwidth estimation for each cache level for both sequential access (scanning) and random access, cache line size, the replacement of cache entries from operations other than the join under consideration (for example subsequent joins of the same query) and the existence in cache of relevant entries from previous operations (for example scanning of the same relation in a previous query). Obtaining such cost models for hierarchical memory systems has been studied in [30], where cost functions are defined for basic access patterns and then combinations of these functions can be used to derive the cost of complex compound access patterns. As a prerequisite, specific hardware measurements should be known, which can be obtained through a separate calibration program that estimates cache and CPU characteristics.

In our case, decision has to be made during runtime for each produced tuple and each join of the query. Instead of using an analytical cost model, we opt for a fast and lightweight method using two assumptions: a uniform distribution of integers in the first array of each table and that existing cache contents have an impact

proportional to the cost of either binary search or scanning. The second assumption simply denotes that existing cache contents can improve both methods, but they will not change which the methods is more efficient in each case. For example, if binary search is preferable with completely empty cache, it will remain so independently of the cache contents and vice versa. As a result we base our decision on the difference between the last accessed element and the element that we are currently searching for. Specifically, we pass as argument to the algorithm a threshold which is computed during data loading for each table. This threshold takes into consideration an estimation about the maximum distance of the position of the last accessed element and the position of the element to be found in the array, in order for sequential search to be preferable. To switch from distance in the array to the actual arithmetic distance of the two numbers, we use the uniform distribution assumption, which leads to an estimation that the difference between an element and its subsequent one is $(array[size-1]-array[0])/size$. Note that in Algorithm 1, if $Distance > Threshold$ then we could perform binary search using as starting position the position denoted by $CursorPosition$ instead of 0, and if $Distance < -Threshold$ we could use $CursorPosition$ as the end position instead of $size$. In theory this reduces the steps needed from binary search, but in practice it is not efficient, as always performing binary search on the whole array leads to the array positions visited during the first steps to frequently occur in cache.

Regarding the determination of the threshold, a calibration process shown in Algorithm 2 is used. This process takes place after data loading, prior to query execution, and tries to determine a distance (called $WindowSize$) such that when searching for a value $ToFind$ in the $Array$ and the position of $ToFind$ is at distance $WindowSize$ from the position of the last accessed element ($CursorPosition$), then $BinarySearch$ and $SequentialSearch$ perform roughly the same. Specifically, the ratio of the larger to the smaller execution times of these two methods should be smaller than a value close to 1.0 which is specified in the input of the algorithm ($Threshold$). For each calibration step, each process is called $NoOfSearches$ times, each time searching for a value whose distance from the previous one is estimated to be equal to $CurrentWindowSize$. If the ratio is larger than the $Threshold$, calibration continues such that the window size is multiplied by this ratio (in case time spent on binary search is larger) or divided (otherwise). This calibration process is different from a calibration needed when using an analytical cost model, in the sense that we directly make an estimation for a value related to processing, instead of estimating values about several hardware characteristics. Once the calibration process terminates, we precompute the estimated value distance (corresponding to the position distance that we obtained) for each property, such that during query execution we only need to perform one integer subtraction, one absolute value computation and one comparison for each tuple (lines 2-3 of Algorithm 1).

4.2 ID-to-Position Index

Our join method takes advantage of initial sorting and performs cache-friendly joins even when only a partial order of input triples is possible, but when ordering does

Algorithm 1: Adaptively switching between binary and sequential search

```
1 Search (Array, Value, CursorPosition, Threshold, Size);
  Input  : Array: an array of integers (subjects of an S-O table or objects of an O-S table),
           Value: integer value to find, CursorPosition:pointer to current cursor position,
           Threshold: integer, Size: size of array
  Output: nonnegative integer corresponding to the position of Value in Array or a negative
           integer if Value is not present in the Array
  Uses   : Binary_Search(Array, Value, CursorPosition, Size),
           Sequential_Search(Array, Value, CursorPosition, Size)
2 Distance := Array[CursorPosition] − Value;
3 if |Distance| <= Threshold then
4 |   return Sequential_Search(Array, Value, CursorPosition, Size);
5 else
6 |   return Binary_Search(Array, Value, CursorPosition, Size);
7 end
```

not help we must resort to binary search. In this section we describe the structure of an ID-to-Position index that is used to avoid binary search and directly locate the position of a given integer on the property array. A separate such ID-to-Position index must be built for each S-O or O-S table, but its usage is auxiliary, in the sense that our system can operate without all or some of these indexes. Given an RDF dataset with N distinct values and a corresponding dictionary with IDs from 1 to N, in order to directly locate the position of a given value in a table, we need to store an integer array of length N, such that the value at index p denotes the exact position at the table where it is located the resource whose ID value according to the dictionary is p, or a special value to denote absence of the specific resource from the table.

For example, given the property shown in Figure 1 and supposing that the maximum ID contained in the dictionary is 45, we would need an array of integers with length 45, such that at position 5 of the array we would have the value 0, at position 7 the value 1, at position 13 the value 2 and so on for positions $18, 24, 29, 33$ and 45, and all other position of the array would have a value denoting absence. If we use M-byte integers, then for each table the memory requirement would be $M * N$ bytes. In order to save space, we use a different layout on out ID-to-Position index, such that we only use an integer to denote the position of the property table at specific intervals, and for all other positions we use a bit value to simply denote presence or absence of value from the property table. Finding the exact position for a value requires reading the previous integer and then counting bits set to 1 up to the position of the ID-to-Position Index corresponding to the value. For example, if we choose the interval to be equal to 8, then our index will store the integer -1 at start, followed by bit values $0, 0, 0, 0, 1, 0, 1, 0$, then integer value 1 and bit values $0, 0, 0, 0, 1, 0, 0, 0$, then integer value 2 and bit values $0, 1, 0, 0, 0, 0, 0, 1$, then integer value 4 and bit values $0, 0, 0, 0, 1, 0, 0, 0$, then integer value 5 and bit values $1, 0, 0, 0, 0, 0, 0, 0$ and finally integer value 6 and bit values $0, 0, 0, 0, 1$. If we want to find the position of value 29 at the property we can directly check bit at position $((29 \div 8) + 1) * M * 8 + 29$. If bit is not set, then value is not present in property table. If bit is set we read integer value that starts at bit position $(29 \div 8) * M * 8 + (29 \div 8) * 8$ at the array and we add to this the number of bits that are set after this number for $29 \ mod \ 8$ positions.

Algorithm 2: Calibration Process

```
1  Calibrate (Array, NoOfSearches,
     StartingWindowSize, Threshold);
   Input  : Array: an array of integers (subjects of an S-O table or objects of an O-S table),
            NoOfSearches: number of times to run sequential and binary search in each
            calibration step, StartingWindowSize: initial window size used in first step of
            calibration, Threshold: A threshold ratio to stop calibration
   Output: integer corresponding to the window size such that if two values in array are longer
            apart then binary search is preferable
2  NextWindowSize = StartingWindowSize;
3  AvgGap = (Array[Size − 1] − Array[0])/Size;
4  do
5      WindowSize = NextWindowSize;
6      TotalGap = AvgGap * WindowSize;
7      PreviousSearchPosition = 0;
8      StartTime = getTimeNow();
9      ToFind = Array[0];
10     for K ← 0 to NoOfSearches do
11         Binary_Search(Array, ToFind, 0, &PreviousSearchPosition);
12         ToFind+ = TotalGap;
13     end
14     TimeBinary = getTimeNow() − StartTime;
15     toFind = Array[0];
16     PreviousSearchPosition = 0;
17     StartTime = getTimeNow();
18     for k ← 0 to noOfSearches do
19         Sequential_Search(array, toFind, &PreviousSearchPosition);
20         ToFind+ = TotalGap;
21     end
22     TimeScan = getTimeNow() − StartTime;
23     TimeDiff = |TimeBinary − TimeScan|;
24     if TimeBinary > TimeScan then
25         Fraction = TimeBinary/TimeScan;
26         NextWindowSize = WindowSize * Fraction;
27     else
28         Fraction = TimeScan/TimeBinary;
29         NextWindowSize = WindowSize/Fraction;
30     end
31 while Fraction > Threshold;
32 return WindowSize;
```

With this layout, given an interval A we only need $N/8 + ((N/A) * M)$ bytes. Also, given that the integer and the number of bits followed up to the next integer fit into a single cache line (with proper alignment of the index in the memory), we only need one memory access and some computation that can be done efficiently as a popcount operation in order to determine the position.

As an example, using the dataset LUBM 10240 described in Section 6, which contains about 1.4 billion triples, 17 distinct properties and about 336 million distinct resources, using 4-byte integers and choosing the interval to be 480 we only need 44.8 MB for each property, leading to a total memory usage of about 1.5 GB if we choose to create all possible indexes for S-O and O-S tables, in contrast to a memory requirement of 45.7 GB if we had used the simple layout.

Regarding modification of the join processing in case the ID-to-Position index is used, the only change that needs to be addressed is a different threshold resulted from calibration process. Specifically, since we anticipate that using the index will have better behavior in comparison with binary search, we need to estimate two different thresholds with regards as to when sequential search is preferable, with the threshold when ID-to-Position index is used being smaller than the threshold when binary search is used.

5 Query Execution Over Ontological Hierarchies

We begin this section by describing the system design and general architecture of our approach which is based on [43]. Essentially, it modifies the mentioned system by removing the external RDBMS that stores RDF data with the semantic index schema, and replacing it with an extension of PARJ, able to provide access to virtually complete ontological hierarchies in an efficient manner.

5.1 System Design

Three main reasons for the presence of long, and as a result highly inefficient, rewritings have been specified in [43]: i) subqueries with existentially quantified variables, ii) large ontological hierarchies and iii) multiple mappings for each ontology term. The last reason is usually relevant when arbitrary relational schemas are used as the target storage. In our case, where we have a specialized triple store as a back-end, a single trivial mapping has to be created for each predicate, with the exception of the rdf:type property, where a separate mapping has to be created for each distinct object, but this does not create problems as we will see later. Regarding the first reason, it is observed that exponential number of rewritings due to existential quantification seems to be rarely observed in real world ontologies and queries, leaving the second reason to be the most commonly encountered, as it is demonstrated in the following example. Consider the following OWL statements:

```
AssistantProfessor rdfs:subClassOf Professor
FullProfessor rdfs:subClassOf Professor
teaches rdfs:domain Professor
hasBScDegreeFrom rdfs:subPropertyOf degreeFrom
hasMScDegreeFrom rdfs:subPropertyOf degreeFrom
hasDoctoralDegreeFrom rdfs:subPropertyOf degreeFrom
FullProfessor rdf:subClassOf _R1
_R1 owl:onProperty hasDoctoralDegreeFrom
_R1 owl:someValuesFrom University
```

The first (resp. second) OWL statement declares that if an individual is an assistant professor (resp. full professor), then he is a professor. The third statement declares that if an individual teaches something, then he is a professor. The next three statements declare that if an individual has a Bsc, Msc or doctoral degree from a given entity, then this individual has (in general) a degree from this entity. Finally, the last three statements encode the knowledge that if an individual is a full

professor, then there exists a university such that this individual has a doctoral degree from this university. In DL parlance this is denoted as: $FullProfessor \sqsubseteq \exists hasDoctoralDegree.University$. The first three statements correspond to three axioms defining a class hierarchy, the next three statements correspond to three axioms defining a property hierarchy, whereas the last three statements correspond to an axiom involving existential quantifier. Following [43], the main observation upon which we base the architecture of our approach is that if the underlying data are (virtually or actually) complete with respect to class and property hierarchies, then during query rewriting all ontology axioms related to these hierarchies (in our example the first six statements) can be ignored, and we can use any rewriting method (tree-witness rewriting [24] is used in our case) for OWL2 QL ontologies to perform query rewriting with respect to the remaining axioms. Then, the produced rewriting is transformed to the query language of the underlying query execution engine using R2RML [6] mappings.

As Ontop is designed to work with R2RML mappings to an external relational database, we provide to it an abstraction of PARJ consisting of relational tables according to the vertical partitioning schema as presented in Section 3. In other words, we present each property as a relational table with two columns (subject and object) and provide SQL mappings over these tables, whereas in reality the underlying storage schema is the one described in Section 3. During system startup we automatically create the following mappings, without any manual user intervention, based on the data stored in PARJ:

- For each distinct property P defined in the ontology we add the mapping
  $P(x, y) \leftarrow$

  ```
  SELECT d1.value as x, d2.value as y
  FROM propI, dictionay d1, dictionary d2
  WHERE propI.subject=d1.id and
  propI.object=d2.id
  ```

  where $propI$ is the table corresponding to property P
- For each distinct named class C in the ontology we add the mapping
  $C(x) \leftarrow$

  ```
  SELECT d.value as x
  FROM propT, dictionary d
  WHERE propT.object=N and
  propT.subject=T.id
  ```

  where $propT$ is the table corresponding to the $rdf : type$ property and N is the integer value that corresponds to class C. During startup we prefetch these values for all the named classes of the ontology.

In the abstraction of the PARJ schema that it is provided to Ontop, there is no distinction between the S-O and O-S table replicas. The exact access methods will be decided internally by PARJ after the final rewriting has been produced. Also, as

[6] https://www.w3.org/TR/r2rml/

```
Professor(x) ← SELECT subject as x
FROM PropT where object=t1
/*t1 corresponds to Professor*/
Professor(x) ← SELECT subject as x
FROM PropT where object=t2
/*t2 corresponds to FullProfessor*/
Professor(x) ← SELECT subject as x
FROM PropT where object=t3
/*t3 corresponds to AssistantProfessor*/
Professor(x) ← SELECT DISTINCT subject as x
FROM PropN
/*table PropN corresponds to property teaches*/
```

Fig. 2: $\mathcal{T}$-Mappings entry for class Professor

the id column of the $dictionary$ table is unique, it can be considered a primary key, with both columns $subject$ and $object$ of all property tables to be foreign key referencing this primary key. As a result, when joining subject or object values obtained from different mappings in a query, joins can be performed directly on IDs instead of URIs, and only use the dictionary tables for lookups on the IDs that exist in the SELECT clause of the final query. For ease of presentation, in what follows we omit explicit references to the dictionary table. After these initial mappings have been declared, Ontop compiles knowledge about ontological hierarchies into the mappings in order to obtain the so-called $\mathcal{T}$-Mappings. Given the ontological axioms from the previous example, for the class $Professor$ three extra mappings will be added to the $\mathcal{T}$-Mappings, one containing as body the SQL query corresponding to the initial mapping for $AssistantProfessor$, one for the class $FullProfessor$ and one for the property $Teaches$, leading to four mappings in total, including the initial mapping for class $Professor$, as shown in Figure 2. Essentially, in order to obtain all professors from the data according to the property hierarchy, one should take the union of the four SQL queries in the body of the mappings (including duplicate elimination in the final result). In this case, for the first three mappings, two OR conditions could be introduced in the $WHERE$ clause, instead of union.

5.2 Union Wrappers for Ontology Hierarchies

As an end-to-end example regarding query rewriting, consider the following query from Example 4 over the ontology, asking for individuals x that work for an entity u and these individuals are professors and have a degree from the same entity u:

Example 4 Consider the following query, that contains an extra filter:

```
SELECT ?x ?u
WHERE {
?x worksFor ?u.
?x rdf:type Professor .
```

```
?x degreeFrom ?u.  }
```

First, the query will be rewritten with respect to axioms not involving hierarchies. In our case the result of this process will be the query unchanged, as the relevant axiom invovling existential quantifier is not applicable. Then, the query will be unfolded with respect to the $\mathcal{T}$-Mappings. One way to obtain the final query is to perform the union of each query atom first, and then perform the joins on these intermediate results. On the other hand, one could choose to flatten the queries by pushing the joins inside the unions. In our example, with four mappings for $Professor$, four mappings for $degreeFrom$ and one mapping for $worksFor$, this would result to a union of 16 queries. Finally one could follow a hybrid approach, based on some cost estimation. When a relational back-end stores the data, all these approaches can be highly inneficient. In the first case, a separate union with duplicate elimination should be performed for each atom with more than one mapping, and temporary results should be created and possibly indexed in order to perform subsequent joins efficiently. On the other hand, in the second approach the number of subqueries can be very large, with common tasks (for example a table scan) performed many times. Furthermore, duplicate answers coming from different subqueries lead to redundant processing and also a duplicate elimination must be performed to the final result. Having PARJ as the back-end, we chose to follow the first approach, having implemented a virtual union wrapper operator that eliminates the mentioned disadvantages, and at the same time it keeps the advantages of the standard PARJ design. Specifically out approach: i) performs pipelined union along with duplicate elimination ii) uses the original index (sort order) and adaptively decides for the access method iii) provides results in sorted order for subsequent operators as much as possible and iv) can be parallelized efficiently.

The main idea is to create a *union wrapper* for each class or property hierarchy encountered in a query. This wrapper acts as a virtual table that contains the complete answers for the given predicate. As an example, the union wrapper for the query atom $Professor$ is shown in Figure 3. This wrapper, as all wrappers that correspond to a class hierarchy, defines a virtual table with only one column and it provides two operations: i) scan the virtual table and return an iterator with the ordered distinct values that it contains and ii) search for a specific integer value and return it if it is contained in the virtual table, otherwise return an empty answer. The first operation is used when the union wrapper is the leftmost table in query execution according to the join order, whereas the second operation is used in all other cases. In our example the union wrapper contains four input vectors: the subject vectors of the O-S replica of the rdf:type property for the objects that correspond to values Professor, AssistantProfessor and FullProfessor and the subjects for the S-O replica of property $teaches$. We use the S-O replica of $teaches$ that contains the distinct subjects, as $Professor$ is defined to be the domain of the $teaches$ property. If it was the range of the property, the distinct objects array of the O-S replica would be used instead. During scan, all input vectors are scanned, and current value for each one is sent to a min heap implemented with a priority queue, that sorts them and outputs the minimum value. Only the first time a value is encountered it is sent to output, in order to have distinct results. When the minimum value of the min heap is being sent to output (or discarded)

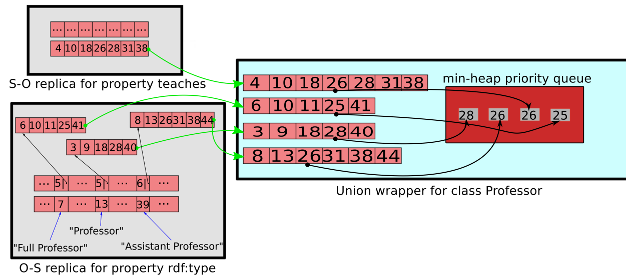

Fig. 3: Union Wrapper for Class Professor

the input vector that provided that value sends the next one. In the case of search, all input vectors are searched using the adaptive search algorithm from the previous section. Search stops as soon as the value is found to at least one of the input vectors and the given value is sent to output terminating the operation. In case of search, an alternative can be more efficient depending on the number of input vectors that correspond in the rdf:type property. For the specific example, the first three vectors can be replaced by the object vector of the S-O replica of the rdf:type property for the subject that corresponds to the value we are searching for and perform set intersection between this vector and a vector containing the values corresponding to Professor, AssistantProfessor and FullProfessor. This way we can avoid three different searches and only perform one search in the form of merge join (assuming that we first sort these three values) that terminates upon outputting the first result.

In case of the union wrappers for property hierarchies, such as the degreeFrom, the virtual table contains two columns and are three different operations: i)scan the whole table and provide an ouptut sorted on both columns, ii)search for a specific value in the first column and provide the output corresponding to this value sorted on the second column and iii) search for a specific pair of values. In the first case, we use a min heap with two input values sorting output to both, in the second case we search the input tables for the specific value in the first column and we use a min heap with one value, and in the last case we search the input tables for both values and stop the operation upon finding the first such result.

Regarding intra-query parallelism, incorporating the union wrappers does not require modifications to the approach described in Section 3, with the exception of scanning a union wrapper table in case it is the leftmost table according to join order. In this case, special effort is needed in order to ensure that the same values residing in different input tables will be produced from the same thread, so that duplicate elimination in the min heap will work properly, otherwise duplicate answers may be present in results of different threads. As a solution, we modify the way we assign a different portion of the first array of each input table to each thread. Instead of using positions in the array, we assign specific value intervals in each thread, based on the minimum and maximum values encountered in all input tables.

5.3 Join Ordering and Selectivity Estimation

As in RDF-3X and TriAD, we employ a bottom-up dynamic programming optimizer. As the level of parallelism during execution is determined by the number of threads, we assume that the benefit of each possible join order from parallelism will be a fixed proportion of its centralized cost, that is the execution cost if we consider that each property is consisting of a single shard. As a result of this assumption, we disregard parallelism during optimization. During cost estimation, we assume that a specific choice will be followed for all tuples of a join, either binary search or scanning. The latter will only take place when the join inputs are already fully sorted and it is estimated to be cheaper than binary search. Adaptivity during execution is expected to give a cost equal or lower to this estimation. For each property of a specific join order we choose to use the replica that leads to more efficient search. When we search for a specific subject or object, the choice is straightforward. When we are searching for both values, we use the replica that searches for ordered values according to the previous join. When we scan a table (leftmost table in the join order) we use the replica tha provides values sorted for the subsequent join. In union wrappers for property hierarchies, for some of the input tables we should use the S-O replicas and for others the O-S replicas. This decision is defined by the combination of two factors. First, whether we search for the subject or object of the given property and second, if the inverse of a subproperty is used in the definition of the hierarchy.

Regarding selectivity estimation, in order to estimate the sizes of intermediate results we use equi-depth histograms for each property. Such histograms are also built for the union wrappers, based on information about the hierarchies that can be found in the ontology. The reason for this is that the exact size of tuples in a union wrapper can be very difficult to estimate, as the common tuples in the different input tables that introduce duplicate results are discarded. Consequently, the final size can vary significantly and this can lead to very poor execution plans. In the two extremes, the final result size can be the sum of all input tables (no duplicates at all), or can be equal to the size of the larger input table. For this reason we take a sample and estimate the number of distinct tuples that will be produced from each union wrapper. These histograms are built after the initial data loading, and similar to the analyze command in RDBMSs they do not need to be recomputed, unless the underlying data are subject to significant change. As it is known that often estimates based on such histograms may not be accurate especially in the case of RDF data [35], we precompute some cardinalities between pairs of properties during data loading and use these as a corrective step. Specifically, for each pair of properties $prop_i$ and $prop_j$ we compute the cardinality of $prop_i \bowtie_{subject=subject} prop_j, prop_i \bowtie_{subject=object} prop_j, prop_i \bowtie_{object=subject} prop_j$ and $prop_i \bowtie_{object=object} prop_j$. All these computations can be done in parallel and also, using our storage schema, we do not need to access the second array of storage at all, as we only need the number of objects for joins on subjects and the number of subjects for joins on objects. We plan to implement more elaborate techniques for cardinality estimation in the future, like for example estimations based on characteristic sets [34] or RDF data summaries [51].

6 Experiments

In this section we present an experimental evaluation of our approach. One first objective is to compare both versions our system (PARJ and PARJ-Ontop) with other centralized and distributed state of the art systems of similar functionality, in terms of query execution time. Furthermore, we aim to investigate the scalability of PARJ by performing experiments with varying number of datasets and threads, and also examine the effect of the ID-to-Position index during query execution. Finally, we want to empirically evaluate the effect of the adaptive query processing method in comparison with standard binary and sequential search

In-memory data storage and query processing for PARJ have been implemented in C as an extension of a SQLite, which is used as disk-based storage. Disk-based tables are created and saved during data import from RDF files. On application start-up the in-memory data structures are created reading from the tables. The dictionary can either be loaded in memory or kept in disk where for IRI-to-ID transformation (during query optimization) a clustered B+ tree on IRI is used and for ID-to-IRI transformation (during IRI construction of answer tuples) a clustered B+ tree on id. PARJ is called through a wrapper written in Java, where also query parsing and optimization is implemented.

All experiments were conducted on a 16-core server with Intel E5-4603 processors at 2.20 GHz and 128 GB RAM running Debian 8. We used the popular Lehigh University Benchmark (LUBM) [17] and Waterloo SPARQL Diversity Test Suite (WatDiv) [6] benchmarks, as well as the real-world YAGO dataset [19] which contains data from Wikipedia, WordNet and GeoNames. Our implementation of PARJ is publicly available and open-source, and all material required to reproduce the experiments is available online [7].

6.1 Setup

We have carried out experiments with both the stand-alone version of PARJ, which is not capable of reasoning, and also with our PARJ-Ontop implementation which is able to perform OWL2-QL query ansering. Results for the latter version are presented in section 6.2.4. Regarding the stand-alone version, we compare PARJ to other systems that do not support OWL2-QL reasoning in two sets of experiments: in the first one we test the efficiency of our approach in the single-thread setting. In this setup we use as competitors the in-memory RDF store RDFox (SVN version: 2776) and also RDF-3X [35] (version 0.3.8) for comparison with a state of the art disk-based system. The second setup is about multi-threaded execution. In the second setup we use as competitor the TriAD system which in [18] it is shown to outperform all competitors in the centralized parallel setting. We have used the optimized build for TriAD, as it is suggested in the installation manual.

Due to a hard-coded limit in the TriAD source code, we could not execute queries using more than 20 workers[8]. Note that in PARJ, each worker corresponds exactly to

[7] https://github.com/dbilid/experiments

[8] This was verified with the TriAD implementors

one thread, so given that hyper-threading is enabled, we found that the optimal performance was achieved when we used two threads for each processing core, resulting in 32 workers/threads in our testing machine. More details regarding the behavior of PARJ for different number of threads are given in Section 6.2.3. For TriAD it was not clear which number of workers should be the optimal, as this could be query depended. This is also the reason that we do not use TriAD in the single-thread setting. To have a better image and find the optimal setup, we executed TriAD with different number of workers, and we also modified the hard-coded limit and tried with up to 32 workers. For most queries, TriAD performance is degrading for more than 20 workers. From our testing we found that the overall best performance was achieved for 16 workers and this is the setup we used for TriAD in our experiments. Also, we present results for both TriAD settings: with summary mode enabled and disabled. For summary mode, we used the same number of partitions used in [18]: 200K for LUBM 10240 and 70K for WatDiv 1000.

Regarding result handling, as our intention is to concentrate in join processing, all systems were tested in the so called ”silent“ mode, that is we do not include the time for dictionary lookups and result tuple construction. In multi threaded execution this also means that we do not measure the time to aggregate the results together. Each query was executed 10 times and the average execution time is shown. We have deployed RDF-3X using an in-memory filesystem and as a result there is no need to report cold and warm cache times.

6.2 Results

We present results for scale 10240 of the LUBM benchmark in Table 2 (about 1.4 billion triples), for YAGO2 in Table 3 and for scale 1000 of the WatDiv benchmark (about 110 million triples). For WatDiv we used both basic test workload (Table 4) and incremental linear and mixed linear extensions of basic workload (Table 5). For WatDiv we generated all the queries proposed in the workloads. For LUBM we used the seven queries commonly used to test systems that do not perform reasoning tasks, which can be found in [58], and are labeled LUBM1-LUBM7, and we also used three extra queries from [39] (LUBM8-LUBM10). A timeout of 30 minutes was used for all queries. For YAGO2 we used the same raw data (about 285 million triples) and queries as in [2].

Regarding single thread execution, we first observe that RDFox is comparable to PARJ for some queries, but for other queries, especially for queries from the WatDiv incremental and mixed linear extensions, is highly inefficient. This confirms that this system is not optimized for query answering, but instead, it aims at efficient parallel materialization of RDF implications. Regarding RDF-3X, we can see that it performs more than one order of magnitude slower from PARJ for most queries. The reason is that despite the fact that it is deployed in an in-memory filesystem, its processing is oriented towards optimizing disk access, as it is not aware that it operates in memory. For example, it uses B+ trees to minimize the number of disk pages needed, it skips records with its sideways-information passing optimization only when it reads a new disk-page into memory, it uses compression on a per page basis and also its cost

estimation is based on disk access. Nevertheless, there are some queries, for example queries in the ML-2 set or LUBM8, where RDF-3X outperforms the single-threaded PARJ execution. These are queries with large intermediate results, but only few final answers, where the record skipping using sideways information passing in RDF-3X results in substantial gains.

Regarding multi-thread execution we can see that for most queries the summary mode of TriAD is inferior to the simple mode, sometimes by a large margin. For example, for query LUBM 3 in Table 2 the execution time increases from 2 seconds to more than 15 seconds. For the specific query we saw that execution over the summary graph takes up most of the execution time. In any case, the results show that for parallel execution on a centralized environment the pruning from the graph summaries does not contribute to an important improvement which can justify the overhead of graph partitioning.

A comparison of PARJ with the best TriAD mode shows that we outperform TriAD by more than an order of magnitude for the average execution time of the LUBM 10240 queries: from 838 milliseconds for PARJ to 13263 for TriAD (Table 2). For basic WatDiv testing (Table 4), though TriAD performs slightly better for simple queries, PARJ performs better overall with a total average execution time of 11.27 ms (geomean: 7.76) whereas TriAD has a total average execution time of 13.95 (geomean: 6.8). For the more complex queries of WatDiv extended workloads (Table 5) PARJ clearly outperforms TriAD. For some queries the difference is more than two orders of magnitude. As an example, for query ML1-7 the time increases from 7 ms to 2154. The specific query contains a series of subject-object joins, which leads TriAD to perform blocking data transfers between workers and rehashing over large intermediate results, though the final result is relatively small.

Regarding the difference between the silent mode and the full result handling, we have executed all queries with full result handling (except from printing) in PARJ. That is we include answer tuple construction, dictionary lookups and sending all results to the coordinating thread. We do not include these results, as we saw that for most queries, usually with results up to a few thousand tuples, the difference is not important, but for queries with many million results the difference can be significant. This can be seen especially for query 2 from the LUBM benchmark (about 10M results) where execution time in multi threaded execution increases from 151 milliseconds in silent mode to 610 milliseconds in full result handling. The same holds for queries C3 (about 4.3M results) and IL-3-5 to IL-3-10 from WatDiv which have more than 50M results. Query IL-3-8 has by far the largest number of results (about 1.6 billion tuples with 9 columns). This is the reason that TriAD runs out of memory for the specific query, since even in silent mode, each worker keeps in memory all the results instead of using an iterator to send the results to the master (or discard the results in silent mode) as they are produced, as it is the approach used by PARJ. Execution times for the full result handling mode of PARJ are included in the online material to reproduce experiments.

Table 2: Results for LUBM 10240 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
LUBM1	15369	96677	1329510	800	4188	4467
LUBM2	2437	40368	21870	151	965	1101
LUBM3	5338	136554	23179	605	2004	15243
LUBM4	5	1	8	10	12	5
LUBM5	1	1	6	4	2	2
LUBM6	3	3	190	5	95	5
LUBM7	9213	31180	68769	473	13400	14125
LUBM8	9899	44144	6485	1336	2838	3906
LUBM9	58082	187192	208839	4014	42932	32982
LUBM10	14606	26690	51235	982	65925	41510
Avg	11495	56281	171009	838	13263	11334
Geomean	864	2536	5581	180	1071	881

Table 3: Results for YAGO2 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
Y1	9	56	102	11	12	8
Y2	11	2390	380528	13	830	1381
Y3	165	1409	2915	20	280	137
Y4	5	20	110	10	9	3
Avg	48	969	95914	14	283	382
Geomean	17	248	1878	13	71	46

6.2.1 Effect of Runtime Join Optimization

In order to examine the effect of our adaptive join method, we have executed the queries of both datasets using four different strategies as shown in Table 6. For WatDiv benchmark we only report the average and geometric mean of all execution times. In the first (Binary) column we report the execution times when we always use binary search. In the second column (AdBinary) we use our adaptive join method in order to switch from binary to sequential search. In third column (Index) we always use the ID-to-Position index, whereas in the last column (AdIndex) we use the adaptive join method in order to switch from ID-to-position index to sequential search. One can observe that the impact of the adaptive join method is more important when binary search is employed (comparison of first and second column), whereas when the ID-to-Position index is used (comparison between third and fourth column) its contribution to better performance is smaller. This is in line with the result of our calibration method, where when binary search is used, the result threshold is about 200 positions, whereas when ID-to-Position index is used the threshold is about 20 positions. Also, it seems that the impact is more important for LUBM queries, where in case of binary search it leads to a decrease of 23% in average execution time. The reason for that is that the average execution time for WatDiv queries is heavily affected by the IL-3 queries, where the impact of the adaptive method is not important, as sequential search can rarely be used in these queries. This is also the reason for the

Table 4: Results for WatDiv Basic Workload scale 1000 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
L1	5	5	40	10	3	5
L2	8	43	30	5	5	6
L3	2	244	13	4	2	3
L4	3	7	19	4	2	8
L5	9	57	40	6	3	46
Avg	5	71	28	6	3	14
Geomean	5	29	26	5	3	8
S1	49	1209	18	47	34	116
S2	3	284	27	3	4	17
S3	4	17	7	3	2	18
S4	4	153	10	5	5	29
S5	4	1*	14	4	4	20
S6	1	5	8	5	2	3
S7	1	695	7	5	2	3
Avg	9	338*	13	10	8	29
Geomean	4	61*	12	6	4	15
F1	5	24	15	6	5	19
F2	12	153	27	10	37	13
F3	3	59	73	9	29	74
F4	56	249	83	19	9	66
F5	3	10	108	7	40	58
Avg	16	99	61	10	24	46
Geomean	8	56	48	9	18	37
C1	21	50	140	12	39	598
C2	76	178	441	16	40	1574
C3	266	4810	127	45	43**	527**
Avg	121	1679	236	24	41**	900**
Geomean	75	350	199	21	41**	792**

* RDFox returns an empty result-set for query S5, whereas the correct answer is not empty.
** TriAD returns only distinct answers for query C3, even though modifier DISTINCT is not present in the SPARQL query. The number of results returned is only 8162 instead of 4335801.

great reduction in average execution time of WatDiv queries when the ID-to-Position index is used, as the aforementioned queries are greatly profit from the index.

6.2.2 Effect of ID-to-Position Index

We now proceed to describe the evaluation of our ID-to-Position Index compared to standard binary search using the LUBM 10240 dataset in the single-thread setting. Table 7 shows the number of binary searches and the number of sequential searches which were performed using the decision of our adaptive join method, using a threshold of about 200 computed with our calibration algorithm. The fact that sequential searches heavily outnumber binary searches provides a strong indication that ordering is present in the RDF dataset. In order to compare our index with binary search, we kept the threshold the same as computed in the case of binary search, and executed the queries by performing our index based lookup instead of binary search,

Table 5: Results for WatDiv Incremental and Mixed Linear Workloads scale 1000 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
IL-1 5	3	27617	1339	5	584	5082
IL-1 6	4	204898	1832	4	1482	11814
IL-1 7	8	669099	1272	7	1862	14950
IL-1 8	3	700199	1633	5	1615	21238
IL-1 9	26	728518	1396	11	630	23844
IL-1 10	29	734363	1923	9	618	25752
Avg	12	510782	1566	7	1132	17113
Geomean	8	335194	1546	6	1002	15068
IL-2 5	2	6574	1525	6	476	5340
IL-2 6	5	62149	2046	4	952	11156
IL-2 7	2	78211	1794	3	344	58749
IL-2 8	4	80453	1865	16	1148	62448
IL-2 9	9	86995	1998	6	1062	67045
IL-2 10	4	87872	1867	5	1093	70658
Avg	4	67042	1849	7	846	45899
Geomean	4	51948	1841	6	770	31807
IL-3 5	13259	187101	542948	1494	11195	17093
IL-3 6	58379	397964	357310	7070	13603	25492
IL-3 7	23208	342533	Timeout	1192	1809	23492
IL-3 8	71918	1214564	Timeout	4903	Out Of Memory	Out Of Memory
IL-3 9	26437	966919	Timeout	2082	7182	39462
IL-3 10	41867	951513	175247	1882	8118	46593
Avg	39178	676766		3104		
Geomean	33565	552681		2496		
ML-1 5	2	11481	163	2	56	374
ML-1 6	2	2	83	2	33	1152
ML-1 7	1	1	728	7	2154	4646
ML-1 8	2	1	824	4	103	2018
ML-1 9	5	98058	994	4	198	11766
ML-1 10	4	14111	1482	3	930	9841
Avg	3	20609	712	4	579	4966
Geomean	2	178	478	3	206	2786
ML-2 5	3175	1136335	936	201	413	1849
ML-2 6	2	12182	166	5	92	1041
ML-2 7	121	27151	678	15	296	895
ML-2 8	69	818424	2863	19	1996	24500
ML-2 9	4335	919541	282	259	330	1587
ML-2 10	52	849283	1952	9	728	32449
Avg	1292	627153	1146	85	643	10387
Geomean	151	249327	741	30	419	3599

measuring the exact number of total execution cycles used in the index lookup or binary search procedure each time, as well as the cache misses for each cache level. If we exclude queries no 1 and 3-6, as they nearly perform only sequential searches, we can see that our ID-to-Position index results in more than 30% decrease in total execution cycles and similar or larger decrease in the number of cache misses for all levels of cache hierarchy.

Table 6: Impact of Adaptive Processing for LUBM 10240 and WatDiv 1000 (times in ms) for 1 thread

Query	Binary	AdBinary	Index	AdIndex
LUBM1	22186	15454	16557	15369
LUBM2	2877	2443	2535	2437
LUBM3	6562	5491	6415	5338
LUBM4	5	7	7	5
LUBM5	1	1	1	1
LUBM6	2	2	2	3
LUBM7	12246	11866	9197	9213
LUBM8	15725	9782	10420	9899
LUBM9	77468	63586	58171	58082
LUBM10	22359	14892	16217	14606
Avg	15943	12352	11952	11495
Geomean	1034	892	898	864
Watdiv1000 Avg	8439	8003	5013	4869
WatDiv 1000 Geomean	33	28	25	23

Table 7: Number of binary searches and sequential searches for LUBM10240 chosen by out adaptive join method and comparison of binary search with ID-to-Position index with respect to total execution cycles and L1, L2 and L3 cache misses

Query	#Binary	#Sequential	Binary Search				ID-to-Position Index			
			Cycles	L1 M	L2 M	L3 M	Cycles	L1 M	L2 M	L3 M
LUBM1	1	107525748	2236	130	49	9	3135	102	43	8
LUBM2	204795	10854018	502M	26.7M	10.8M	3.5M	355M	18.3M	4.4M	543K
LUBM3	1	33169741	2401	140	50	8	4175	139	42	3
LUBM4	4	68	38745	666	368	235	16862	469	182	34
LUBM5	1	10	2423	94	29	0	2395	162	83	5
LUBM6	1	570	2033	106	26	0	2003	130	48	0
LUBM7	2257238	28768005	2.95B	254M	80.1M	2.30M	2.12B	211M	58.9M	1.08M
LUBM8	8645	84755793	17.4M	1.20M	682K	84.1K	11.2M	841K	351K	21.7K
LUBM9	409590	351307982	1.06B	53.6M	19.7M	2.92M	655.7M	39.1M	11.18M	639.7K
LUBM10	558279	116015419	1.22B	66.7M	24.2M	2.98M	798.2M	50.76M	12.7M	634.3K

6.2.3 Scalability

In this section we experimentally show the scalability of PARJ with regard to a varying number of threads and varying dataset size. As far as the first issue is concerned, we can already observe from Section 6.2 and specifically from Tables 2, 4 and 5, that running PARJ in multi threaded mode with 32 threads performs on average about 15 times better than the single thread version, but for the simple queries, when execution time is less than few tens milliseconds, multi-threaded execution does not seem to provide important gains. There are two reasons for that. The first one is the overhead of spawning multiple threads and the second is that query parsing and optimization take up a large fragment of the total execution time, which cannot be avoided in multi-threaded execution. The best example of this is query S1 from WatDiv benchmark which is a star join query with 9 triple patterns and more than 40 milliseconds of the reported time of 49 milliseconds is spent on producing the join order in the optimizer.

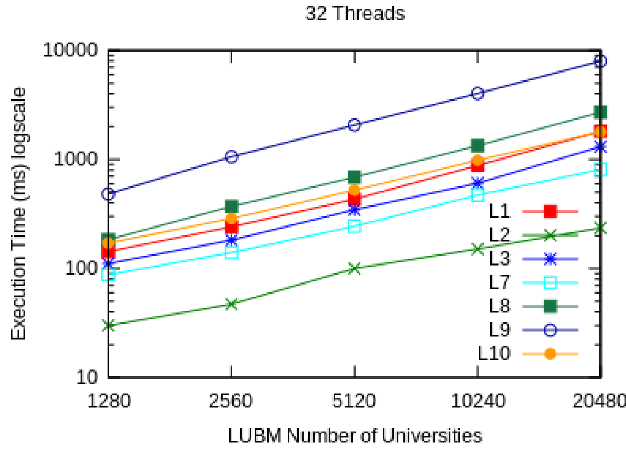

Fig. 4: LUBM 32 threads execution times in ms for different dataset sizes

In order to better examine the behavior of PARJ for a varying number of threads we have executed the queries from LUBM benchmark for scale 10240 with 1, 2, 4, 8 and 16 threads as shown in Figure 5. We exclude from this presentation simple and very selective queries L4, L5 and L6 that do not appear to improve from parallelism, since already in the single-threaded execution their execution time is only a few milliseconds, much of which is due to query parsing and optimization. On the other hand, complex queries L1, L3, and L7-L10, and also the simple but not selective query L2 show large and nearly linear improvement. The reason that we do not show results beyond 16 threads in Figure 5 has to do with the capabilities of our testing machine, which has exactly 16 processing cores. As stated before, best results were obtained with 32 threads as hyper-threading was enabled, but improvement from 16 to 32 threads cannot be evaluated and interpreted reliably for the specific scalability experiment, as here we aim to examine the behavior of PARJ for a varying number of threads given that the underlying hardware can provide full processing resources to each thread.

We have also examined the scalability of our system for a varying dataset size. Findings in Figure 4 show a similar situation for a varying number of universities up to 20480 (about 2.83 billion triples) in the execution with 32 threads, confirming the excellent scalability of PARJ.

6.2.4 Results for Query Execution over OWL2 QL Ontologies

In this section we provide experimental comparison of our approach for query execution over OWL2 QL ontologies with other state of the art methods. We use the name PARJ$_{Ontop}$ for our prototype as described in Section 5. As in the stand-alone version of PARJ, this implementation is publicly available [9]. As main competitors we

[9] https://github.com/dbilid/PARJ-Ontop

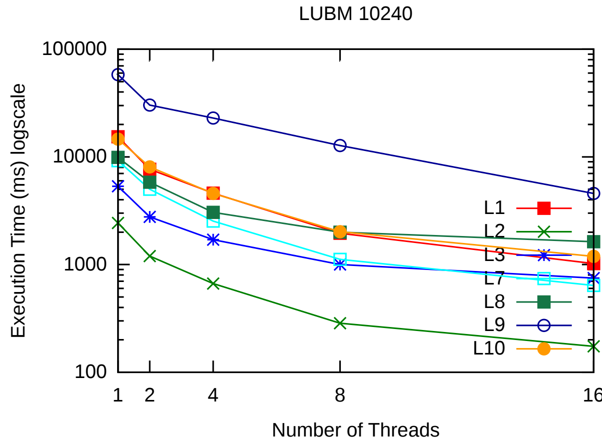

Fig. 5: LUBM 10240 execution times in ms for different number of threads

have used the semantic index mode of Ontop with PostgreSQL 10 as backend and a commercial RDF store providing support for OWL2 QL query answering via query rewriting, which we will call system A, as due to its license we cannot reveal its real name when providing experimental results. System A is a disk-based system, but for the experiments we have deployed its database in an in-memory filesystem. We have also tried to use a second commercial system which is based mainly on materialization, but for the dataset used in our experiments data loading was not completed even after 8 days. For the semantic index experiments we have used an older version of Ontop (1.12), as in latest versions the semantic index mode is not maintained. The dataset used in the experiments is from the LUBM$^{\exists}_{20}$ benchmark [28] for scale of 1000 universities and with incompleteness ratio 5%, which contains about 147 million triples and the raw data size in NTriples format is 25.5 GB. The queries are the same used in [43]. In contrast with the experiments described in previous sections, here, for all the systems, we include in the results the time needed to perform dictionary lookups and tuple construction. The reason is that we could not modify System A so as to exclude these features. PARJ$_{Ontop}$ was the only system that successfully executed all queries in a 30 minutes time limit per query, as q1 was timed out for semantic index, and q5 returned an error in System A. For the rest of the queries, even the single thread version of PARJ is on average about an order of magnitude faster than the other two systems. Regarding the overhead of union wrappers, we have executed all the queries with the hierachies materialized in PARJ, in order to estimate the impact they have on query execution. The average execution time for single thread execution decreased from 15554 milliseconds to 13790, with the overhead being less than 13%, even though all queries involved union wrappers of large hierarchies, some of them, like q7, with up to six different union wrappers involved in the same query.

Table 8: Results for LUBM$^{\exists}_{20}$ 1000 (times in ms)

	PARJ-1	PARJ-32	SI-PostgreSQL	SystemA
q1	197997	10619	TIMEOUT	673268
q2	3977	251	32477	2610
q3	2369	190	59530	15803
q4	3890	482	2749	47029
q5	4828	459	60490	ERROR
q6	5	11	25318	240
q7	467	70	19413	8730
q8	184	26	924	7853
q9	1	1	1	145
r1	1	1	1	151
r2	522	66	3049	525
r3	3170	390	26433	30745
r4	181	25	803	797
r5	170	62	1	1505

6.2.5 Comparison With Distributed RDF Stores

A comparison of a parallel centralized system with distributed systems is not straightforward, as many factors come into play in order to have a result that will be as fair and complete as possible. In this section we attempt some first comparison of PARJ with existing RDF stores based on a recently published survey [2] and we plan to further investigate this issue experimentally in the future. The aforementioned survey presents an experimental comparison of 12 distributed systems designed for shared-nothing clusters, chosen as the most competitive and innovative from a variety of approaches and characteristics. The experiments were performed on a cluster with 12 servers, each with 148GB of memory and 24 cores, using, among others, the LUBM 10240 (only queries LUBM1-LUBM7) and WatDiv 1000 (only basic workload) benchmarks. For both these benchmarks the single server results of PARJ (in the full result handling mode) are comparable with the faster of the reported systems which is the non-adaptive version of AdPart (the adaptive version is not included in the results of [2]). Specifically, the average and geometric mean of execution times for first seven queries of LUBM 10240 are 918 and 75 milliseconds respectively (compared with 419 and 103 for PARJ in full result handling mode) whereas the geometric means for the 4 query categories of the basic workload of WatDiv 1000 are 9, 7, 160 and 111 milliseconds (compared with 9, 10, 12 and 48 for PARJ in full result handling mode).

7 Conclusions and Future Work

We have presented a centralized in-memory system for parallelizing join processing on RDF graphs. We have shown that our design has excellent scaling capabilities and performance. For future work, we first plan to perform a more thorough experimental comparison with distributed RDF stores. As we mentioned, it is straightforward to extend PARJ to a “cluster” version through full replication, such that during query execution each worker start processing from different initial shard. We plan to imple-

ment and compare this version with the current state of the art distributed systems. We also want to further evaluate PARJ on a high-end server with larger available memory, in order to load and process larger RDF graphs. Based on the scaling capabilities presented during the experiments, we anticipate that our approach will be able to efficiently handle such datasets. Finally, we plan to explore the possibility of PARJ-Ontop acting as a mediator system for federating multiple data-sources in a data integration scenario, by importing on-the-fly external data, using canonical IRIs [56] for recognizing the same conceptual entity in different databases (endpoints), as it has been explored in real-world use cases [23].

Acknowledgments

The present work was funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 825258.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007, pp. 411–422 (2007)
2. Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P.: A survey and experimental comparison of distributed SPARQL engines for very large RDF data. PVLDB **10**(13), 2049–2060 (2017)
3. Al-Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M.: Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. VLDB J. **25**(3), 355–380 (2016)
4. Albutiu, M.C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. Proceedings of the VLDB Endowment **5**(10), 1064–1075 (2012)
5. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ICS-FORTH RDF-Suite: Managing voluminous RDF description bases. In: SemWeb (2001)
6. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I, pp. 197–212 (2014)
7. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: SIGMOD Conference, pp. 1383–1394. ACM (2015)
8. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.G.: DBpedia: A nucleus for a web of open data. In: The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007., pp. 722–735 (2007)
9. Bilidas, D., Koubarakis, M.: Scalable parallelization of RDF joins on multicore architectures. In: Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019, pp. 349–360 (2019). DOI 10.5441/002/edbt.2019.31. URL https://doi.org/10.5441/002/edbt.2019.31
10. Borovica-Gajic, R., Idreos, S., Ailamaki, A., Zukowski, M., Fraser, C.: Smooth scan: Statistics-oblivious access paths. In: Data Engineering (ICDE), 2015 IEEE 31st International Conference on, pp. 315–326. IEEE (2015)
11. Borovica-Gajic, R., Idreos, S., Ailamaki, A., Zukowski, M., Fraser, C.: Smooth scan: robust access path selection without cardinality estimation. The VLDB Journal pp. 1–25 (2018)
12. Bursztyn, D., Goasdoué, F., Manolescu, I.: Teaching an RDBMS about ontological constraints. Proceedings of the VLDB Endowment **9**(12), 1161–1172 (2016)
13. Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering SPARQL queries over relational databases. Semantic Web **8**(3), 471–487 (2017)

14. Chortaras, A., Trivela, D., Stamou, G.: Optimized query rewriting for OWL 2 QL. In: International Conference on Automated Deduction, pp. 192–206. Springer (2011)
15. Du, J., Wang, H., Ni, Y., Yu, Y.: HadoopRDF: A scalable semantic data analytical engine. In: Intelligent Computing Theories and Applications - 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings, pp. 633–641 (2012)
16. Groppe, J., Groppe, S.: Parallelizing join computations of SPARQL queries for large semantic web databases. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 1681–1686. ACM (2011)
17. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem. **3**(2-3), 158–182 (2005)
18. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, pp. 289–300 (2014)
19. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. Artificial Intelligence **194**, 28–61 (2013)
20. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. PVLDB **4**(11), 1123–1134 (2011)
21. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K.S., Kersten, M.L.: Monetdb: Two decades of research in column-oriented database architectures. IEEE Data Eng. Bull. **35**(1), 40–45 (2012)
22. Kaoudi, Z., Manolescu, I.: RDF in the clouds: a survey. The VLDB Journal **24**(1), 67–91 (2015)
23. Kharlamov, E., Hovland, D., Skjæveland, M.G., Bilidas, D., Jiménez-Ruiz, E., Xiao, G., Soylu, A., Lanti, D., Rezk, M., Zheleznyakov, D., et al.: Ontology based data access in statoil. Journal of Web Semantics **44**, 3–36 (2017)
24. Kikot, S., Kontchakov, R., Zakharyaschev, M.: Conjunctive query answering with OWL 2 QL. In: Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning (2012)
25. Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. Proceedings of the VLDB Endowment **2**(2), 1378–1389 (2009)
26. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to ontology-based data access. In: Twenty-second international joint conference on artificial intelligence (2011)
27. Luo, Y., Picalausa, F., Fletcher, G.H., Hidders, J., Vansummeren, S.: Storing and indexing massive RDF datasets. In: Semantic search over the web, pp. 31–60. Springer (2012)
28. Lutz, C., Seylan, I., Toman, D., Wolter, F.: The combined approach to OBDA: Taming role hierarchies using filters. In: International semantic web conference, pp. 314–330. Springer (2013)
29. Manegold, S., Boncz, P., Kersten, M.: Optimizing main-memory join on modern hardware. IEEE Transactions on Knowledge and Data Engineering **14**(4), 709–730 (2002)
30. Manegold, S., Boncz, P., Kersten, M.L.: Generic database cost models for hierarchical memory systems. In: VLDB'02: Proceedings of the 28th International Conference on Very Large Databases, pp. 191–202. Elsevier (2002)
31. Mora, J., Corcho, Ó.: Engineering optimisations in query rewriting for OBDA. In: Proceedings of the 9th International Conference on Semantic Systems, pp. 41–48. ACM (2013)
32. Myung, J., Yeon, J., Lee, S.g.: Sparql basic graph pattern processing with iterative mapreduce. In: Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud. ACM (2010)
33. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: A highly-scalable RDF store. In: International Semantic Web Conference, pp. 3–20. Springer (2015)
34. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: ICDE, pp. 984–994. IEEE Computer Society (2011)
35. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: SIGMOD Conference, pp. 627–640. ACM (2009)
36. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD Conference, pp. 1099–1110. ACM (2008)
37. Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., Koziris, N.: H_2RDF+: an efficient data management system for big RDF graphs. In: SIGMOD Conference, pp. 909–912. ACM (2014)
38. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. In: Journal on data semantics X, pp. 133–173. Springer (2008)

39. Potter, A., Motik, B., Nenov, Y., Horrocks, I.: Distributed RDF query answering with dynamic data exchange. In: International Semantic Web Conference (1), *Lecture Notes in Computer Science*, vol. 9981, pp. 480–497 (2016)
40. Punnoose, R., Crainiceanu, A., Rapp, D.: SPARQL in the cloud using Rya. Inf. Syst. **48**, 181–195 (2015)
41. Qin, W., Idreos, S.: Adaptive data skipping in main-memory systems. In: Proceedings of the 2016 International Conference on Management of Data, pp. 2255–2256. ACM (2016)
42. Ravindra, P., Kim, H., Anyanwu, K.: An intermediate algebra for optimizing RDF graph pattern matching on mapreduce. In: ESWC (2), *Lecture Notes in Computer Science*, vol. 6644, pp. 46–61. Springer (2011)
43. Rodriguez-Muro, M., Kontchakov, R., Zakharyaschev, M.: Ontology-based data access: Ontop of databases. In: International Semantic Web Conference, pp. 558–573. Springer (2013)
44. Rohloff, K., Schantz, R.E.: High-performance, massively scalable distributed systems using the mapreduce software framework: the SHARD triple-store. In: PSI EtA, p. 4. ACM (2010)
45. Rohloff, K., Schantz, R.E.: Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In: DICT@HPDC, pp. 35–44. ACM (2011)
46. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (2010)
47. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: mapping SPARQL to Pig Latin. In: SWIM, p. 4. ACM (2011)
48. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. PVLDB **9**(10), 804–815 (2016)
49. Schmachtenberg, M., Bizer, C., Paulheim, H.: Adoption of the linked data best practices in different topical domains. In: Semantic Web Conference (1), *Lecture Notes in Computer Science*, vol. 8796, pp. 245–260. Springer (2014)
50. Ślezak, D., Wróblewski, J., Eastwood, V., Synak, P.: Brighthouse: an analytic data warehouse for ad-hoc queries. Proceedings of the VLDB Endowment **1**(2), 1337–1345 (2008)
51. Stefanoni, G., Motik, B., Kostylev, E.V.: Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In: Proceedings of the 2018 World Wide Web Conference on World Wide Web, pp. 1043–1052. International World Wide Web Conferences Steering Committee (2018)
52. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-store: A column-oriented DBMS. In: VLDB, pp. 553–564. ACM (2005)
53. Subercaze, J., Gravier, C., Chevalier, J., Laforest, F.: Inferray: fast in-memory RDF inference. Proceedings of the VLDB Endowment **9**(6), 468–479 (2016)
54. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1), 1008–1019 (2008)
55. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF storage and retrieval in jena2. In: SWDB, pp. 131–150 (2003)
56. Xiao, G., Hovland, D., Bilidas, D., Rezk, M., Giese, M., Calvanese, D.: Efficient ontology-based data integration with canonical IRIs. In: European Semantic Web Conference, pp. 697–713. Springer (2018)
57. Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., Liu, L.: Triplebit: a fast and compact system for large scale RDF data. Proceedings of the VLDB Endowment **6**(7), 517–528 (2013)
58. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. PVLDB **6**(4), 265–276 (2013)