Scalable Parallelization of RDF Joins on Multicore Architectures

Dimitris Bilidas
National and Kapodistrian University of Athens
Greece
d.bilidas@di.uoa.gr

Manolis Koubarakis
National and Kapodistrian University of Athens
Greece
koubarak@di.uoa.gr

ABSTRACT

The RDF data model has emerged as the most prominent way to interlink and exchange data on the Web due to its simplicity in the form of subject predicate object statements, but this simplicity comes with the cost of having to execute a large number of joins in order to get the desirable query results. Numerous approaches exist that aim to treat this problem, mainly focusing on disk based storage. In this work we consider a main memory setting and present a physical design and query method aiming to exploit spatial locality for efficient in-memory processing. Our design is also amenable to straightforward parallelization, something crucial for main memory database systems. 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. We have implemented an in-memory prototype that experimentally shows the efficiency and scalability of our proposal, taking advantage of continuously growing sizes of main memory and multi-core environments of modern hardware. Specifically, we show that for a machine with 128 GB of main memory and 16 cores, which is a reasonable amount for an average modern server, our prototype can store and query RDF graphs with up to two billion triples, and it outperforms centralized and distributed state of the art approaches.

1 INTRODUCTION

The *Resource Description Framework* (RDF) 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 2018 the *Linked Open Data* (LOD) cloud [34] contains more than 1000 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.

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 [42] is based on graph exploration and it was the first system to follow this design. TriAD [14] and the extension of the centralized main memory RDF store RDFox with a dynamic data exchange operator [26] also use an asynchronous execution model (In what follows we will refer to the system described in [26] 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.

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 message exchanging. Our method ensures parallel execution 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.

Regarding the physical data storage, our approach is inspired by *column-store* systems such as MonetDB [16] and C-Store [37], 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 way we achieve increased spatial locality during processing. 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 [39]) and we keep two replicas of each two-column table in different sort orders. The main contributions presented in this work are:

- A join processing approach with low memory consumption, able to efficiently parallelize evaluation of arbitrary multi-join BGPs without any communication.
- Physical design method which 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).
- A cache-friendly method which adaptively, during execution, decides to switch from binary search to scan in order to 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.
- An implementation and experimental evaluation for datasets up to 2 billion triples which shows that our proposal outperforms existing state of the art centralized systems. Also, based on published results for other systems, it is shown that for tested datasets, like LUBM 10240, our implementation running on a single 16-core server outperforms (mostly for complex queries) or performs close to the fastest state of the art asynchronous distributed in-memory systems deployed on a cluster of machines.

We proceed by first presenting related work. Then in Section 3 we present details of the physical data storage and in Section 4 we present details of the adaptive join method that allows for incorporating parallelism into processing. We present implementation details and experimental evaluation in Section 5 and we finally conclude and discuss future work.

2 RELATED WORK

RDF storage using relational technology has been a subject of research since the proposal of the RDF data model. 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 [40] 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.

Hexastore enhances the vertical partitioning by replicating the data through six different indexes, corresponding to all possible permutations of subject, predicate and object [39]. RDF-3X [23] 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. Our design follows the vertical partitioning approach, but as in Hexastore, we keep two different replicas for each property in different sort orders (corresponding to POS and PSO indexes) 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, an initial approach using the MapReduce framework is presented in [30, 31]. 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 [20], 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 [11] 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 [15] 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, [32] uses Pig Latin[24] 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, [32] 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 [29]. 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+ [25] uses HBase[1] 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[27] uses Accumulo[2], 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 [33] 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 [18], and also into taking advantage of hardware features such as the SIMD vectorized instructions for efficient parallel sort-merge joins [4, 17]. 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 [28, 35] where additional statistics about each such zone have to be maintained in order to skip scanning certain areas. Adaptivity during run-time regarding the decision of scanning a base relation or use a secondary index has been studied in [9, 10] 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 [21] and Inferray [38] 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 [12] 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 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.

3 PHYSICAL DATA STORAGE

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
```

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

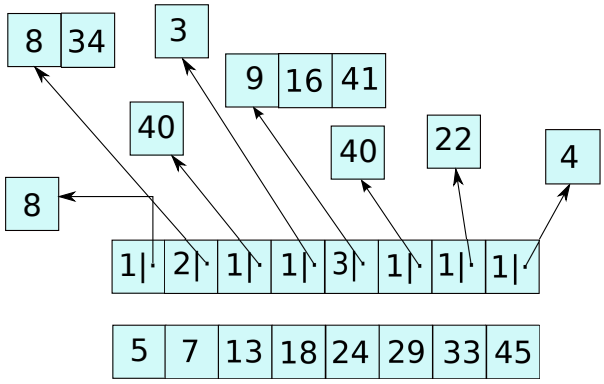

Figure 1: Example of Physical Data Storage for a Property Partition

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. Given that the specific table is 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 join method resembles an index-based nested loops join (or merge join when possible - this will be discussed later) that starts concurrently from different shards of the first table, and runs in parallel, by probing the next table to be joined for each tuple. In this way our method operates on left-deep query join trees as shown in Example 3.1.

Example 3.1. 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 4.3) 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 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 3.2.

Example 3.2. 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 [41] 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 5) 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 5.2.3), 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 4.1. 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, *prop*2 will be probed for values sorted on $?x$, but for the second join, probing *prop*3 will not in general be sorted on $?z$. Nevertheless, for each distinct $?x$, *prop*3 probing will still be sorted on $?z$ and if each subject of *prop*3 is connected to many objects, it may be more efficient to avoid binary search on *prop*3 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 *Sequential_Search* and *Binary_Search* functions.

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 [19], 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 $CursorPosition$ instead of 0 as starting position, 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 estimated to be at distance equal to $CurrentWindowSize$ from the previous one. 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.

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
```

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 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

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;
```

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. 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 5, 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.

4.3 Join Ordering and Cost 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 selective results.

As selectivity estimation is not the focal point of this work, currently, in order to estimate the sizes of intermediate results we use equi-depth histograms. As it is known that often estimates based on such histograms may not be accurate especially in the case of RDF data [23], we precompute some cardinalities between pairs of properties during data loading and use these as a corrective step. We plan to implement more elaborate techniques for cardinality estimation in the future, like for example estimations based on characteristic sets [22] or RDF data summaries [36].

5 EXPERIMENTS

In-memory data storage and query processing for our prototype 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. Our system is called through a wrapper written in Java, where also query parsing and optimization is implemented. We use the name *PARJ* for our implementation, which stands for Parallel Adaptive RDF Joins.

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) [13] and Waterloo SPARQL Diversity Test Suite (WatDiv) [6] benchmarks. All material required to reproduce the experiments is available online [3].

5.1 Setup

We use 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 [23] (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 [14] 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[4]. Note that in PARJ, each worker corresponds exactly to 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 5.2.3. 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 [14]: 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.

5.2 Results

We present results for scale 10240 of the LUBM benchmark in Table 2 (about 1.4 billion triples) and scale 1000 of the WatDiv benchmark (about 110 million triples). For WatDiv we used both

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

[4] This was verified with the TriAD implementors

basic test workload (Table 3) and incremental linear and mixed linear extensions of basic workload (Table 4). 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 [42], and are labeled LUBM1-LUBM7, and we also used three extra queries from [26] (LUBM8-LUBM10). A timeout of 30 minutes was used for all queries.

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 3), 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 4) 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 5: Impact of Adaptive Processing for LUBM 10240 and WatDiv 1000 (times in ms)-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

5.2.1 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 5. 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 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.

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 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.

5.2.2 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 6 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, 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 4: 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

Table 6: Number of binary searches and sequential searches for LUBM10240 chosen by out adaptive join method

Query	#Binary	#Sequential	Binary Search				ID-to-Position Index			
			Cycles	L1 Misses	L2 Misses	L3 Misses	Cycles	L1 Misses	L2 Misses	L3 Misses
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

5.2.3 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 5.2 and specifically from Tables 2, 3 and 4, 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

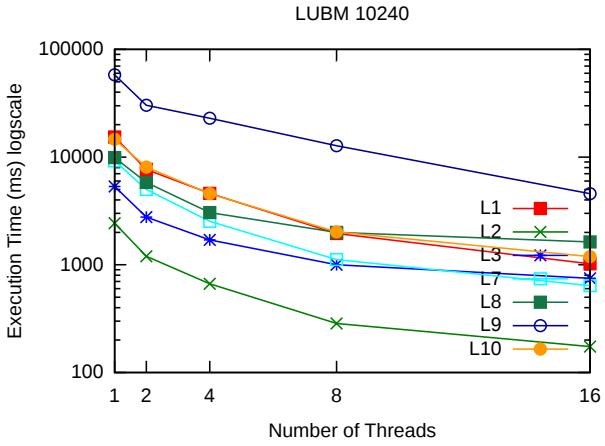

Figure 2: LUBM 10240 execution times in ms for different number of threads

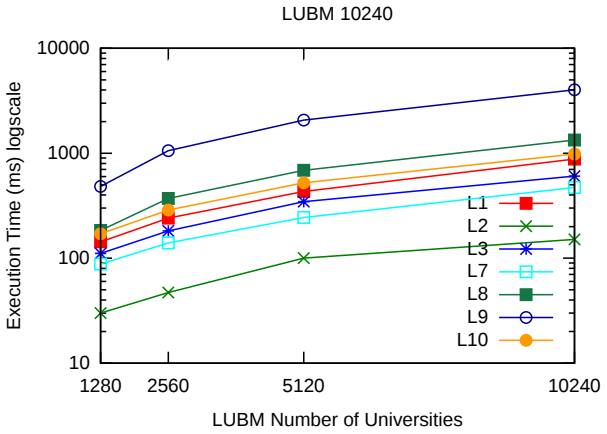

Figure 3: LUBM 32 threads execution times in ms for different dataset sizes

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.

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 2. 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 2 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 3 show a similar situation for a varying number of universities in the execution with 32 threads, confirming the excellent scalability of PARJ.

5.2.4 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).

6 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 implement 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.

Furthermore, we plan to investigate the efficient incorporation of query answering with respect to class and property hierarchies into our join approach. 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 with respect to such *entailment regimes*. Materializing all implied assertions, as it is the case in RDFS reasoning with forward chaining, with respect to these hierarchies 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. We plan to extend our join method to handle such queries, by "unioning" tables during the pipelined join execution in order to provide complete answering with respect to hierarchies, without the need to materialize the implications.

ACKNOWLEDGMENTS

We would like to thank Stratos Idreos for his comments on an early version of this work and the anonymous reviewers for their input and corrections. The present work was co-funded the European Union's Horizon 2020 research and innovation programme under grant agreement No 825258 and by the European Union (ERDF) and Greek national funds through the Operational Program "Competitiveness, Entrepreneurship and Innovation", under the call "RESEARCH-CREATE-INNOVATE" (project code:T1EDK01000).

REFERENCES

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. 2007. 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*. 411–422.

[2] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB* 10, 13 (2017), 2049–2060.

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

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1064–1075.

[5] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. 2001. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *SemWeb*.

[6] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. 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*. 197–212.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*. ACM, 1383–1394.

[8] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. 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*. 722–735.

[9] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2015. Smooth scan: Statistics-oblivious access paths. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 315–326.

[10] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2018. Smooth Scan: robust access path selection without cardinality estimation. *The VLDB Journal* (2018), 1–25.

[11] Jin-Hang Du, Haofen Wang, Yuan Ni, and Yong Yu. 2012. 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*. 633–641.

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

[13] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.* 3, 2-3 (2005), 158–182.

[14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. 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*. 289–300.

[15] Jiewen Huang, Daniel J. Abadi, and Kun Ren. 2011. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB* 4, 11 (2011), 1123–1134.

[16] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

[17] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[18] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 709–730.

[19] Stefan Manegold, Peter Boncz, and Martin L Kersten. 2002. Generic database cost models for hierarchical memory systems. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 191–202.

[20] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. 2010. SPARQL basic graph pattern processing with iterative MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. ACM.

[21] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A highly-scalable RDF store. In *International Semantic Web Conference*. Springer, 3–20.

[22] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*. IEEE Computer Society, 984–994.

[23] Thomas Neumann and Gerhard Weikum. 2009. Scalable join processing on very large RDF graphs. In *SIGMOD Conference*. ACM, 627–640.

[24] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*. ACM, 1099–1110.

[25] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. 2014. H_2RDF+: an efficient data management system for big RDF graphs. In *SIGMOD Conference*. ACM, 909–912.

[26] Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. 2016. Distributed RDF Query Answering with Dynamic Data Exchange. In *International Semantic Web Conference (1) (Lecture Notes in Computer Science)*, Vol. 9981. 480–497.

[27] Roshan Punnoose, Adina Crainiceanu, and David Rapp. 2015. SPARQL in the cloud using Rya. *Inf. Syst.* 48 (2015), 181–195.

[28] Wilson Qin and Stratos Idreos. 2016. Adaptive data skipping in main-memory systems. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2255–2256.

[29] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. 2011. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC (2) (Lecture Notes in Computer Science)*, Vol. 6644. Springer, 46–61.

[30] Kurt Rohloff and Richard E. Schantz. 2010. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *PSI EtA*. ACM, 4.

[31] Kurt Rohloff and Richard E. Schantz. 2011. Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store. In *DICT@HPDC*. ACM, 35–44.

[32] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. 2011. PigSPARQL: mapping SPARQL to Pig Latin. In *SWIM*. ACM, 4.

[33] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB* 9, 10 (2016), 804–815.

[34] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the Linked Data Best Practices in Different Topical Domains. In *Semantic Web Conference (1) (Lecture Notes in Computer Science)*, Vol. 8796. Springer, 245–260.

[35] Dominik Ślezak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. 2008. Brighthouse: an analytic data warehouse for ad-hoc queries. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1337–1345.

[36] Giorgio Stefanoni, Boris Motik, and Egor V Kostylev. 2018. 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*. International World Wide Web Conferences Steering Committee, 1043–1052.

[37] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. ACM, 553–564.

[38] Julien Subercaze, Christophe Gravier, Jules Chevalier, and Frederique Laforest. 2016. Inferray: fast in-memory RDF inference. *Proceedings of the VLDB Endowment* 9, 6 (2016), 468–479.

[39] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *PVLDB* 1, 1 (2008), 1008–1019.

[40] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. 2003. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*. 131–150.

[41] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. *Proceedings of the VLDB Endowment* 6, 7 (2013), 517–528.

[42] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6, 4 (2013), 265–276.