Scalable Parallelization of RDF Joins on Multicore Architectures

Dimitris Bilidas
National and Kapodistrian University of Athens
Greece
d.bilidas@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 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 BGPds 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 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 BGP 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
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.

H2+RDF+ [25] uses HBase1 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 Accumulo2, 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 indexing 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 side-ways 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 propi; 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

1http://hbase.apache.org/
2http://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 \texttt{prop} and \texttt{props} that correspond to properties \texttt{teaches} and \texttt{worksFor} will be created.

### Table 1: Example of Dictionary Encoding

<table>
<thead>
<tr>
<th>Integer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProfessorA</td>
</tr>
<tr>
<td>2</td>
<td>teaches</td>
</tr>
<tr>
<td>3</td>
<td>Mathematics</td>
</tr>
<tr>
<td>4</td>
<td>ProfessorB</td>
</tr>
<tr>
<td>5</td>
<td>Chemistry</td>
</tr>
<tr>
<td>6</td>
<td>ProfessorC</td>
</tr>
<tr>
<td>7</td>
<td>Literature</td>
</tr>
<tr>
<td>8</td>
<td>Physics</td>
</tr>
<tr>
<td>9</td>
<td>worksFor</td>
</tr>
<tr>
<td>10</td>
<td>University1</td>
</tr>
<tr>
<td>11</td>
<td>University2</td>
</tr>
</tbody>
</table>

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 \(2 \ast (\text{number of properties})\) 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 \texttt{prop}, we should look at position \(2 \ast 7\), whereas to get the number of objects for \texttt{prop2}—we should look at position \((2 \ast 7) + 1\).

Figure 1 contains an example of physical storage for a property table. Given that the specific table is for property \texttt{prop1}, then it contains the following triples: \(5 \texttt{prop} 8, 7 \texttt{prop} 8, 7 \texttt{prop} 34, 13 \texttt{prop} 40, 18 \texttt{prop} 3, 24 \texttt{prop} 9, 24 \texttt{prop} 16, 24 \texttt{prop} 41, 29 \texttt{prop} 40, 33 \texttt{prop} 22, 45 \texttt{prop} 4\). 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:

```sparql
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 \texttt{prop2 \ast \texttt{subject=subject props}}. If there are two available threads, our algorithm will start concurrently scanning two different shards of \texttt{prop2}. For each tuple encountered during this process, it will probe, using binary search, table \texttt{props}. 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:

```sparql
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 \texttt{prop} 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 \texttt{props} for \texttt{object} = 10, we obtain the vector of subjects that correspond to \texttt{object} = 10 (in our case it is only value 1). Then we start scanning this vector and probing table \texttt{props2} 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 \texttt{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:

```sql
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 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 are 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 \( (\text{arraysize} - 1) \mod \text{arraysize}/\text{arraysize} \). 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 NumberOfSearches 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 computa-
tion and one comparison for each tuple (lines 2-3 of Algorithm 1).

Algorithm 1: Adapтивely switching between binary and se-
quential 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 tuples
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
data set 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 sup-
posing 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 posi-
tion 7 the value 1, at position 13 the value 2 and so on for pos-
titions 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 de-
note the position of the property table at specific intervals, and

<table>
<thead>
<tr>
<th>Algorithm 2: Calibration Process</th>
</tr>
</thead>
</table>
| 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 AveGap = (Array[Size - 1] – Array[0])/Size; |
| 4 do |
| 5 WindowSize = NextWindowSize; |
| 6 TotalGap = AveGap * WindowSize; |
| 7 SearchDistance = 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 TimeBinary = getTimeNow() – StartTime; |
| 14 TimeScan = gettimeNow() – TimeBinary; |
| 15 EndSearch = Array[0]; |
| 16 PreviousSearchPosition = 0; |
| 17 StartTime = getTimeNow(); |
| 18 for k ← 0 to NoOfSearches do |
| 19 Sequential_search (array, toFind, 
  &PreviousSearchPosition); |
| 20 Time = TotalGap; |
| 21 TimeBinary = getTimeNow() – StartTime; |
| 22 TimeScan = gettimeNow() – TimeBinary; |
| 23 if TimeBinary > TimeScan then |
| 24 Fraction = TimeBinary/TimeScan; |
| 25 NextWindowSize = WindowSize * Fraction; |
| 26 else |
| 27 Fraction = TimeScan/TimeBinary; |
| 28 NextWindowSize = WindowSize/Fraction; |
| 29 while Fraction > Threshold; |
| 30 return WindowSize; |

for all other positions we use a bit value to simply denote pres-
ence 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 0,0,0,0,1,1,0,1,
then integer value 1 and bit values 0,0,0,0,1,0,0,0, then inte-
ger value 2 and bit values 0,1,0,0,0,0,1, then integer value 4
and bit values 0,0,0,0,1,0,0, 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 \((29 \div 8) + 1\) \(\times\) \(M\) \(\times\) 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 \((29 \div 8) \times M + 8 + (29 \div 8) \times 8\) 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 \(N/8 + ((N/A) \times M)\) 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 also 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 that 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 LUBM5, 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 ms (geomean: 6.8).

Table 5: Impact of Adaptive Processing for LUBM 10240 and WatDiv 1000 (times in ns)-1 thread

<table>
<thead>
<tr>
<th>Query</th>
<th>Binary</th>
<th>AdBinary</th>
<th>Index</th>
<th>AdIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM1</td>
<td>22186</td>
<td>15454</td>
<td>16557</td>
<td>15369</td>
</tr>
<tr>
<td>LUBM2</td>
<td>2877</td>
<td>2443</td>
<td>2535</td>
<td>2437</td>
</tr>
<tr>
<td>LUBM3</td>
<td>6562</td>
<td>5491</td>
<td>6415</td>
<td>5338</td>
</tr>
<tr>
<td>LUBM4</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>LUBM5</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LUBM6</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>LUBM7</td>
<td>12246</td>
<td>11866</td>
<td>9197</td>
<td>9213</td>
</tr>
<tr>
<td>LUBM8</td>
<td>15725</td>
<td>9782</td>
<td>10420</td>
<td>9899</td>
</tr>
<tr>
<td>LUBM9</td>
<td>77468</td>
<td>63586</td>
<td>58171</td>
<td>58082</td>
</tr>
<tr>
<td>LUBM10</td>
<td>22359</td>
<td>14892</td>
<td>16217</td>
<td>14606</td>
</tr>
<tr>
<td>Avg</td>
<td>15943</td>
<td>12352</td>
<td>11952</td>
<td>11495</td>
</tr>
<tr>
<td>Geomean</td>
<td>1034</td>
<td>892</td>
<td>898</td>
<td>864</td>
</tr>
</tbody>
</table>

WatDiv1000 Avg 8439 8003 5013 4869
WatDiv 1000 Geomean 33 28 25 23

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.

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 columns), whereas when the ID-to-Position index is used (comparison between third and fourth columns) 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)

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

### Table 3: Results for WatDiv Basic Workload scale 1000 (times in ms)

<table>
<thead>
<tr>
<th></th>
<th>Single Thread</th>
<th>Multi-Thread</th>
<th>PARJ</th>
<th>RDFox</th>
<th>RDF-3X</th>
<th>PARJ-32</th>
<th>TriAD</th>
<th>TriAD-SG 200K</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>5</td>
<td>5</td>
<td>40</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>8</td>
<td>43</td>
<td>30</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3</td>
<td>2</td>
<td>244</td>
<td>13</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L4</td>
<td>3</td>
<td>7</td>
<td>19</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L5</td>
<td>9</td>
<td>57</td>
<td>40</td>
<td>6</td>
<td>3</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td>5</td>
<td>71</td>
<td>28</td>
<td>6</td>
<td>3</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geomean</td>
<td>5</td>
<td>29</td>
<td>26</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 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)

<table>
<thead>
<tr>
<th>Thread</th>
<th>PARJ</th>
<th>RDFox</th>
<th>RDF-3X</th>
<th>PARJ-32</th>
<th>TriAD</th>
<th>TriAD-SG 200K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Thread</td>
<td>Avg</td>
<td>ML-1 5</td>
<td>3</td>
<td>27617</td>
<td>1339</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Geomean</td>
<td>ML-1 4</td>
<td>4</td>
<td>20498</td>
<td>1832</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>IL-1 7</td>
<td>8</td>
<td>66909</td>
<td>1272</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Geomean</td>
<td>IL-1 3</td>
<td>3</td>
<td>700199</td>
<td>1633</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>IL-1 9</td>
<td>26</td>
<td>728518</td>
<td>1396</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Geomean</td>
<td>IL-1 10</td>
<td>29</td>
<td>734363</td>
<td>1923</td>
<td>9</td>
</tr>
<tr>
<td>Avg</td>
<td>12</td>
<td>510782</td>
<td>1566</td>
<td>7</td>
<td>1132</td>
<td>17113</td>
</tr>
<tr>
<td>Geomean</td>
<td>8</td>
<td>35194</td>
<td>1546</td>
<td>6</td>
<td>1002</td>
<td>15068</td>
</tr>
</tbody>
</table>

Table 5: Results for varying dataset size

<table>
<thead>
<tr>
<th>Workload</th>
<th>IL-1 5</th>
<th>IL-1 6</th>
<th>IL-1 7</th>
<th>IL-1 8</th>
<th>IL-1 9</th>
<th>IL-1 10</th>
<th>Avg</th>
<th>Geomean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg</td>
<td>12</td>
<td>510782</td>
<td>1566</td>
<td>7</td>
<td>1132</td>
<td>17113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geomean</td>
<td>8</td>
<td>35194</td>
<td>1546</td>
<td>6</td>
<td>1002</td>
<td>15068</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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...
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]. 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 by 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:TIEDK01000).

REFERENCES


