RDF Chain Query Optimization in a Distributed Environment

Alexander Hogenboom hogenboom@ese.eur.nl
Ewout Niewenhuijse ewoutnwn@gmail.com
Milan Jansen milanjansen@gmail.com
Flavius Frasincar frasincar@ese.eur.nl
Damir Vandic vandic@ese.eur.nl
Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, the Netherlands

ABSTRACT

In order to efficiently disclose the ever-growing amount of distributed RDF data in Semantic Web environments, RDF query engines must optimize the join order of partial query results. Existing methods include two-phase optimization (2PO), a genetic algorithm (GA), and ant colony optimization (ACO), which have mostly been evaluated on a single source. We adapt these methods to a distributed setting and evaluate the effects of distinct join methods, i.e., nested-loop join, bind join, and AGJoin. When optimizing RDF chain queries combining real-world data from 34 different SPARQL endpoints, the ACO method produces the best results in the least amount of time for most chain queries consisting of up to about ten joins. For larger chain queries, each of our considered algorithms may have its benefits, depending on the join method used. When using the least naive join method, i.e., AGJoin, a GA approach produces solutions of a competitive quality in significantly less time than both ACO and 2PO.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms, experimentation, performance

Keywords

RDF chain query optimization, ant colony optimization, genetic algorithm, iterative improvement, simulated annealing

1. INTRODUCTION

Today's decision makers need to process an overwhelming, continuous flow of data in order to extract information, understand its meaning, and acquire knowledge [10]. The Semantic Web [3] enables effective, well-informed decision making, as it allows for an ever-growing amount of linked data to be stored in heterogeneous, interconnected sources. Data is represented by the Resource Description Framework (RDF) for describing and interchanging metadata [13], which facilitates machine-interpretability of data.

The Semantic Web has the potential of addressing complex information needs more effectively and efficiently than the current Web. Semantic Web technologies allow a user to combine data from many sources in order to address very specific information needs. To this end, RDF data sources are typically queried by means of the SPARQL Protocol and RDF Query Language (SPARQL) [16]. Fast RDF query engines are crucial in order for SPARQL queries to efficiently disclose the ever-growing amount of widely distributed RDF data to demanding users in real-time environments.

When querying multiple sources, a query typically consists of multiple subqueries, the results of which are combined in order to match the information need specified by the query as a whole. In order to enable efficient querying, it is crucial to optimize the order in which the distinct parts of a query are executed, as the total execution times of queries depend on these query paths. However, the number of possible query paths grows exponentially with the query size.

Several methods have been proposed in order address the query path optimization problem in an RDF environment. One of the first proposed techniques was a two-phase optimization (2PO) algorithm [20], consisting of iterative improvement (II) followed by simulated annealing (SA). More recently, a genetic algorithm (GA) [11] and an ant colony optimization (ACO) approach [10] have been shown to be promising alternatives when optimizing chained RDF queries.

In our previous work [10, 11], we have assessed the performance of such techniques on a single source only. However, as a typical use case for RDF queries involves querying multiple heterogeneous sources, the purpose of our current endeavors is to adapt existing work in order for it to be useful in a distributed environment, and to subsequently assess the performance of our methods. As such, our current work is a generalization of existing work [10] for a distributed environment in which the join order of the results of a series of SPARQL queries, i.e., chain queries, is to be optimized. Additionally, we evaluate the effects of various join methods.

The remainder of this paper is structured as follows. First, Section 2 provides a short introduction to RDF chain query optimization in a distributed setting. We then describe our considered RDF chain query optimization methods and how they can be applied in a distributed setting in Section 3 and Section 4, respectively. In Section 5, we evaluate the performance of our considered methods. Last, we draw conclusions and propose directions for future work in Section 6.

2. QUERYING DISTRIBUTED RDF DATA

In RDF, data is modeled as a sequence of facts, organized as a collection of triples. An RDF triple consists of a subject, a predicate and an object, and can be visualized as a node-arc-node link in a directed graph [13]. The relationship of a subject node to an object node is defined by an arc denoting a predicate. When querying RDF data, triples are matched against patterns, described in SPARQL queries. These queries may consist of subqueries, the results of which may come from different sources and are joined in order to process the full query. The join order of these results must be optimized in order to enable efficient querying.

2.1 Chain Queries on SPARQL Endpoints

In the subset of SPARQL query patterns we consider in our current work, the WHERE statement essentially joins sets of node-arc-node patterns resulting from SPARQL subqueries. These triple sets are chained together such that one node set contains a mapping to the next node set.

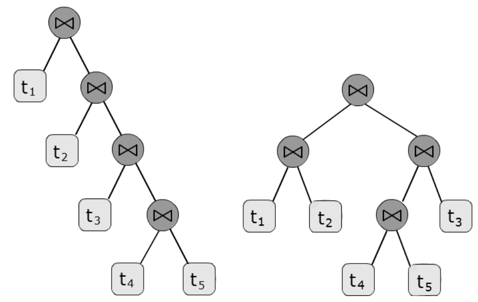
When querying RDF data in on-line knowledge bases, SPARQL endpoints offer a machine-friendly interface to access the underlying RDF data. As more and more data gets published in the distributed Semantic Web, combining on-line SPARQL endpoints allows for answering queries embodying multiple distributed RDF data sources.

A typical example of a chain query on multiple distributed sources is the following[1]. Consider five on-line RDF data sources, accessible through SPARQL endpoints, with triple amounts ranging up to a billion per source. Suppose one wants to know which musicians from countries of a certain population range have contributed to comedy movie soundtracks. This query can be split into the following five local queries: (1) a query on WordNet for hyponyms of the literal "comedy" to capture genres marked by different but narrower terms (e.g., romantic comedy, black comedy, tragicomedy, etc.) too, (2) a query at LinkedMDB for the names of the music contributors for all soundtracks of all movies in all genres, (3) a query at DBpedia for the birth places of all persons, (4) a query on the CIA World Factbook for the population sizes of countries, and (5) a query on FactForge for the countries associated with specific places. In order to resolve the complete query, local queries can be sent to the mentioned data sources, and the local query results can be joined one-by-one in any order.

2.2 Solution Space

A sequence of joins in a chain query can be visualized as a tree, where the leaf nodes represent local query results and the internal nodes model algebra operations, enabling one to specify basic retrieval requests on the inputs [6]. The leaf nodes of a chain query tree are RDF triple responses of local queries and the internal nodes represent join operations. The nodes in a chain query tree can be ordered in many different ways, referred to as query paths. The result of executing a query is independent of the ordering of the nodes in a query tree. Yet, the total execution time of a query depends on the order of the joins of its subqueries. Therefore, query path optimization is crucial in today's distributed RDF environments. In this optimization process, a solution space is typically explored, where each solution is associated with solution costs – quantified as its execution time – and where similar solutions are located close to one another.

[1]Available on-line at `http://people.few.eur.nl/hogenboom/files/chainquery.rq`.



(a) Right-deep tree. (b) Bushy tree.

Figure 1: Examples of join orders for an RDF chain query with five joins. Square nodes represent triples matching subqueries, whereas circular nodes represent joins.

The solution space can contain two types of query trees, i.e., bushy trees and their subset of right-deep trees. In bushy trees, base relations as well as results from earlier joins can be joined, whereas right-deep trees require the left hand operands to be base relations. For example, Figure 1 shows a right-deep and bushy query tree for the query in Section 2.1, where $\{t_1, t_2, t_3, t_4, t_5\}$ corresponds with its local query results, and $\bowtie$ represents a join. For n base relations, or leaf join sets, and thus $n-1$ joins, there are $n!$ possible solutions for right-deep trees, whereas there are $\binom{2(n-1)}{(n-1)}(n-1)!$ possible solutions for bushy query trees [19]. In practice, in 70% of the cases, a bushy tree solution exceeds right-deep solutions in terms of quality, i.e., has an execution order associated with lower costs [19].

2.3 Two-Way Join Techniques

In a query path, an individual join is executed over two data sources, thus resulting in a two-way join. The most naive two-way join method is the nested-loop join, where all (intermediate) RDF triple responses in the join operands are compared with one another in order to check whether they meet the join condition. An attractive alternative to this method is the bind join method [8], which is essentially a nested-loop join in which intermediate results from the smallest operand are used as a filter for the largest join operand [7, 17].

In a distributed environment, the data sources to be joined are not readily available from memory or hard disk, but should be accessed via a wide-area network (WAN), e.g., the (Semantic) Web. Therefore, rather than waiting for all data to be loaded before joining, a distributed join technique should start as soon as both data sources are streaming.

Recent work [1] describes such a technique, called adaptive group join (AGJoin). It is based on the symmetric hash join and XJoin operators and designed to rapidly produce answers from streamed data accessible trough a WAN [1].

3. RDF CHAIN QUERY OPTIMIZATION

As not every query path is as efficient as others, the challenge in query path optimization is to find a query path that minimizes query execution costs. In order to explore the potentially large solution spaces associated with the RDF chain query optimization problem, several soft computing techniques have been proposed in existing work.

3.1 Two-Phase Optimization

A 2PO approach [19] explores the solution space by means of an II phase, followed by a SA phase. In the second phase, the optimum obtained in the first phase is improved in an attempt to reduce the risk of obtaining a local optimum.

In the II phase, random starting points are generated, from which the exploration of the solution space is started. For each encountered solution, a number of random neighbors is explored. In this process, as soon as a neighbor with lower associated execution costs is found, a step to this neighboring solution is performed. Here, a neighboring solution is defined as a solution that can be reached by performing one of the following moves: join commutativity, join associativity, left join exchange, and right join exchange [12]. This procedure is repeated until no solution with lower cost can be found or a time limit is reached.

A drawback of II is that it is likely to find local optima. To counter this drawback, SA complements II by allowing, with a declining probability, to move to a neighbor with higher cost, as this move may, in the future, reveal a neighbor with a lower-cost local optimum. Usually, SA is only applied on the best local optimum found in the II phase.

3.2 Genetic Algorithm

An alternative to the 2PO approach is to use a GA [11, 19]. In a GA, a population of solutions is subject to a process of simulated evolution, adhering to the principle of survival of the fittest. Here, the fitness of a solution is inversely proportional to its associated execution costs. The starting population is typically a collection of randomly selected solutions from the solution space. Then, for each subsequent generation, a fraction of the fittest solutions is selected for proliferation. Additionally, randomly selected solutions from the current generation are combined in order to generate offspring for the subsequent generation, where their selection probability depends on their fitness. Last, mutation is applied to a fraction of the new generation. The evolution process continues until the maximum number of generations has been simulated or a number of generations have not yielded any improvement. The fittest member of the last generation's population is selected as solution.

3.3 Ant Colony Optimization

ACO [4] is an optimization method inspired by the foraging behavior of ant colonies, where ants explore their environment and mark their paths to a food source with traces of pheromones. The paths taken by foraging ants partly depend on these pheromone traces. Over time, shorter paths are traversed with increasing frequency and as such attract an increasing amount of pheromone, whereas the pheromone traces on less efficient paths evaporate over time. The colony thus converges to using a short path from the nest to the food source.

The ACO algorithm translates the foraging behavior of ant colonies to a graph used by artificial ants, dropping pheromones on edges. The graph contains paths from a starting node s, representing the nest, to an ending node t, representing the food source.

If ant k arrives at node u, the probability $p^k_{uv}(i)$ of an edge between nodes u and v to be chosen at iteration i is defined as

$$p^k_{uv}(i) = \begin{cases} \frac{\phi^\alpha_{uv}(i-1)\eta^\beta_{uv}}{\sum_{z\in N_u}\phi^\alpha_{uz}(i-1)\eta^\beta_{uz}}, & v \in N^k_u, \\ 0, & v \notin N^k_u, \end{cases} \tag{1}$$

where $\phi_{uv}(i-1)$ represents the pheromone level at the edge between nodes u and v, as deposited at iteration $i-1$, and η_{uv} is a local heuristic measure capturing the inverse of the length of the edge connecting nodes u and v, and N^k_u represents all unvisited nodes by ant k after visiting node u. Last, α controls the weight of the pheromone level, whereas β is a weight for the local heuristic measure.

After an artificial ant has selected and traversed an edge e_{uv}, it drops a quantity of pheromones on this edge, which is defined for iteration i as

$$\Delta\phi^k_{uv}(i) = \begin{cases} \frac{Q}{L_k(i)}, & e_{uv} \in T_k(i), \\ 0, & e_{uv} \notin T_k(i), \end{cases} \tag{2}$$

where Q is a constant and $L_k(i)$ is the total length of the path $T_k(i)$ from s to t, taken by ant k at iteration i. The pheromone drop is computed based on the total length of the tour, instead of the length of individual edges, as an individual lengthier edge can be part of an overall shorter tour [4].

The quantity of pheromones at an individual edge decreases in a continuous process, referred to as evaporation. The level of pheromone $\phi_{uv}(i)$ on an edge at the end of iteration i is defined as

$$\phi_{uv}(i) = (1-\rho)\,\phi_{uv}(i-1) + \sum_{k=1}^{m} \Delta\phi^k_{uv}(i), \tag{3}$$

where ρ represents the evaporation factor and $\Delta\phi^k_{uv}(i)$ represents the pheromone deposit for each ant k out of m ants that have traversed edge e_{uv}.

In our previous work [10], we have applied ACO to RDF chain query optimization, by using a graph-based encoding scheme, which is based on the ordinal encoding scheme proposed in [19]. This ordinal encoding scheme iteratively joins two operands, i.e., base relations or results of earlier joins. These operands occur in an ordered list of operands. The result of a join of two operands which is saved in the position of the first appearing operand. The sequence of pairs of indices of operands thus obtained, i.e., one pair for each join, is used to encode the solution.

For example, the query path in Figure 1(b) has an initial ordered list $\{t_1, t_2, t_3, t_4, t_5\}$, and then first joins t_4 with t_5, encoded as $(4, 5)$. This initial join yields the ordered list $\{t_1, t_2, t_3, t_4t_5\}$. A subsequent join of t_1 with t_2, encoded as $(1, 2)$, results in the ordered list $\{t_1t_2, t_3, t_4t_5\}$. A third join, i.e., a join of t_4t_5 with t_3, encoded as $(3, 2)$, yields the ordered list $\{t_1t_2, t_4t_5t_3\}$. A final join of t_1t_2 with $t_5t_4t_3$, encoded as $(1, 2)$, results in the ordered list $\{t_1t_2t_5t_4t_3\}$ and the encoded query path $[(4, 5), (1, 2), (3, 2), (1, 2)]$.

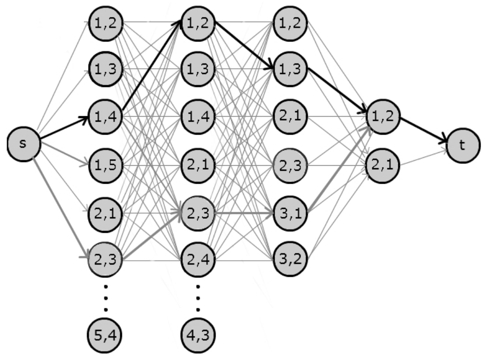


Figure 2: Our considered graph representation for the example query discussed in Section 2.1. A directed path from s to t represents one query path, where each node other than s or t represents a join of two operands given by their index and an edge represents the choice towards a subsequent join. In case of ACO, pheromones are dropped on edges, and a thicker edge in this representation signals a higher pheromone concentration. The bold black edges highlight a typical query path $s-(1,4)-(1,2)-(1,3)-(1,2)-t$. The full graph contains all possible query paths in the bushy solution space.

The graph-based encoding scheme models an encoded query path as a directed path between s and t, with a number of intermediate nodes, each representing a choice for a subsequent join of two operands. Starting from node s, all possible choices for a subsequent join are given by the outgoing edges to any node v, where a choice is described by the comma-separated index of both operands. Figure 2 depicts how all possible choices for subsequent joins can be represented in one graph. In such a graph, the ant's observed distance of moving from one node to another is based on its estimated costs of making that move, given the partial path taken so far. Each time an ant moves from a node u to a subsequent node v, it drops an amount of pheromone on the connecting edge e_{uv}, as computed in (2). A thicker edge in Figure 2 represents a bigger amount of pheromone, indicating a part of a better query path.

4. DISTRIBUTED OPTIMIZATION OF CHAIN QUERIES

Existing work [10, 11] assesses the performance of techniques like those discussed in Section 2 on a single source only. In our current endeavors, we adapt existing methods in order for them to be useful in a distributed environment. In this light, we propose a distributed solution cost model, focused on the transmission cost of local query results.

4.1 Distributed Solution Cost Model

A query path consists of two-way joins, each joining two local query results. To execute an individual join, the local query results need to be transmitted and subsequently merged. A join j in query path p is associated with join costs c_{p_j}. The total costs c_p associated with all J joins in a full query path p can be modeled as

$$c_p = \sum_{j=1}^{J} c_{p_j}. \tag{4}$$

The costs c_{p_j} associated with individual joins j in a query path p depend on the join method. A substantial part of the join costs is formed by transmission costs. We model the transmission costs $\tau_{o_{pj\Omega}d_\Omega h}$ for join operand Ω as

$$\tau_{o_{pj\Omega}d_\Omega h} = \gamma_{d_\Omega h} + \left|o_{pj\Omega}\right| \chi_{d_\Omega h} \lambda_{o_{pj\Omega}}. \tag{5}$$

Here, $\gamma_{d_\Omega h}$ represents the initialization costs for retrieving anything from the data source d_Ω for join operand Ω to the query processing host h, $\chi_{d_\Omega h}$ quantifies the transmission costs per character from source d_Ω to processor h, $\lambda_{o_{pj\Omega}}$ denotes the average triple length of join operand Ω, and $\left|o_{pj\Omega}\right|$ represents the cardinality of this join operand.

For nested-loop joins [5], we define the nested-loop join costs $c^{\mathrm{N}}_{p_j}$ as a function of the transmission costs of all RDF triples in its join operands, and the host's processing costs associated with comparing all combinations of these triples, i.e.,

$$c^{\mathrm{N}}_{p_j} = \tau_{o_{pj1}d_1 h} + \tau_{o_{pj2}d_2 h} + \left|o_{pj1}\right| \left|o_{pj2}\right| \psi_h, \tag{6}$$

with $\tau_{o_{pj1}d_1 h}$ and $\tau_{o_{pj2}d_2 h}$ representing the transmission costs for the first join operand o_{pj1} and second join operand o_{pj2}, respectively. Additionally, $\left|o_{pj2}\right|$ and $\left|o_{pj2}\right|$ in (6) represent the respective cardinalities of the first and second join operand. Last, ψ_h quantifies the processing costs per comparison of two triples.

When using a bind join method [8], a nested-loop join is performed, where the intermediate results of the first operand are used as a filter for the second join operand, thus realizing a join selectivity σ_{p_j} on the second operand. As such, we define the bind join costs $c^{\mathrm{B}}_{p_j}$ as

$$\begin{aligned} c^{\mathrm{B}}_{p_j} =& \tau_{o_{pj1}d_1 h} + \left|o_{pj1}\right| \cdot \left(\gamma_{d_2 h} + \sigma_{p_j} \left|o_{pj2}\right| \chi_{d_2 h} \lambda_{o_{pj2}}\right), \\ & \left|o_{pj1}\right| \leq \left|o_{pj2}\right|. \end{aligned} \tag{7}$$

Using an AGJoin operator [1], the merge process can be started as soon as the first data streams in, provided that two data streams are processed simultaneously. As local merging is much faster than transmission over a WAN, the merge process can finish the moment after the transmission of the last triple is finished, giving no significant additional merge costs. Consequently, the AGJoin costs $c^{\mathrm{A}}_{p_j}$ of a join j in query path p are modeled as the maximum of the transmission costs $\tau_{o_{pj1}d_1 h}$ and $\tau_{o_{pj2}d_2 h}$ of the join operands from their respective data sources d_1 and d_2 to the query processing host h, i.e.,

$$c^{\mathrm{A}}_{p_j} = \max\left(\tau_{o_{pj1}d_1 h}, \tau_{o_{pj2}d_2 h}\right). \tag{8}$$

4.2 Cardinality Estimation

A chain query path p consists of multiple joins, where each join j results in a join set p_j. For base join sets, the cardinality can be accurately measured by counting the number of triples that match the pattern, or by using VoID [2] provided by the SPARQL endpoints. The cardinality of a non-base join set p_j, however, is the result of a join and depends on both join sets. Multiple approaches exist to estimate the cardinality of a non-base join set in, e.g., distributed databases, by using histograms [14] or Bloom filters [9, 15].

In the worst-case scenario, no triple occurs in both sets and the new cardinality $\left|o_{p_j}\right|$ is a Cartesian product of both operands $\left|o_{p_{j1}}\right|$ and $\left|o_{p_{j2}}\right|$. Conversely, in the best-case scenario, all triples occur in both sets, resulting in a new cardinality equal to $\max\left(\left|o_{p_{j1}}\right|, \left|o_{p_{j2}}\right|\right)$. In a typical case, the cardinality will be in between both extremes. Therefore, the join cardinality is defined as

$$\left|o_{p_j}\right| = \left|o_{p_{j1}}\right| \left|o_{p_{j2}}\right| \sigma_{p_j}, \tag{9}$$

where σ_{p_j} represents the selectivity of join p_j for query path p. In our current endeavors, no mapping is applied yet and therefore, the join selectivity is expected to be unknown a priori. Consequently, we apply a rule of thumb from traditional databases in order to estimate the join selectivity, by giving σ_{p_j} a value of 10% [18]. This estimate can be updated on the fly in real-time, real-life systems.

5. EVALUATION

Our proposed distributed solution cost model enables the application of several methods for RDF chain query optimization in a distributed setting. These methods have been tested on a single source previously [10], and in our current endeavors, we assess the performance in terms of quality and optimization time on multiple distributed sources, while applying various cost functions.

5.1 Experimental Setup

In order to assess the performance of the considered methods for RDF chain query optimization in a distributed setting, we have run experiments on a 64-bit 3.4 Ghz Intel i7 − 2600K machine with 16 GB physical memory. We evaluate the performance of RDF chain query optimization by means of 2PO (RCQ-2PO), a GA (RCQ-GA), and ACO (RCQ-ACO) on multiple distributed sources, i.e., 34 SPARQL endpoints, and one host for executing all joins. Using these sources, we evaluate our considered algorithms on random RDF chain queries with lengths of 3 to 20 local query results, thus covering problem sizes of 2 to 19 joins. For each query length, we optimize 1,000 different chain queries and evaluate each algorithm's execution times until convergence and costs of found solutions. In our experiments, we consider three complete solution spaces with bushy query trees, i.e., solution spaces associated with nested-loop joins, bind joins, and AGJoins.

In order to enable computation of the transmission speed for a local query result, we have first sampled the responsiveness of each considered SPARQL endpoint host at 100 different points in time, such that in our query optimization process, an estimate of the transmission speed of an endpoint can be modeled as a random value drawn from our empirically generated distribution of response times for that specific endpoint. These empirical distributions have been generated by sending ASK and SELECT queries to all considered endpoints, respectively querying for any result, e.g., ASK (?*x* ?*y* ?*z*), and querying for 500 triples that match the pattern (?*x* ?*y* ?*z*), e.g., SELECT ?*x* ?*y* ?*z* WHERE (?*x* ?*y* ?*z*) LIMIT 500. For each query, we measured the time period between sending and receiving all results, and for each SELECT query, we additionally measured the size of the result in characters. The responsiveness is measured in milliseconds and represents the time period of the ASK query. The performance is measured as the transmission speed in bytes per millisecond, which is computed as character length of the 500 triples response for the SELECT query, divided by the time period for the SELECT query minus the time period for the ASK query[2].

The considered algorithms need to be configured for our current purposes. For RCQ-2PO, we adopt the settings proposed in [19]. We thus start the II phase with 10 random starting points for random walks in the solution space. The best local optimum obtained from II is taken as starting point for the extended random walk in the SA phase. The system's temperature is initialized at 10% of the starting solution's associated costs. For each next solution on the path traversed through the solution space, the algorithm tries to move to neighboring solutions for a limited number of times, which is set as 16 times the number of joins in the query. After 16 tries, the system's temperature is reduced with 5%. The system is considered to be frozen when the temperature drops below 1 or when the best solution so far has not been improved in four consecutive temperature reductions.

The RCQ-GA algorithm is configured in accordance with the settings suggested in [11]. As such, a set of 64 chromosomes is exposed to a process of simulated evolution with a crossover rate of 0.65 and a mutation rate of 0.05. In this process, fitness-based selection is applied. In addition to this, elitist selection is applied, such that in each generation, the best chromosome is always selected for proliferation in the next generation. The RCQ-GA algorithm is considered to have been converged after 30 consecutive generations without any improvement in terms of fitness of the best encountered solution, with the fitness being computed as the inverse of the associated solution costs.

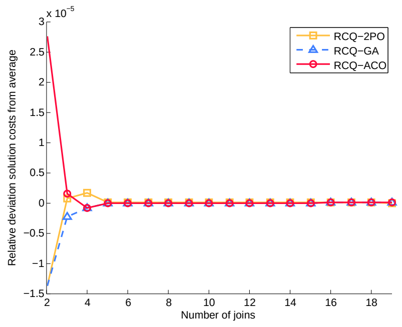
In order to stimulate the RCQ-ACO algorithm to explore different parts of our considered solution spaces, we propose to use six times as many ants as the number of local query results, with these ants relying as much on global pheromone trails as they do on local heuristics, i.e., by setting both α and β to 1. Additionally, we promote relatively quick convergence of RCQ-ACO by using an evaporation rate ρ of 0.25, and by considering the colony to have been converged after five consecutive iterations without improvement in solution quality. In addition to this, we limit the maximum number of iterations to 15 in order for the algorithm not to spend too much time optimizing already good solutions. Last, the constant Q is set to 10.

5.2 Experimental Results

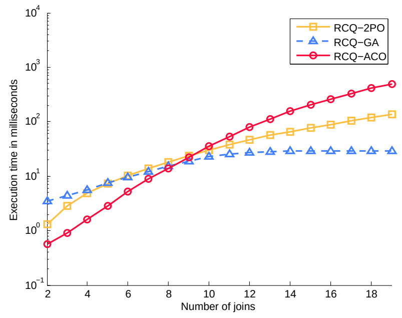
Our RDF chain query optimization experiments reveal differences among the considered algorithms and solution spaces in terms of both execution time for the optimization process and quality of the optimized solution, quantified by its associated execution costs. Figures 3, 4, and 5 demonstrate that the patterns exhibited by our considered RDF chain query optimization algorithms are consistent across solution spaces. However, the quality of the optimized solutions appears to be sensitive to the join method used.

Irrespective of the cost function used, RCQ-ACO is the fastest performing algorithm for smaller queries, consisting of up to about ten joins, whereas RCQ-GA is the fastest algorithm for larger queries. The optimization time needed by RCQ-2PO typically falls in between the extremes of RCQ-ACO and RCQ-GA.

[2]For the results, see `http://people.few.eur.nl/hogenboom/files/hosts.dat`.

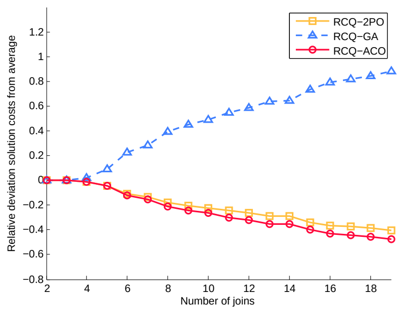


(a) Solution costs for optimized solutions.

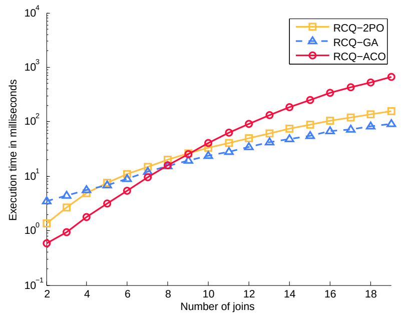


(b) Execution times of optimization.

Figure 3: Performance of our considered methods over chain queries with 2 **to** 19 **joins, averaged over** 1,000 **queries for each query size, when using nested-loop joins.**



(a) Solution costs for optimized solutions.



(b) Execution times of optimization.

Figure 4: Performance of our considered methods over chain queries with 2 **to** 19 **joins, averaged over** 1,000 **queries for each query size, when using bind joins.**

When using the naive nested-loop join method, the solutions produced by our considered algorithms hardly differ from one another in terms of solution costs, even though the RCQ-ACO method mostly tends to produce solutions of a marginally better quality. As such, when using the nested-loop join method, RCQ-ACO is the fastest algorithm that moreover produces the best results for most smaller queries. For larger queries, RCQ-GA is the fastest algorithm that produces solutions of a competitive quality.

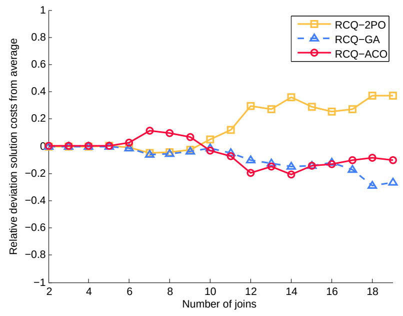
The quality of solutions produced by RCQ-GA is less competitive when using bind joins. Here, RCQ-ACO consistently produces the best results, whereas RCQ-2PO produces solutions of a quality inferior to RCQ-ACO, but superior to RCQ-GA. Therefore, for smaller queries, RCQ-ACO produces the best query paths with bind joins in the least amount of time. For larger queries, there is a trade-off between solution quality (RCQ-ACO) and execution time (RCQ-2PO).

When considering a space with solutions that use AGJoins, all algorithms tend to produce solutions of a comparable quality for queries consisting of up to ten joins. For larger queries, the solutions produced by RCQ-2PO are of an inferior quality, compared to RCQ-ACO and RCQ-GA. As such, for smaller queries, it is best to optimize the join order of RDF chain queries that use AGJoins by means of RCQ-ACO, as this algorithm is the fastest algorithm that produces solutions of a quality that is comparable with the quality of those produced by the alternative optimization algorithms. For larger queries that use AGJoins, RCQ-GA is the fastest algorithm with a competitive solution quality.

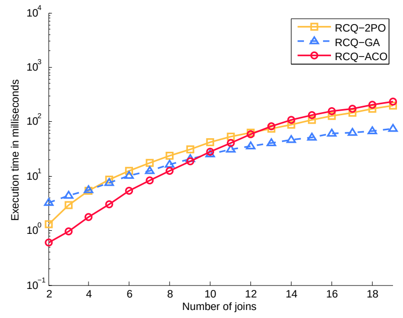
6. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated that our distributed solution cost model facilitates efficient chain querying in a distributed Semantic Web environment, by enabling the application of our considered query optimization methods to distributed SPARQL chain queries. Our experiments on 34 SPARQL endpoints exhibit the competitiveness of the ACO query optimization approach, as compared to the 2PO and GA methods. For most smaller chain queries consisting of up to ten joins, the ACO method produces the best query plans in the least amount of time. For larger chain queries, each of our considered algorithms may have its benefits, depending on the join method used. When using the least naive join method, i.e., AGJoin, a GA approach is the most promising method, as it produces solutions of a quality similar to the quality of ACO solutions in significantly less time than both ACO and 2PO.

Our considered ACO algorithm has more potential than our current experiments demonstrate, as the algorithm can be run continuously in order to make real-time adaptations to changes in the environment, such as fluctuating availability and accessibility of SPARQL endpoints or ever-changing data. Therefore, in future work, we plan to optimize the scalability of the ACO algorithm and to subsequently evaluate all of our methods in a setting in which the algorithms can be run continuously. This would allow for realistic, continuously updated cardinality estimations as well. Another direction of future research is to implement our work in a SPARQL 1.1 engine, which supports federated queries.



(a) Solution costs for optimized solutions.



(b) Execution times of optimization.

Figure 5: Performance of our considered methods over chain queries with 2 to 19 joins, averaged over 1,000 queries for each query size, when using AGJoins.

Last, we would like to extend our work to be able to perform query optimization for other types of queries, e.g., star queries or cyclic queries.

7. REFERENCES

[1] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *10th International Conference on The Semantic Web (ISWC 2011)*, volume 7031 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2011.

[2] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing Linked Datasets with the VoID Vocabulary, 2011. Available online, `http://www.w3.org/TR/2011/NOTE-void-20110303/`.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[4] M. Dorigo and T. Stutzle. *Ant Colony Optimization*. MIT Press, 2004.

[5] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 6 edition, 2010.

[6] F. Frasincar, G. Houben, R. Vdovjak, and P. Barna. RAL: An Algebra for Querying RDF. *World Wide Web Journal*, 7(1):83–109, 2004.

[7] O. Gorlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *2nd International Workshop on Consuming Linked Data (COLD 2011)*, volume 782 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[8] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries across Diverse Data Sources. In *23rd International Conference on Very Large Data Bases (VLDB 1997)*, pages 276–285. Morgan Kaufmann Publishers Inc., 1997.

[9] A. Harth, L. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data Summaries for On-Demand Queries over Linked Data. In *19th International World Wide Web Conference (WWW 2010)*, pages 411–420. ACM, 2010.

[10] A. Hogenboom, F. Frasincar, and U. Kaymak. Ant Colony Optimization for RDF Chain Queries for Decision Support. *Expert Systems with Applications*, 40(5):1555–1563, 2013.

[11] A. Hogenboom, V. Milea, F. Frasincar, and U. Kaymak. RCQ-GA: RDF Chain Query Optimization Using Genetic Algorithms. In *10th International Conference on Electronic Commerce and Web Technologies (EC-Web 2009)*, volume 5692 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2009.

[12] Y. Ioannidis and Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *1990 ACM SIGMOD International Conference on Management of Data (SIGMOD 1990)*, pages 312–321. ACM, 1990.

[13] G. Klyne and J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax – W3C Recommendation 10 February 2004, 2004.

[14] J. Lee, D. Kim, and C. Chung. Multi-Dimensional Selectivity Estimation Using Compressed Histogram Information. In *1999 ACM SIGMOD International Conference on Management of Data (SIGMOD 1999)*, pages 205–214. ACM, 1999.

[15] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality Estimation and Dynamic Length Adaptation for Bloom Filters. *Distributed and Parallel Databases*, 28(2):119–156, 2010.

[16] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF – W3C Recommendation 15 January 2008, 2008.

[17] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *5th European Semantic Web Conference (ESWC 2008)*, volume 5021 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2008.

[18] P. Selinger, M. Astrahan, D. Chamberli, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *1979 ACM SIGMOD International Conference on Management of Data (SIGMOD 1979)*, pages 23–34. ACM, 1979.

[19] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *The VLDB Journal*, 6(3):191–208, 1997.

[20] H. Stuckenschmidt, R. Vdovjak, J. Broekstra, and G. Houben. Towards Distributed Processing of RDF Path Queries. *International Journal of Web Engineering and Technology*, 2(2-3):207–230, 2005.