Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data

Maribel Acosta[1] and Maria-Esther Vidal[2]

[1] Institute AIFB, Karlsruhe Institute of Technology
maribel.acosta@kit.edu
[2] Universidad Simón Bolívar
mvidal@ldc.usb.ve

Abstract. Client-side query processing techniques that rely on the materialization of fragments of the original RDF dataset provide a promising solution for Web query processing. However, because of unexpected data transfers, the traditional optimize-then-execute paradigm, used by existing approaches, is not always applicable in this context, i.e., performance of client-side execution plans can be negatively affected by live conditions where rate at which data arrive from sources changes. We tackle adaptivity for client-side query processing, and present a network of Linked Data Eddies that is able to adjust query execution schedulers to data availability and runtime conditions. Experimental studies suggest that the network of Linked Data Eddies outperforms static Web query schedulers in scenarios with unpredictable transfer delays and data distributions.

1 Introduction

The Linking Open Data cloud has experienced an impressive growth over the last decade [11], and consequently, the number of Linked Data applications is progressively increasing [6]. Although this situation evidences the success of Linked Open Data movements, it also encourages the Semantic Web community to urgently develop computational tools that effectively manage Linked Data.

Managing Linked Data usually requires accessing RDF datasets through specific Web access interfaces, e.g., SPARQL endpoints [5] or Triple Pattern Fragments (TPFs) [15]. SPARQL endpoints allow users to pose any SPARQL query against SPARQL servers, whereas TPFs are specific for triple-patterns, and their evaluations can be paged and retrieve metadata about the fragment page size, and the approximated fragment size. Further, SPARQL query engines implement data management techniques and execute queries against these Web access interfaces. Examples include federated query engines for SPARQL endpoints [1, 7, 12], and the client-side SPARQL query engine [15] against TPF servers.

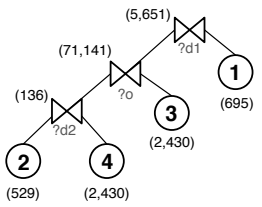
Despite these developments, the Web-alike characteristics of Linked Data sources impose fundamental challenges on Linked Data management. The lack of statistics about selectivities and data distributions, unpredictable data transfer rates and server workload, can negatively impact the effectiveness of query

engines against Linked Data, even in presence of the innovative querying capabilities offered by SPARQL endpoints and TPFs. This problem is mainly generated because existing Linked Data query engines implement execution query strategies that rely in some way, on the traditional *optimize-then-execute* paradigm, instead of following adaptive query strategies that adjust query executions to unexpected data source conditions. Thus, our main research problem is to devise adaptive query processing techniques that exploit properties of Linked Data technologies, and opportunistically adjust schedulers according to data availability and runtime conditions. Thus, query plans will be changed on a tuple-by-tuple basis, and answers will be produced as soon as they become available.

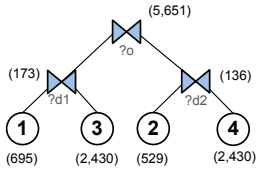
Adaptive query processing strategies have been extensively studied in the context of heterogeneous databases [3, 4, 10]. They can be divided into intra- and inter-operator solutions, and routing operators. Additionally, adaptivity can be implemented at different granularity levels: Fine-grained granularity indicates adaptation of small processes, e.g., per-tuple basis; while granularity is coarse-grained whenever adaptivity is attempted for large processes. Intra-operator techniques implement fine-grained granularity adaptivity, even in the context of a fixed query plan. Contrary, inter-operator techniques re-schedule initial plans based on: uncertainties in the execution cost, size of intermediate results, and unexpected delays. Finally, eddies [9] are routing operators that continuously reorder a query execution, by routing each intermediate tuple through the query operators in a variety of orders that simulate different query plans. Routing policies determine the routing destination of intermediate tuples. Eddies can be executed in a distributed fashion to avoid bottlenecks of a centralized eddy [13].

Building on these query processing strategies, we devise a novel client-side query processing engine that builds a network of Linked Data Eddies (nLDE) to opportunistically execute SPARQL queries against TPF servers. First, an nLDE relies on TPF metadata [15] to identify an initial bushy tree plan that reduces intermediate results. Leaves of the plan are grouped in star-shaped subtrees and internal nodes represent adaptive physical operators. Thus, intra-operator adaptivity is initially achieved. Simultaneously, eddies are created and empowered with Linked Data metadata to route tuples through the adaptive operators by following a pipeline strategy. We propose an innovative eddy routing policy that considers well-known SPARQL optimization heuristics [14]. In our approach, eddies are autonomous and any of them can produce query answers from tuples that have been already routed through all the nLDE adaptive operators. In this way, nLDE addresses adaptivity by executing different plans per tuple.

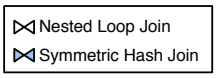
We empirically study the effectiveness of our network of Linked Data Eddies engine (nLDE engine) on SPARQL queries against RDF data exposed via TPF servers. Under the assumption of networks with no delays, we compare our query optimization techniques and adaptive strategies with the current TPF client. Experimental outcomes suggest that nLDE plans conduce to execution schedulers able to overcome drawbacks caused by the lack of data distributions even for queries with large intermediate results. Furthermore, we study the performance of our nLDE engine in presence of data transfer delays. The observed results con-



(a) Left-linear plan



(b) Bushy tree plan



Metrics	Left-linear	Bushy tree
Execution Time (sec.)	318.90	3.03
Results	1,398	5,651
Requests	1,693	67

(c) Execution results

Fig. 1. Different query plans to execute the query from Listing 1.1. The number of intermediate results produced by each operator are enclosed in parenthesis.

firm that routing adaptive query processing strategies provide a flexible solution for Linked Data management in unpredictable environments.

This paper comprises five additional sections. The following section illustrates a motivating example. We then define our approach in Section 3, and Section 4 presents experimental results. The related work is summarized in Section 5. Finally, we conclude in Section 6 with an outlook to future work.

2 Motivating Example

Consider the query from Listing 1.1 to *retrieve the drugs classified as DBpedia and Yago alcohols that share same routes of administration* to be executed using the TPFs for the English version of DBpedia.[1] The page size of these fragments is 100 and further metadata for each triple pattern is shown in Listing 1.1.

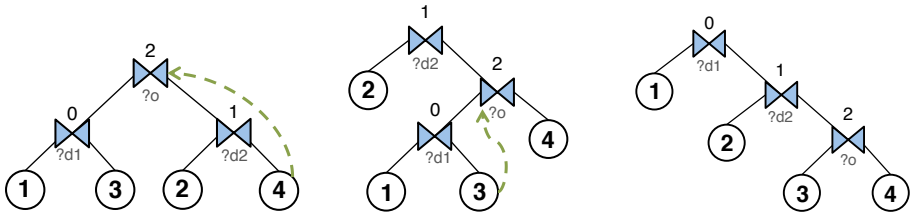
Listing 1.1. SPARQL query against DBpedia to retrieve information about resources classified as alcohols. Prefixes are used as in http://prefix.cc/

```
0 SELECT * WHERE {
1  ?d1 dcterms:subject dbpedia:Category:Alcohols. # Count:  695
2  ?d2 rdf:type yago:Alcohols.                    # Count:  529
3  ?d1 dbprop:routesOfAdministration ?o.          # Count: 2430
4  ?d2 dbprop:routesOfAdministration ?o. }        # Count: 2430
```

We executed the query from Listing 1.1 using first the current TPF client[2], which follows a combination of *left-linear* plans with Nested Loop Joins to evaluate the query, as depicted in Figure 1(a). In this approach, the triple pattern with the smallest cardinality (Count) is executed first; in our example, this corresponds to tp_2 with approximately 529 results. For each binding of tp_2, the TPF client instantiates the next triple pattern, in our example this would be tp_4, and retrieves all the resulting fragments. The execution continues with this strategy for each tuple of the intermediate results. The results of executing the example

[1] http://fragments.dbpedia.org/2014/en

[2] https://github.com/LinkedDataFragments/Client.js



(a) Tuples from tp_4 are routed to join operator 2, instead of operator 1

(b) Tuples from tp_3 are routed to join operator 2

(c) Plan generated by routings from Figures 2 (a) and (b)

Fig. 2. Diverse execution plans generated by re-ordering the execution of operators during query execution. Dashed lines represent routing of tuples to operators.

SPARQL query are reported in the table of Figure 1(c). The execution stopped after 318.90 seconds, produced 1,398 results, and performed 1,693 requests.

Consider now executing the example query with the plan depicted in Figure 1(b). The shape of this plan corresponds to a *bushy tree* in which several subtrees can be executed simultaneously, reducing the number of intermediate results. For instance, the left-linear plan in Figure 1(a) for the example query produces $136 + 71,141 = 71,277$ intermediate results, while the bushy tree plan in Figure 1(b) for the same query produces $173+136 = 309$ intermediate results. Moreover, joining the results with a symmetric operator is less expensive in this case considering the cardinalities and page size of the fragments. For instance, joining tp_2 and tp_4 with a Nested Loop Join results in $\sim$ 535 requests (6 requests to retrieve the fragment of tp_2 plus 529 requests for each binding), while performing a Symmetric Hash Join generates only $\sim$ 31 requests (6 requests for tp_2 plus 25 requests for tp_4). The execution of the bushy tree plan successfully finalized in 3.03 seconds, and produced 5,651 results[3] with 67 requests.

These results were obtained under the assumption of a network with no delays. However, even efficient plans, like the one from Figure 1(b), can be affected under the presence of data transfer delays. To illustrate, consider that the source that resolves tp_2 becomes very slow; then, tuples retrieved for tp_4 can be routed to another join operator as depicted in Figure 2(a). The result of re-routing tuples from tp_4 is a new plan shown in Figure 2(b), in which the delayed source is evaluated at the end. The plan can further change, as depicted in Figure 2(c). We executed the plan from Figure 2(a) on a network with a total delay[4] of 1.99 seconds. When implementing the adaptivity presented in Figure 2, all the results were produced in 3.86 seconds, which suggests that adaptivity was able to hide 1.16 seconds of the total delay. We tackle adaptivity in Linked Data manage-

[3] The same number of results was obtained when executing the query against the DBpedia endpoint at `http://dbpedia.org/sparql`.

[4] Sum of all elapsed waiting times between receiving a fragment page i and the subsequent page $i + 1$.

ment, and propose a client-side query engine that builds a network of routing operators able to adjust execution schedulers to this type of scenarios.

3 Our Approach

We devise a query processing engine tailored to issue SPARQL queries in which RDF sources are accessed in a triple-pattern fashion. In particular, we focus on optimizing and executing queries against Triple Pattern Fragment (TPF) servers [15]. The main components of our engine are: i) The **query optimizer**, tailored to reduce the number of intermediate results and requests posed to the data source; and ii) The **adaptive routing query engine** that implements a network of eddies, able to dynamically adapt the optimized plan according to current execution conditions, e.g., network delays or unpredictable selectivities.

3.1 Query Optimizer

We propose a query optimizer to devise physical plans that can be efficiently executed against TPF servers, and make use of the metadata provided in each fragment. Given a query Q, our optimizer (see Algorithm 1) starts retrieving metadata for each triple pattern in Q (lines 1-2); in particular, it selects the estimated number of triples or cardinality of the fragment (*count*) and the number of triples accessed per fragment page (*pagesize*). Then, the algorithm orders the triple patterns according to their *count* value (line 3). Following our example query Q from Listing 1.1, triple patterns are ordered as follows: $Q' = \langle tp_2, tp_1, tp_3, tp_4 \rangle$.

The optimizer then proceeds in three phases as follows. In the first phase, the algorithm groups triple patterns as *star-shaped groups* (SSGs), i.e., sets of triple patterns that share one variable;[5] SSGs can be efficiently executed against RDF data [16]. The optimizer starts by selecting the first triple pattern of the list Q' (line 6), i.e., s is the pattern with the smallest cardinality, which in our example is tp_2. Then, s is joined with a triple pattern tp'_i in Q' that shares variables in common. If the number of accesses to retrieve the fragment of tp'_i is less than the estimated number of instances in s, then the optimizer places a Symmetric Hash Join ($\bowtie_{SHJ}$), otherwise a Nested Loop Join ($\bowtie_{NL}$) is placed (lines 10-13). For instance, as shown in Section 2, ($tp_2 \bowtie_{NL} tp_4$) results in 535 requests, while ($tp_2 \bowtie_{SHJ} tp_4$) generates only 31 requests. The value *count* of the star is updated (line 14) with an estimation of the number of intermediate results that will be generated, i.e., *cardinalityEstimation*. In the absence of selectivity factors of triple patterns, we empirically tested different estimators (sum, product, and maximum) to approximate *cardinalityEstimation*; we selected the sum since it provided a more realistic estimation. This stage is completed when all triple patterns in Q belong to a SSG. The result of this stage is the set $\mathcal{S}$ with SSGs, which in our running example would be: $\mathcal{S} = \{(tp_2 \bowtie_{SHJ} tp_4), (tp_1 \bowtie_{SHJ} tp_3)\}$.

In the second phase, the optimizer builds bushy tree plans by combining subtrees created so far, e.g., the star-shaped groups identified previously. In

[5] A star-shaped group can be composed of only one triple pattern.

Algorithm 1: Physical Optimizer

```
   Input: Query Q = {tp_1, tp_2, ..., tp_n}
   Output: Bushy tree plan P_Q for Q
   // Get triple pattern metadata
 1 for tp_i ∈ Q do
 2 |  (tp_i.count, tp_i.pagesize) ← getMetadata(tp_i)
   // Order Q such that tp'_i.count ≤ tp'_{i+1}.count
 3 Q' ← ⟨tp'_1, tp'_2, ..., tp'_n⟩
   // Phase 1: Build index star-shaped groups (SSG)
 4 S ← ∅
 5 while Q'.length() > 0 do
 6 |  s ← Q'.getFirst()
 7 |  vars_s ← vars(s)
 8 |  for tp'i in Q' do
 9 |  |  if |vars_s ∩ vars(tp'_i)| = 1 then
10 |  |  |  if (tp'_i.count/tp'_i.pagesize) ≤ s.count then
11 |  |  |  |  s ← (s ⋈_SHJ tp'_i)
12 |  |  |  else
13 |  |  |  |  s ← (s ⋈_NL tp'_i)
14 |  |  |  s.count ←cardinalityEstimation(s.count, tp'i.count)
15 |  |  |  Q'.remove(tp'i)
16 |  S ← S ∪ {s}
   // Phase 2: Build bushy tree to combine SSGs with common variables
17 P_Q ← S
18 do
19 |  P'_Q ← P_Q
20 |  Select s_i and s_j from P_Q such that vars(s_i) ∩ vars(s_j) ≠ ∅
21 |  P_Q ← P_Q − {s_i, s_j}
22 |  P_Q ← P_Q ∪ {(s_i ⋈_SHJ s_j)}
23 while P'_Q ≠ P_Q
   // Phase 3: Place joins between SSGs with no common variables
24 do
25 |  Select s_i and s_j from P_Q
26 |  P_Q ← P_Q − {s_i, s_j}
27 |  P_Q ← (P_Q ⋈_SHJ s_j)
28 while |P_Q| > 1
29 return P_Q
```

order to join two subtrees, the subtrees must share at least one variable in common (line 20). Following the running example, subtrees ($tp_2 \bowtie_{SHJ} tp_4$) and ($tp_1 \bowtie_{SHJ} tp_3$) are joined since they share the variable ?o. All subtrees are joined in this stage with Symmetric Hash Join operators; which allows for executing different subtrees of the plan simultaneously. This stage finishes when no subtrees can be further combined (line 23). The outcome is a set of bushy trees P_Q.

Finally, in the third stage, subtrees that could not be joined before (since they share no variable in common) are combined. For the example query, our algorithm managed to build the efficient plan from Figure 1(b). In general, the optimizer produces a bushy tree plan P_Q for Q that allows for reducing intermediate results, and opportunistically places join operators aiming at reducing the number of requests to the sources.

3.2 Adaptive Routing Query Engine

The plan P_Q devised by the optimizer is then executed by the adaptive query engine designed to operate in unpredictable environments. Our query engine per-

forms *routing operator* adaptivity [9], able to change the order of the initial plan according to the current conditions of execution. Tuples generated during query execution can be routed to physical operators following a different order than the one designated by the optimizer, but respecting the relationships between operators in P_Q. In our engine, adaptivity is performed on a tuple-based basis.

In order to perform this routing adaptivity, physical operators used to execute the plan P_Q should follow a *pipelining* strategy [9], i.e., able to produce tuples incrementally as soon as data from a source become available. This type of operators are denominated *adaptive operators*. Considering that P_Q contains n adaptive operators, each operator is identified with a different label from 0 to $n-1$. For example, in Figure 2, the label of the Join operator between tp_1 and tp_3 is 0. In addition, each operator has a priority initially given by the execution order induced by P_Q, but operator priorities are updated as the execution goes on. In the following we define an adaptive operator in our query engine.

Definition 1 (Adaptive Operator). *Given an initial query plan P_Q for a query Q, an adaptive operator o is a physical non-blocking operator in P_Q. Each operator o in P_Q is annotated with two numbers denoted by label(o) and priority(o), such that:*

- *label(o) corresponds to an identifier of o in P_Q and is unique;*
- *priority(o) represents the priority of o in P_Q and induces the order in which o has to be executed in P_Q.*

During query execution, tuples are sent from adaptive operators to eddies. An eddy [2] is an operator that serves as a tuple router, that dynamically flows tuples through plan operators. To do so, eddies rely on tuple annotations denominated *Ready* and *Done* vectors. The *Ready* vector of a tuple indicates operators eligible to process that tuple. In our running example, tuples resulting from tp_1 should be processed by operators 0 and 2, but not by operator 1 – according to the plan from Figure 2(a); therefore, the *Ready* vector of these tuples is 101. The *Done* vector of a tuple indicates the operators that have already processed that tuple. For instance, if a tuple has only been processed by operator 1, then its *Ready* vector is 010. All tuples that flowed into an eddy e are introduced into a routing buffer RB_e, and are routed to the next adaptive operator following a routing policy RP_e (cf. Section 3.3). Operators that have not processed a tuple t in RB_e are computed by performing the bitwise operation $Ready_t - Done_t$; then, one adaptive operator is selected by its priority according to the implemented routing policy RP_e. Figure 3 illustrates the components RB_e and RP_e of an eddy. In this example, the tuple t = {d1=dbpedia:Bupranolol, o="Oral, topical"} in RB_e is annotated with *Ready*=101 and *Done*=100; RP_e decides to route the tuple to operator 2 since it is the only operator that has not processed t yet.

Eddies in our approach are enhanced with the capability of directly outputting results when a tuple has been processed by all operators. This allows for pipelining final results efficiently. In contrast, in the distributed eddies proposed by Tian and DeWitt [13], final results are routed to an intermediary eddy (*eddy sink*). When queries produce large amount of results, the eddy sink could become

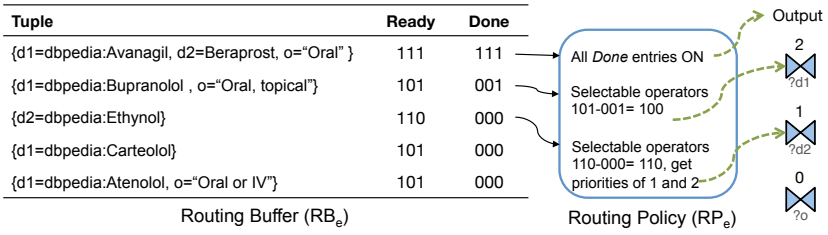


Fig. 3. Eddy operator e: Tuples are inserted into the Routing Buffer (RB_e), annotated with *Ready* and *Done* vectors. The Routing Policy (RP_e) selects the operator to route tuple t. Eddy outputs a tuple when it has been processed by all operators (*Done*=111)

a bottleneck, while in our approach the final output is produced in parallel by several autonomous eddies. In the following, we provide a definition of an eddy.

Definition 2 (Eddy Operator). *Given an initial query plan P_Q with n adaptive operators. An eddy e to execute P_Q is defined as a 2-tuple=(RB_e, RP_e) where RB_e corresponds to a routing buffer and RP_e is a routing policy. RB_e contains a set of tuples generated during the execution of P_Q. Each tuple t in RB_e is annotated with a pair of n-bit vectors named $Ready_t$ and $Done_t$, such that:*

- *A value of ON in the entry i of the $Ready_t$ vector of t indicates that t should be processed by the adaptive operator o such that $label(o) = i$.*
- *A value of ON in the entry i of the $Done_t$ vector of t indicates that t has been already processed by the adaptive operator o such that $label(o) = i$.*
- *t is produced as an output of the evaluation of P_Q when all entries in its $Done_t$ vector are ON (e is autonomous).*

RP_e is a function to route tuples from the eddy e to adaptive operators of P_Q. RP_e receives a tuple t in RB_e and outputs the identifier $label(o)$ of the adaptive operator o where t will be sent to.

Our query engine implements an adaptive network to execute query plans, called *network of Linked Data Eddies* (nLDE). An nLDE is composed of a set of adaptive operators and a set of eddies that dynamically send tuples among each other, constructing a bipartite graph G (see Figure 4). The number of adaptive operators is given by the plan to be executed. An eddy can get "clogged" when non-selective queries are executed against sources, and the transfer rate is faster than what the eddy is able to process. In order to avoid a "clogged" eddy, several eddies can be part of an nLDE such that the workload is distributed. This is particularly important when executing non-selective queries in which large amounts of intermediate results (tuples) have to travel through the network. Future work could focus on studying the optimal number of eddies in a network given the characteristics of a query, or even creating eddies on demand.

Figure 4 depicts an nLDE with two eddies for the query plan from Figure 2(a) of our running example. Edges in graph G from eddies to adaptive

operators indicate that tuples were sent through these routes. Assuming that Eddy 0 is the one depicted in Figure 3, the nLDE contains an edge from Eddy 0 to operators 2 and 1 since tuples {d1=dbpedia:Bupranolol, o="Oral, topical"} and {d2=dbpedia:Ethynol} were routed to these operators, respectively. Analogously, an edge from an adaptive operator to an eddy indicates that at least a tuple was sent through that route. For instance, Figure 4 depicts an edge from the Join operator with label 0 to the Eddy 0. When inspecting the routing buffer of Eddy 0 (Figure 3), the tuple {d1=dbpedia:Bupranolol, o="Oral, topical"} is annotated with *Done*=001, indicating that this tuple was only processed by the operator with label 0, therefore this operator was the one that sent the tuple to Eddy 0.

Besides eddies and adaptive operators, nLDE takes into consideration the characteristics of SPARQL queries and properties of Linked Data sets accessed to resolve different portions of a query. This information is denominated *Triple Pattern Descriptor* (TD) and consists of annotating the triple patterns from the query with metadata. A TD is then exploited by eddies in an nLDE to devise efficient routes to process RDF data. Figure 4 illustrates the TD for our running example: Triple patterns of the query are annotated with their corresponding cardinality (number of triples) and with the position of joins (e.g., joins by subject-subject and object-object) with other patterns. However, one important factor when executing queries is the selectivity of operators: Operators with high selectivity produce less intermediate results. Due to skewed data distribution in RDF datasets, selectivity may vary depending on the RDF resources that are being processed and cannot be *a priori* estimated by solely analyzing triple pattern cardinalities. We propose therefore an eddy routing policy (cf. Section 3.3) tailored for RDF data that considers not only the productivity of operators but also the position of joins in SPARQL queries [14] to favor the routing of tuples to join operators where the estimated selectivity is high. In the following, we define a network of Linked Data Eddies and its components.

Definition 3 (Network of Linked Data Eddies). *Given a query Q and a query plan P_Q for Q, a network of Linked Data Eddies for P_Q is a 2-tuple $nLDE = (G, TD)$, where G is a bipartite graph $G = (E \cup O, V)$ and TD is a triple pattern descriptor. E is a set of eddy operators, O is the set of adaptive physical operators in P_Q, and V is a set of directed edges, such that:*

- $V \subseteq (E \times O) \cup (O \times E)$.
- *If (e, o) belongs to V then the eddy e has routed at least one tuple to the adaptive operator o.*
- *If (o, e) belongs to V then the adaptive operator o has sent at least one tuple to the eddy e.*

TD corresponds to a set of pairs (tp, $\mathcal{M}_{tp}$), where tp is a triple pattern of Q and $\mathcal{M}_{tp}$ corresponds to metadata of tp. Example of metadata properties could be: join position, RDF data source, cardinality, and fragment page size.

In order to ensure the correct processing of tuples, eddies and adaptive operators should respect a set of rules. For instance, eddies cannot route a tuple to

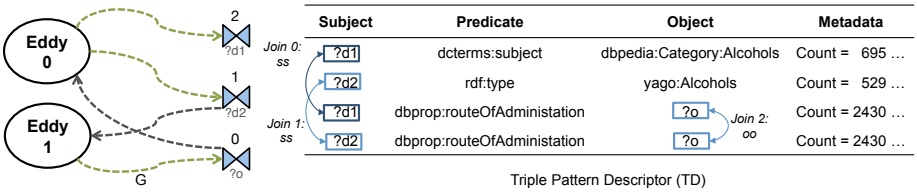


Fig. 4. Network of Linked Data Eddies (nLDE). Eddies and adaptive operators constitute a bipartite graph G. Edges in G represent routing paths of tuples. The Triple Pattern Descriptor (TD) of an nLDE maintains information about triple patterns from the query: metadata and operator position, e.g., subject-subject (ss), object-object (oo)

an arbitrary operator, but it has to consider the processing history of the tuple – given by its *Ready* and *Done* vectors. This restriction is defined in the following.

Definition 4 (Routing Rule from Eddy to Adaptive Operator). *Given an eddy operator $e = (RB_e, RP_e)$ and a set of adaptive operators O in an nLDE, RP_e routes tuples from RB_e according to the following rule:*

- *e can route a tuple t in RT_e to an adaptive operator $o \in O$ with identifier $label(o) = i$ only if $Ready_t[i] = ON$ and $Done_t[i] = OFF$; the set of operators that meet these conditions for t are denominated 'eligible operators of t'.*

Note that an adaptive operator has no restrictions on selecting an eddy to send a tuple to. However, before sending a tuple to an eddy, the adaptive operator has to build the *Ready* and *Done* vectors of the tuple. The correct creation of the *Ready* vector ensures that the tuple will not be processed more than once by an adaptive operator. Furthermore, the correct creation of the *Done* vector guarantees that the tuple will be processed by all the corresponding operators. In the following, we present the rules to create *Ready* and *Done* vectors of tuples.

Definition 5 (Rules to Create Ready and Done Vectors). *Given an adaptive operator o in an nLDE and a set of eddy operators E. Consider a tuple t produced by a binary operator o when combining tuples t_i and t_j. The tuple t is sent to an eddy operator $e \in E$ respecting the following rules:*

- *$Ready_t$ corresponds to the bitwise OR logical operation of the Ready vectors of tuples t_i, t_j, and the identifier of o represented by $label(o)$,*
- *$Done_t$ corresponds to the bitwise OR logical operation of the Done vectors of tuples t_i and t_j.*

In case the operator o is unary, $Done_t$ is updated by performing the bitwise OR logical operation with $label(o)$, while $Ready_t$ remains the same.

The execution of a query Q with an nLDE satisfies the following property:

Property 1 (Soundness). Given a query Q and a network of Linked Data Eddies $nLDE = (G = (E \cup O, V), TD)$ for a query plan P_Q. A tuple t produced by an eddy $e \in E$ belongs to the set of answers of the query Q if and only if all the entries of the $Done_t$ vector are equal to ON.

3.3 Routing Policies

Routing Policy from Eddy to Adaptive Operator. Tuples in the routing buffer of an eddy are processed following a strategy first-come, first-served (FCFS), i.e., oldest tuples are attended first. When a tuple t is routed from an eddy, the routing policy selects among the 'eligible operators of t' the one with the highest priority. Operator priorities are initialized according to the plan devised by the optimizer: Operators with the highest priority value should be executed first. In our running example, operators 0 and 1 have higher priority values than operator 2. During query execution, the priority of operator o with $label(o) = i$ is updated as follows: $priority(i) = 1 - \frac{\#tuples\ received\ from\ i}{\#tuples\ routed\ to\ i}$. Measuring the ratio of tuples produced vs. consumed by an operator allows for estimating its selectivity. When join operators exhibit similar performance, an operator is chosen over the others based on the join position specified in the triple pattern descriptor (TD) of the nLDE, following the HEURISTIC 2 by Tsialiamanis et al. [14]. Additionally, our routing policy respects the following restrictions: 1) Tuples are not routed to non-symmetric operators, otherwise the number of requests to sources could be increased; 2) Tuples are not routed to operators that do not share variables in common, to avoid the generation of large amount of tuples in the network.

Routing Policy from Adaptive Operator to Eddy. As explained in Section 3.2, there are no restrictions when routing tuples to eddies. However, when several eddies are part of an nLDE, it is important to design routing policies from adaptive operators that allow for distributing the workload among several eddies. In this work, we implement a simple routing policy in which an operator randomly chooses an eddy following a uniform distribution, i.e., all eddies have the same probability to be selected. We empirically tested this policy and observed that it is able to fairly spread tuples among eddies in the network.

4 Experimental Results

We empirically assess the effectiveness of a client-side network of Linked Data Eddies (nLDE engine) to adapt query execution schedulers to unknown data distributions and unexpected data transfer delays. The client-side Web query engine of Triple Pattern Fragments (TPF client) [15] is used as the baseline of the study. Below we describe the configuration settings used in our experiments.
Datasets and Query Benchmarks[6]**:** TPFs for the English version of DBpedia are used as RDF data servers. We designed two benchmarks of queries by analyzing triple patterns and sub-queries answerable for DBpedia. Benchmark 1 comprises 20 queries composed of basic graph patterns of between 4 and 14 triple patterns; these queries are non-selective and produce a large number of intermediate results. Benchmark 2 is composed of a total of 25 queries that have basic graph patterns of between three and six triple patterns; five queries about topics in five domains: *Historical*, *Life Sciences*, *Music*, *Sports*, and *Movies*.

[6] Benchmarks 1 and 2 are available at `http://people.aifb.kit.edu/mac/nlde/`.

Implementations: We implement proxies to configure data transfer delays. Both the nLDE engine and proxies are implemented in Python 2.7.6. We evaluate our experiments on a network with no delays, and in a fast network which is simulated with a gamma distribution ($\alpha = 1$, $\beta = 0.3$) of response latency resulting in an average latency of 0.3 secs. The setting 'nLDE (No Policy)' represents the basic query optimization (no adaptivity): The plan devised by the optimizer does not change. Experiments were executed on a Debian Wheezy 64 bit machine with CPU: 2x Intel(R) Xeon(R) CPU E5-2670 2.60GHz (16 physical cores), and 256GB RAM. Timeout was set to 1,800 secs.
Evaluation Metrics: The following metrics are computed separately for each benchmark. *i*) *Execution Time:* Elapsed time spent by a query engine to complete the execution of a query. It is measured as the absolute wall-clock system time as reported by the Python `time.time()` function. *ii*) *Number of Requests:* Total number of requests submitted to the servers during query execution. *iii*) *Number of Answers:* Total number of answers produced during the execution of a query plan. Queries were run five times and we report on the average time.

4.1 Effectiveness of nLDE Optimization Techniques

The goal of this study is to determining the impact that query selectivity and size of intermediate results have on the performance of client-side query engines in networks with no delays. We compare the nLDE engine with the TPF client on queries of Benchmark 1 and Benchmark 2. To compare the query optimization and execution techniques of both engines under the same conditions, the nLDE engine does not follow any routing policy, i.e., intermediate tuples are processed following the plan originally produced by the nLDE optimizer (Algorithm 1). Queries in Benchmark 1 are non-selective and produce a large number of results, while Benchmark 2 comprises very selective queries that produce a small number of results. Given the selectivity of queries in Benchmark 1, the timeout at 1,800 secs. is reached in some of the queries. Thus, we present the number of answers produced before timing out, in addition to the execution time. Figures 5(a) and (b) report on *Execution Time* and *Number of Answers* in logarithmic scale, respectively. We can observe that plans generated by the nLDE engine not only speed up the execution time, but they are able to produce more answers for the executed queries. The nLDE engine only consumes more time than the TPF client in queries `Q5, Q8, Q10,` and `Q11`, but as reported in Figure 5(b), the TPF client produces less number of answers than the nLDE engine in these queries. These results suggest that bushy trees comprised of star-shaped groups in conjunction with the nLDE adaptive operators, provide efficient execution schedulers to access TPF servers. Furthermore, we evaluate the overhead that these engines may cause to the data servers during query execution. Results of the execution of queries of Benchmark 2 are presented on Figures 6(a) and (b); because both engines produce all answers for each query, we just report on execution time and the number of requests submitted by each of the engines to the TPF servers. As can be seen, nLDE bushy plans speed up query schedulers by up to one order of magnitude, while they submit less requests to the TPF

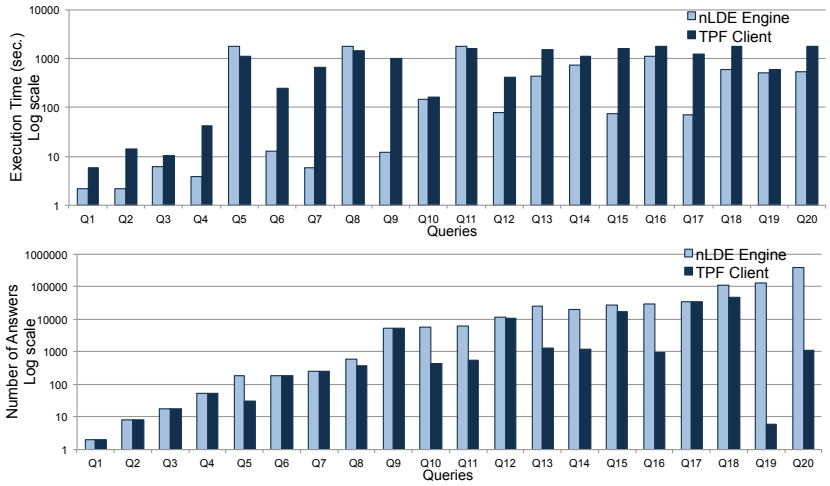


Fig. 5. Results for queries of Benchmark 1 for nLDE engine and TPF client; 20 non-selective queries against TPFs for the English version of DBpedia. a) Execution Time in secs. (log. scale), b) Number of Answers (log. scale). No delays in data transfer

servers in the majority of the queries. The reason for this is that left-linear plans as the ones generated by the TPF client in conjunction with Nested Loop Join operators, may produce a large number of intermediate results that conduce to large number of requests to the TPF servers. In contrast, bushy plans composed of star-shaped groups minimize the number of intermediate results and in consequence, submit a small number of requests to the TPF servers. Thus, the nLDE engine is able to retrieve data from the TPF servers in a more efficient fashion, providing in this way, an effective approach for Linked Data management even in ideal scenarios of simple queries (Benchmark 2) and networks with no delays.

4.2 Adaptivity of the nLDE Engine

The goal of this study is to evaluate the performance of the routing policies implemented by the nLDE engine. Networks with delays allow for evaluating the adaptivity of engines to unpredictable changes. We simulate a fast network where data transfer rates are configured to respect a gamma distribution with $\alpha = 1$, $\beta = 0.3$. Further, we compare the execution time of the nLDE engine when intermediate tuples are executed following the original plan (No Policy) and when execution schedulers are adapted to the data transfer rates according to our Routing Policy. Both instances of the nLDE engine produce the same number of query answers and server requests, so we report on *Execution Time* in milliseconds in Figure 7. As can be observed, the nLDE engine with the routing policy exhibits better performance than the nLDE engine with no policy. It is important to highlight that this scenario is quite troublesome for a routing policy. When queries produce a small number of intermediate results the policy might not have enough information to devise an efficient routing. Additionally,

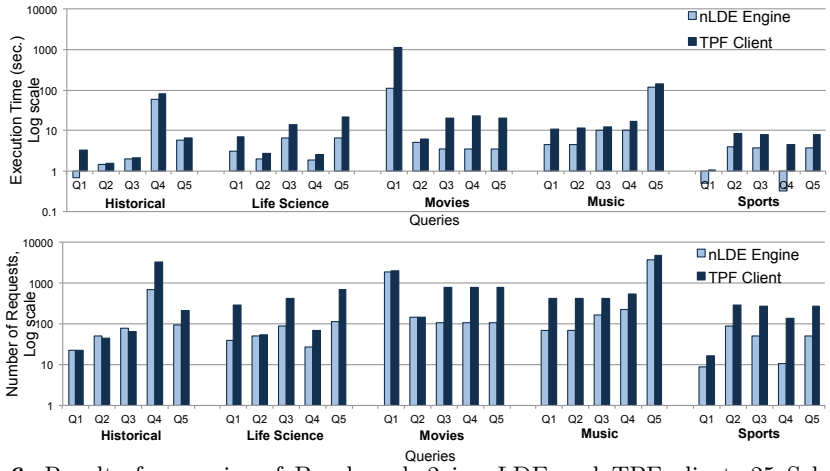


Fig. 6. Results for queries of Benchmark 2 in nLDE and TPF client; 25 Selective Queries against TFPs for the English version of DBpedia; Five Queries per Domain. a) Execution Time in secs. (log. scale), b) Number of Requests (log. scale)-No delays

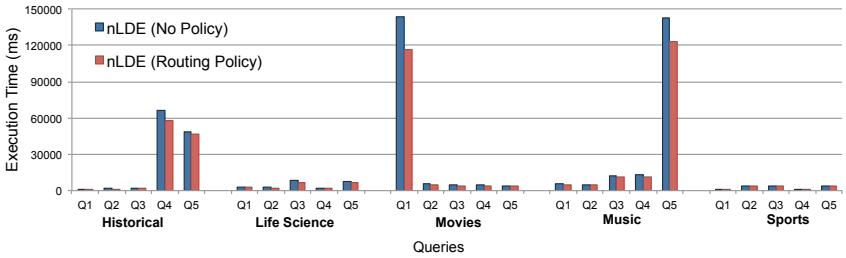


Fig. 7. Results for queries of Benchmark 2 in nLDE with No Routing Policy and nLDE with our Routing Policy. 25 Selective Queries against TFPs for the English version of DBpedia; Five Queries per Domain. Execution Time in msecs. Fast network simulated with a Gamma distribution ($\alpha = 1$, $\beta = 0.3$) of delays

when network is fast with a relative low latency the policy has to be lightweight enough to process tuples arriving with fast rates. Despite of these conditions and the overhead caused by routing intermediate results, the nLDE engine with our routing policy is able to faster produce the complete results of the studied queries. We hypothesize that even better performance will be observed in slower networks and in presence of messy data distributions.

5 Related Work

We analyze the *adaptivity granularity* achieved by Web query processing approaches that rely on HTTP interfaces to access RDF data.

SPARQL endpoints exploit SPARQL expressiveness and efficiently access RDF data. Nevertheless, they may suffer from typical Web-publishing problems,

i.e., connections may be slow or may, in the extreme, become unavailable. Existing federated engines, e.g., ANAPSID [1], FedX [12], and SPLENDID [7], implement adaptivity and mitigate in some way, the impact of these problems. FedX and SPLENDID, the adaptivity granularity is *coarse-grained*, supporting the generation of fixed logical query plans according to the available endpoints. In addition, ANAPSID implements a *fine-grained* granularity adaptivity, and provides an intra-operator strategy and non-blocking operators. Thus, ANAPSID detects when SPARQL endpoints become unavailable, and opportunistically produces results as quickly as data arrives from the endpoints. Although these adaptive query processing techniques may empower SPARQL endpoints, because the *optimize-then-execute* paradigm is followed, completeness of the query results or query execution efficiency is not always achieved. Contrary, our network of Linked Data Eddies implements *routing operator* strategies able to change the logical query plan according to the conditions of the RDF data sources.

Hartig et al. [8] propose a Linked Data traversal approach and provide an inter-operator approach where source selection and link traversal are interleaved during query execution time. A non-blocking iterator model that relies on an asynchronous pipeline of iterators is used for traversing relevant links. Iterators are executed in an order heuristically determined, e.g., the most selective iterators are executed first. Further, the query engine is able to adapt the execution to source availability by on-the-fly detecting whenever an HTTP server stops responding. This approach adapts execution schedulers to uncontrollable network conditions; nevertheless, in presence of arbitrary data distributions, plans cannot be adapted and performance may be negatively impacted.

Finally, Verborgh et al. [15] propose a novel HTTP interface to access RDF data that rely on *Linked Data Fragments* (*LDF*) which can be easily generated by RDF data providers. Verborgh et al. also present a client-side Web query processing strategy for *Linked Data Fragments* of triple patterns (TPFs). This client-side query engine enhances Web clients with the capability of executing SPARQL queries and implements the non-blocking iterator model proposed by Hartig et al. [8] to adapt the query execution scheduler to different cardinality distribution of the retrieved TPF servers. Adaptivity is implemented at the level of TPF pages by interleaving TPF server selection with TPF requests to ensure thus that requests of more selective pages are executed first. Although TPF clients may effectively adapt query schedulers to TPFs with arbitrary data distributions, data transfer delays can negatively impact their performance. In contrast, the nLDE engine relies on both metadata provided by TPFs and novel routing techniques to identify efficient query plans that reduce execution time and number of requests. Therefore, the nLDE engine dynamically adapts execution schedulers to changing conditions of the TPF servers.

6 Conclusions and Future Work

We have defined the nLDE engine, a client-side query processing engine that builds a network of Linked Data Eddies to efficiently access TPF servers. The

nLDE engine implements adaptivity at intra-operator levels as well as routing strategies that allow for the adaptation of execution schedulers to real-world conditions. Reported experimental results suggest that the nLDE engine is able to generate plans that not only increase the number of answers produced, but also reduce execution time and number of server requests. Moreover, in presence of unexpected data transfer delays, the nLDE engine is able to route intermediate results according to data availability and produce answers as soon as they are retreived from the servers. In the future, we plan to define different routing policies and cost models that better estimate selectivity of TPFs.

References

1. M. Acosta, M.-E. Vidal, J. Castillo, T. Lampo, and E. Ruckhaus. ANAPSID: An adaptive query processing engine for SPARQL endpoints. In *ISWC*, pages 18–34, 2011.
2. R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.
3. S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005.
4. A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
5. L. Feigenbaum, G. Williams, K. Clark, and E. Torres. SPARQL 1.1 protocol, 2013.
6. I. Fundulaki and S. Auer. Linked Open Data - Introduction to the special theme. *ERCIM News*, 2014(96), 2014.
7. O. Görlitz and S. Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *COLD Workshop*, 2011.
8. O. Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *ESWC*, pages 154–169, 2011.
9. J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23(2):7–18, 2000.
10. K. Laddhad and S. Sudarshan. Adaptive query processing. Technical Report 05329014, Kanwal Rekhi School of Information Technology, Indian Institute of Technology, Bombay, Mumbai, 2006.
11. M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the Linked Data best practices in different topical domains. In *ISWC*, pages 245–260, 2014.
12. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *ISWC*, pages 601–616, 2011.
13. F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.
14. P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for SPARQL. In *EDBT*, pages 324–335, 2012.
15. R. Verborgh, O. Hartig, B. D. Meester, G. Haesendonck, L. D. Vocht, M. V. Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. V. de Walle. Querying datasets on the Web with high availability. In *ISWC*, pages 180–196, 2014.
16. M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in SPARQL queries. In *ESWC*, pages 228–242, 2010.