ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints

Maribel Acosta[1], Maria-Esther Vidal[1], Tomas Lampo[2], Julio Castillo[1], and Edna Ruckhaus[1]

[1] Universidad Simón Bolívar, Caracas, Venezuela
{macosta,mvidal,ruckhaus}@ldc.usb.ve, julio@gia.usb.ve
[2] University of Maryland, College Park, USA
tlampo@cs.umd.edu

Abstract. Following the design rules of Linked Data, the number of available SPARQL endpoints that support remote query processing is quickly growing; however, because of the lack of adaptivity, query executions may frequently be unsuccessful. First, fixed plans identified following the traditional optimize-then-execute paradigm, may timeout as a consequence of endpoint availability. Second, because blocking operators are usually implemented, endpoint query engines are not able to incrementally produce results, and may become blocked if data sources stop sending data. We present ANAPSID, an adaptive query engine for SPARQL endpoints that adapts query execution schedulers to data availability and run-time conditions. ANAPSID provides physical SPARQL operators that detect when a source becomes blocked or data traffic is bursty, and opportunistically, the operators produce results as quickly as data arrives from the sources. Additionally, ANAPSID operators implement main memory replacement policies to move previously computed matches to secondary memory avoiding duplicates. We compared ANAPSID performance with respect to RDF stores and endpoints, and observed that ANAPSID speeds up execution time, in some cases, in more than one order of magnitude.

1 Introduction

The Linked Data publication guideline establishes the principles to link data on the Cloud, and make Linked Data accessible to others[3]. Based on these rules, a great number of available SPARQL endpoints that support remote query processing to Linked Data have become available, and this number keeps growing. Additionally, the W3C SPARQL working group is defining a new SPARQL 1.1 query language to respect the SPARQL protocol and specify queries against federations of endpoints [19]. However, access to the Cloud of Linked datasets is still limited because many of these endpoints are developed for very lightweight use. For example, if a query posed against a linkedCT endpoint [4] requires more than 3 minutes to be executed, the endpoint will timeout without producing any answer. Thus, to successfully execute real-world queries,

[3] http://www.w3.org/DesignIssues/LinkedData.html
[4] Clinical Trials data produced by the ClinicalTrials.gov site available at http://linkedCT.org. and http://hcls.deri.org/sparql.

it may be necessary to decompose them into simple sub-queries, so that the endpoints will then be capable of executing these sub-queries in a reasonable time. Additionally, since endpoints may unpredictably become blocked, execution engines should modify plans on-the-fly to contact first the available endpoints, and produce results as quickly as data arrives.

Several query engines have been developed to locally access RDF data [1, 10, 12, 17, 24]. The majority have implemented optimization techniques and efficient physical operators to speed up execution time [12, 17, 24]; others have defined structures to efficiently store and access RDF data [17, 25], or have developed strategies to reuse data previously stored in cache [1, 10, 17]. However, none of these engines are able to gather Linked Data accessible through SPARQL endpoints, or hide delays from users.

Recently several approaches have addressed the problem of query processing on Linked Data [2, 7, 9, 13–16, 21]; some have implemented source selection techniques to identify the most relevant sources for evaluating a query [7, 14, 21], while others have developed frameworks to retrieve and manage Linked Data [2, 8, 9, 13, 15, 16], and to adapt query processing to source availability [9]. Additionally, Buil-Aranda et al. [4] have proposed optimization techniques to rewrite federated queries specified in SPARQL 1.1, and reduce the query complexity by generating well-formed patterns. Finally, some RDF engines[18, 20] have been extended to query federations of SPARQL endpoints. Although all these approaches are able to access Linked Data, none of them can simultaneously provide an adaptive solution to access SPARQL endpoints.

In this paper we present ANAPSID, an engine for SPARQL endpoints that extends the adaptive query processing features presented in [22], to deal with RDF Linked Data accessible through SPARQL endpoints. ANAPSID stores information about the available endpoints and the ontologies used to describe the data, to decompose queries into sub-queries that can be executed by the selected endpoints. Also, adaptive physical operators are executed to produce answers as soon as responses from available remote sources are received. We empirically analyze the performance of the proposed techniques, and show that these techniques are competitive with state-of-the-art RDF engines which access data either locally or remotely.

The paper is comprised of six additional sections. We start with a motivating example in the following section. Then, we present the ANAPSID architecture in section 3 and describe the query engine in section 4. Experimental results are reported in section 5, and section 6 summarizes the related work. Finally, we conclude in section 7 with an outlook to future work.

2 Motivating Example

LinkedSensorData[5] is a dataset that makes available sensor weather data of approximately 20,000 stations around the United States. Each station provides information about weather observations; the ontology O&M-OWL[6] is used to describe these observations; a Virtuoso endpoint is provided to remotely access the data. Further, each

[5] http://wiki.knoesis.org/index.php/LinkedSensorData.
[6] http://knoesis.wright.edu/ssw/ont/sensor-observation.owl

station is linked to its corresponding location in *Geonames*[7].

Consider the acyclic query: *Retrieve all sensors that detected freezing temperatures on April 1st, 2003, between 1:00am and 3:00am*[8]. The answer comprises 1,600 sensors.

```
prefix om-owl:<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix weather:<http://knoesis.wright.edu/ssw/ont/weather.owl#>
prefix sens-obs:<http://knoesis.wright.edu/ssw/>
prefix xsd:<http://www.w3.org/2001/XMLSchema#>
prefix owl-time:<http://www.w3.org/2006/time#>
prefix gn:<http://www.geonames.org/ontology#>
SELECT DISTINCT ?sensor
WHERE {
?sensor om-owl:generatedObservation ?observation .
?observation rdf:type weather:TemperatureObservation .
?observation om-owl:samplingTime ?time .?time owl-time:inXSDDateTime ?xsdtime .
?observation om-owl:result ?result .?result om-owl:floatValue ?value .
?result om-owl:uom weather:fahrenheit .FILTER(?value <= "32.0"^^xsd:float).
FILTER(?xsdtime >= "2003-04-01T01:00:00-07:00^^http://www.w3.org/2001/XMLSchema#dateTime")
FILTER(?xsdtime <= "2003-04-01T03:00:00-07:00^^http://www.w3.org/2001/XMLSchema#dateTime").
?sensor om-owl:hasLocatedNearRel ?location .?location om-owl:hasLocation ?ga. ?ga gn:name ?name}
```

Using the *LinkedSensorData* endpoint[9], we executed several versions of the former query with different date ranges. Table 1 reports on the observed execution time values. Different instantiations of the SPARQL endpoint parameter SPARQL SPONGE [10] were set up to indicate the type of dereferences to be executed during query processing.

Table 1. Execution Time (secs.) of Queries Against the *LinkedSensorData* SPARQL Endpoint; SPONGE parameter: Local, Grab All, Grab All *seeAlso*, Grab Everything

	Local	Grab All	Grab All (*seeAlso*)	Grab Everything
Average	0.35	100.78	Timeout	Timeout
Standard Deviation	0.04	38.32	Timeout	Timeout
Minimum	0.30	58.95	Timeout	Timeout
Maximum	0.45	155.59	Timeout	Timeout

We can observe that if the query is run on data locally stored in the endpoint, i.e., SPONGE is equal to Local and only one endpoint is contacted, the queries can be executed in less than one second. However, if IRI's are dereferenced by using the *Grab All* option, the execution time increases in average two orders of magnitude. Moreover, if the *seeAlso* references are considered and the corresponding endpoints are contacted (Grab All *seeAlso*), the execution reaches a timeout of 86,400 secs. (one day). Similarly, if all the referred resources are downloaded (Grab Everything), the endpoint reaches the timeout without producing any answer. These results suggest that when the *LinkedSensorData* endpoint requires to download data from remote endpoints, it may become blocked waiting for answers; this may be caused by a blocking query processing model executed by existing endpoints.

[7] http://www.geonames.org/ontology.

[8] Time is specified in Mountain Time; temperature in Fahrenheit scale.

[9] http://sonicbanana.cs.wright.edu:8890/sparql

[10] http://docs.openlinksw.com/virtuoso/virtuososponger.html.

Traditionally, query processing engines are built on top of a blocking iterator model that fires a query execution process from the root of the execution plan to the leaves, and does not incrementally produce any result until its corresponding children have completely produced the answer. Thus, if any of the intermediate nodes becomes blocked while producing answers, the root of the plan will also be blocked. We consider plans whose leaves are endpoints; however, similar problems may occur, if leaves corresponds to URIs that need to be dereferenced.

To overcome limitations of existing execution models when Linked Data is dereferenced, some state-of-the-art approaches have proposed adaptive query engines that are able to produce answers as data becomes available [9, 14]. For example, Hartig et al. [9] extend the traditional iterator model and provide an adaptive query engine that hides delays that occur when any linked dataset becomes blocked. This adaptive iterator detects when a URI look-up stops responding, and resumes the query execution process executing other iterators; results can be incrementally produced, and delays in retrieving data during URI look-ups are hidden from the users. Further, Ladwig et al. [14] use a non-blocking operator to opportunistically produce answers as soon as dereferenced data is available. However, none of these approaches support the access to a federation of SPARQL endpoints. Finally, some RDF engines have been extended to deal with SPARQL queries against federations of endpoints[4, 18, 20], but no adaptive query techniques have been implemented, and queries are frequently unsuccessful when endpoints become blocked. In this paper we present an adaptive engine that makes use of information about endpoints, to decompose the query into simple sub-plans that can be executed by the remote endpoints. Also, we propose a set of physical operators that gather data generated by the endpoints, and quickly produce responses.

3 The ANAPSID Architecture

ANAPSID is based on the architecture of wrappers and mediators [26] to query federations of SPARQL endpoints (Figure 1).

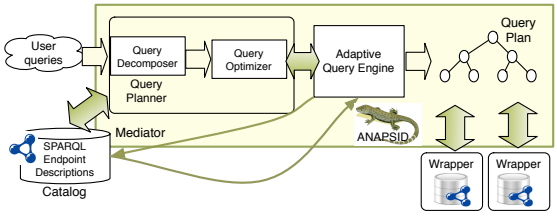


Fig. 1. The ANAPSID Architecture

Lightweight *wrappers* translate SPARQL sub-queries into calls to endpoints as well as convert endpoint answers into ANAPSID internal structures. *Mediators* maintain information about endpoint capabilities, statistics that describe their content and performance, and the ontology used to describe the data accessible through the endpoint.

Following the approach developed in previous work [11], the Local As View (LAV) approach is used to describe endpoints in terms of the ontology used in the endpoint dataset. Further, mediators implement query rewriting techniques, decompose queries into sub-queries against the endpoints, and gather data retrieved from the contacted endpoints. Currently, only SPARQL queries comprised of joins are considered; however, the rewriting techniques have been extended to consider all SPARQL operators, but this piece of work is out of the scope of this paper. Finally, mediators hide delays, and produce answers as quickly as data arrives; they are composed of the following components:

- *Catalog*: maintains a list of the available endpoints, their ontology concepts and capabilities. Contents are described as views with bodies comprised of predicates that correspond to ontology concepts; execution timeouts indicate endpoint capabilities. Statistics are updated on-the-fly by the adaptive query engine.
- *Query Decomposer*: decomposes user queries into multiple simple sub-queries, and selects the endpoints that are capable of executing each sub-query. Simple sub-queries are comprised of a list of triple patterns that can be evaluated against an endpoint, and whose estimated execution time is less than the endpoint timeout. Vidal et al. [24] suggest that the cardinality of the answer of sub-queries comprised of triple patterns that share exactly one variable, may be small-sized; so the query decomposer will try to identify low cost sub-queries that meet this property.
- *Query Optimizer*: identifies execution plans that combine sub-queries and benefits the generation of bushy plans composed of small-sized sub-queries. Statistics about the distribution of values in the different datasets are used to identify the best combination of sub-queries. These statistics and capabilities of the endpoints are collected by following an Adaptive Sampling Technique [3, 24], or on-the-fly during query execution.
- *Adaptive Query Engine*: implements different physical operators to gather tuples from the endpoints. These physical operators are able to detect when endpoints become blocked, and incrementally produce results as the data arrives. Additionally, the query engine can modify an execution plan on-the-fly to execute first the requests against the endpoints that are available; information gathered during runtime is used to update catalog statistics, and to re-optimize delayed queries.

4 The ANAPSID Query Processing Engine

The ANAPSID query engine provides a set of operators able to gather data from different endpoints. Opportunistically, these operators produce results by joining tuples previously received even when endpoints become blocked. Additionally, the physical operators implement main memory replacement policies to move previously computed matches to secondary memory, ensuring no duplicate generation. Each join operator maintains a data structure called Resource Join Tuple (RJT), that records for each instantiation of the join variable(s), the tuples that have already matched. Suppose that for the instantiation of the variable $?X$ with the resource r, the tuples $\{T_1, ..., T_n\}$ have matched, then the RJT will be the pair $(r, \{T_1, ..., T_n\})$, where the first argument, *head* of the RJT, corresponds to the resource and the second, *tail* of the RJT, is the list of tuples.

4.1 The Adaptive Group Join (agjoin)

The **agjoin** operator is based on the Symmetric Hash Join [5] and Xjoin [22] operators, defined to quickly produce answers from streamed data accessible through a wide-area network. Basically, the Symmetric Hash Join and Xjoin are non-blocking operators that maintain a hash table for the data retrieved from sources A and B. Execution requests against A and B are submitted in parallel, and when a tuple is generated from source A (resp. B), it is inserted in the hash table of A (resp. hash table of B) and probed against the hash table of B (resp. hash table of A). An output is produced each time a match is found. Further, the Xjoin implements a main memory replacement policy that flushes portions of the hash tables to secondary memory when they become full, and ensures that no duplicates are generated. Even though these operators produce results incrementally, results are produced one-by-one because tuples are first inserted in the corresponding hash table and then probed against the other hash table to find one match at a time. To speed up query answering, we propose the **agjoin** operator. The **agjoin** maintains for source A (resp. B) a list L_A (resp. L_B) of RJTs, which represents for each instantiation, $\mu(?X)$, of the tuples already received from source A, the tuples received from B that match $\mu(?X)$. L_A (resp. L_B) is indexed by the values of $\mu(?X)$ that correspond to the heads of the RJTs in L_A (resp. L_B); thus, **agjoin** provides a direct access to the RJTs. When a new tuple t with instantiation $\mu(?X)$ arrives from source A, **agjoin** probes against L_A to find an RJT whose head corresponds to $\mu(?X)$; if there is a match, the **agjoin** quickly produces the answer as the result of combining t with all the tuples in the tail of RJT of $\mu(?X)$; if not, nothing is added to L_A. Independently of the success of the probing process, t is inserted in its corresponding RJT in L_B. Figure 2 illustrates main memory contents during the execution of **agjoin** between sources A and B.

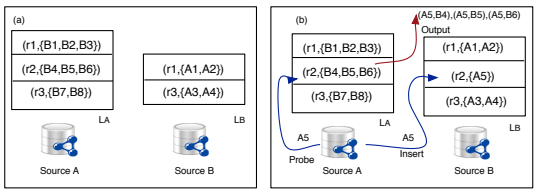


Fig. 2. agjoin between sources A and B: (a) L_A and L_B current state; (b) effects of arriving a tuple A5 from source A, three tuples are immediately produced, and RJT (r2,{A5}) is inserted in L_B.

L_A and L_B in Figure 2 (a) indicate that tuples (B1,A1), (B1,A2), (B2,A1), (B2,A2), (B3,A1), (B3,A2), (B7,A3), (B7,A4), (B8,A3), (B8,A4) have been already produced; also, at this time, no tuples with $\mu(?X) = r_2$ have been received from A, while three of these tuples have arrived from source B. Figure 2 (b) shows the current state of L_A and L_B after a tuple A5 with $\mu(?X) = r_2$ arrives from source A, i.e., shows the effects in L_B of arriving a new tuple A5 with $\mu(?X) = r_2$ from source A. In this case, A5 is probed against L_A and three outputs are produced immediately; concurrently, the insert process is fired, and RJT (r2,{A5}) is inserted in L_B.

Property 1 *Consider the current state of lists L_A and L_B in an instant t, the number of answers produced until t, NAP_t, is given by the following formula:*

$$NAP_t = \sum_{RJT_a \in L_A \wedge RJT_b \in L_B \wedge head(RJT_a)=head(RJT_b)} (|tail(RJT_a)| \times |tail(RJT_b)|).$$

A three-stage policy is implemented to flush RJTs; completeness and no duplicates are ensured. A first stage is performed while at least one source sends data; a second stage is fired when both sources are blocked, and the third is only executed when all data have completely arrived from both sources. Note that the same operator can execute first or second stages at different times and depending on the availability of the sources, it can move from one stage to the other; however, the third stage is executed only once.

In a first stage, when a tuple t arrives from source A, it is inserted in an RJT in L_B; the probe time of t in L_A and the insert time of t in L_B are stored with t. Further, if a portion of the main memory assigned to A becomes full, an RJT victim is chosen based on the time of the last probe; thus, the least recently probed RJT is selected, flushed to secondary memory, and annotated with the flush time. In case RJTs with the same head are chosen as victims at different times, only one RJT will be stored to secondary memory; the tail will be comprised of the tails of the different victim RJTs; these tails will be annotated with the respective flush time. Figure 3 illustrates the process performed when a main memory failure occurs and the timestamps of the stored tuples.

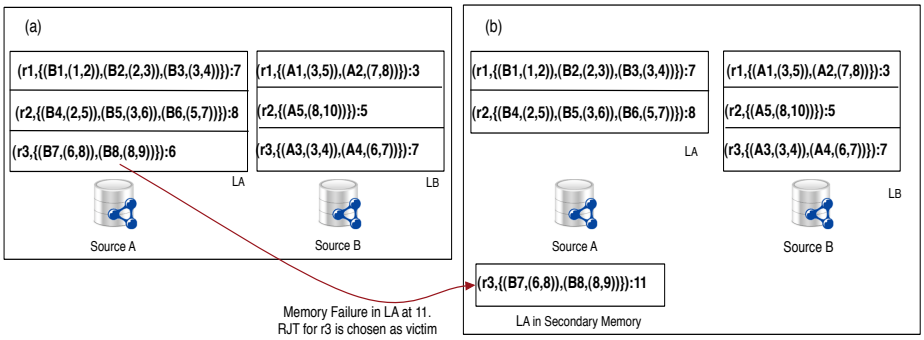


Fig. 3. Timestamp annotations and Main Memory Failures: (a) L_A and L_B timestamps; (b) effects of a main memory failure in L_A; RJT for r3 is flushed.

Figure 3 (a) illustrates the RJTs in Figure 2, annotated with the probe and insert times of the tuples, and the RJTs probe times[11]. Thus, we can say that B1 was probed at time 1 and inserted in L_A at 2; also, timestamp 7 associated with the RJT of r1 in L_A, indicates that the last probe of a tuple from source B was performed against this RJT at time 7. Further, suppose that a failure of memory occurs at time 11 in the portion of

[11] An RJT probe time corresponds to the most recent probe time of the tuples in the RJT.

main memory assigned to source A, then the RJT with head r3, is flushed to secondary memory and its flush time is annotated with 11. Figure 3 (b) illustrates the final state of L_A (main and secondary memory) and L_B after flushing the RJT to secondary memory. Definition 1 states the conditions to meet when tuples are joined during a first stage.

Definition 1 *Let RJT_i and RJT_j be Resource Join Tuples in L_A and L_B, respectively, such that, head(RJT_i)=head(RJT_j). Suppose RJT_j has been flushed to secondary memory. Then, a tuple $B_j \in$ tail(RJT_i) was matched to tuples of tail(RJT_j) during a first stage of the* **agjoin***, i.e., before RJT_j was flushed, if and only if:*

$$probeTime(B_j) < flushTime(RJT_j).$$

A second stage is fired when both sources become blocked; Definition 2 establishes the conditions to be satisfied by tuples that are matched in a second stage.

Definition 2 *Let RJT_i and RJT_j be Resource Join Tuples in L_A and L_B, respectively, such that, head(RJT_i)=head(RJT_j). Suppose RJT_j has been flushed to secondary memory. Then, a tuple $B_j \in$ tail(RJT_i) was matched to tuples of tail(RJT_j) during a second stage of the* **agjoin***, i.e., before RJT_i was flushed to secondary memory[12], if and only if, there is a second state ss:*

$$flushTime(RJT_j) < insertTime(B_j) < TimeSecondStage(ss) < flushTime(RJT_i).$$

To produce new answers during a second stage, the **agjoin** selects the largest RJTs in secondary memory, and probes them against their corresponding RJTs in main memory. To avoid duplicates, conditions in Definitions 1 and 2 are checked. The execution of a second stage is finished, when one source becomes unblocked, and all the RJTs in secondary memory are checked to find new matches. A global variable named *TimeLastSecondStage*, is maintained and updated when a second stage finishes; also, for each second stage, we maintain the time it was performed.

Suppose tuple t from RJT_i matches tuples in RJT_j in the second stage at time st, then the probe time of t and the probe time of its RJT in main memory are updated to st. To illustrate this process, consider the current state of L_A and L_B reported in Figure 3 (b); also suppose that the last second stage was performed at time 14. Following the policy to select RJTs in secondary memory, (r3,{(B7,(6,8)),(B8,(8,9))}) in the secondary memory version of L_A, is chosen and probed against (r3,{(A3,(3,4)),(A4,(6,7))}) in L_B; the RJT in secondary memory was chosen because it has the longest tail. Since conditions in Definition 1 hold for tuples B7 and B8, no new answers are produced and their timestamps are not changed. Finally, one of the sources becomes available at time 15, then the second stage finalizes, and *TimeLastSecondStage* is updated to 15.

The third stage is fired when data has been completely received. Tuples that do not satisfy conditions in Definitions 1 and 2 are considered to produce the rest of the answers. First, RJTs in main memory are probed with RJTs in secondary memory. Then, RJTs in secondary memory are probed to produce new results. Figure 4 illustrates states of L_A and L_B right after all the tuples have been received at time 100 and the third stage

[12] If an RJT is in main memory, then its flush time is ∞.

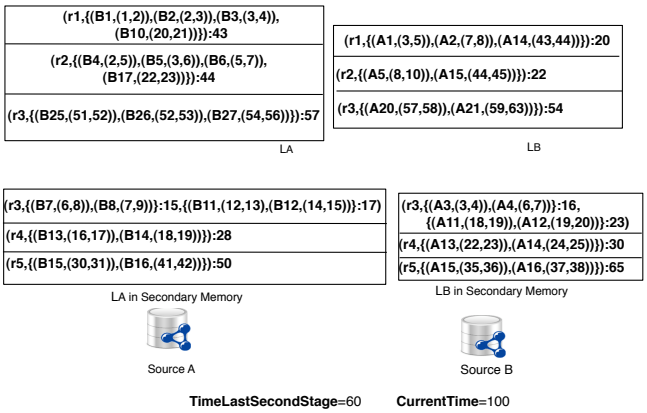


Fig. 4. The **agjoin** third stage at time 100, after having the last second stage at time 60.

is fired; the last second stage was performed at time 60. First, **agjoin** tries to combine RJT of r3 in secondary memory of source *A* with RJT of r3 in main memory of source *B*. Because A21 was inserted in the RJT at time 63, i.e., after the last second stage was performed, the combination of A21 with all the tuples of RJT of r3 in secondary memory of source *A*, must be output. The rest of the combinations between tuples in these RJTs were already produced. Then, RJT of r3 in secondary memory of *B* and RJT of r3 in main memory of source *A* are considered, and no answers are produced because all the tuples satisfy conditions in Definition 2. Next, RJTs in secondary memory are combined, but no answers are produced: (a) tuples of RJTs of r3 in secondary memory were matched in a first stage, (b) tuples of RJTs of r4, and tuples of RJTs of r5, were matched in a first stage; at this point **agjoin** finalizes.

Property 2 *Let A and B be sources joined with the* **agjoin** *operator, no duplicates are generated. Additionally, if A and B send all the tuples, the output is complete.*

4.2 The Adaptive Dependent Join (adjoin)

The **adjoin** extends the Dependent join operator [6] with the capability to hide delays to the user. The Dependent join is a non-commutative operator, that is required when instantiations of input attributes need to be bound to produce the output. Similarly, the **adjoin** is executed when a certain binding is required to execute part of a SPARQL query. For example, suppose triple pattern t_1={s p_1 ?*X*} is part of an outer sub-query, triple pattern t_2={?*X* p_2 o} is part of the inner sub-query, and the predicate p_1 is `foaf:page`, `rdfs:seeAlso`, or `owl:sameAs`. For each instantiation μ of variable ?*X*, dereferences of μ must be performed before executing the inner sub-query, i.e., the **adjoin** is used when instantiations from the outer sub-query need to be dereferenced to execute the inner sub-query. Also, the clause `BINDINGS` in SPARQL 1.1 represents this type of dependencies. We implemented the **adjoin** as an extension of the **agjoin** operator, but instead of asynchronously accessing sources *A* and *B*, accesses to source *B* are only fired when tuples from source *A* are inserted in L_B. The rest of the operator remains the same.

5 Experimental Study

We empirically analyze the performance of the proposed query processing techniques, and report on the execution time of plans comprised of ANAPSID operators versus queries posed against SPARQL endpoints, and state-of-the-art RDF engines.

Dataset	Number of triples
LinkedSensorData-blizzards	56,689,107
linkedCT	9,809,330
DBPedia	287,524,719

(a) Dataset Cardinality

Benchmark	#patterns	answer size
1	24-30	1,298-9,008
2	13-17	1-99
3	16-20	0-7

(b) Query Benchmarks

Fig. 5. Experiment Configuration Set-Up

Datasets and Query Benchmarks[13]**:** LinkedSensorData-blizzards[14], linkedCT[15], and DBPedia (english articles)[16] were used; datasets are described in Table of Figure 5(a). Sensor data [17] was accessed through a Virtuoso SPARQL endpoint; the timeout was set to 86,400 secs. We could not execute our benchmark queries against existing endpoints for clinical trials because of timeout configuration, so we implemented our own Virtuoso endpoint with timeout equal to 86,400 secs.[18] Three sets of queries were considered (Table of Figure 5(b)); each sub-query was executed as a query against its corresponding endpoint. Benchmark 1 is a set of 10 queries against LinkedSensorData-blizzards; each query can be grouped into 4 or 5 sub-queries. Benchmark 2 is a set of 10 queries over linkedCT with 3 or 4 sub-queries. Benchmark 3 is a set of 10 queries with 4 or 5 sub-queries executed against linkedCT and DBPedia endpoints.

Evaluation Metrics: we report on runtime performance, which corresponds to the *user time* produced by the `time` command of the Unix operation system. Experiments were executed on a Linux CentOS machine with an Intel Pentium Core2 Duo 3.0 GHz and 8GB RAM. Experiments in RDF-3X were run in both cold and warm caches; to run cold cache, we cleared the cache before running each query by performing the command `sh -c "sync ; echo 3 > /proc/sys/vm/drop_caches"`; to run on warm cache, we executed the same query five times by dropping the cache just before running the first iteration of the query. Each query executed by ANAPSID and SPARQL endpoints was run ten times, and we report on the average time.

Implementations: ANAPSID was implemented in Python 2.6.5.; the SPARQL Endpoint interface to Python (1.4.1)[19] was used to contact endpoints. To be able to configure delays and availability, we implemented an endpoint simulator in Python

[14] http://wiki.knoesis.org/index.php/LinkedSensorData
[15] http://linkedCT.org
[16] http://wiki.dbpedia.org/Datasets
[17] http://sonicbanana.cs.wright.edu:8890/sparql
[18] http://virtuoso.bd.cesma.usb.ve/sparql
[19] http://sparql-wrapper.sourceforge.net/

2.6.5. This simulator is comprised of servers and proxies. Seven instances of this script were run and listened on different ports, simulating seven endpoints. Servers materialize intermediate results of queries in Benchmark 2, and were implemented using the Twisted Network framework 11.0.0[20]. Proxies send data between servers and RDF engines, following a particular transfer delay and respecting a given size of messages; they were implemented using the Python low level networking socket interface.

5.1 Performance of the ANAPSID Query Engine

We compare ANAPSID performance with respect to Virtuoso SPARQL endpoints, ARQ 2.8.8. BSD-style[21], and RDF-3X 0.3.4.[22]. RDF-3X is the only engine that accessed data stored locally, so we ran queries in both cold and warm caches. Execution times in warm caches indicate a lower bound on the execution time, and correspond to a best scenario when all the datasets are locally stored and physical structures are created to efficiently access the data. Datasets linkedCT and DBPedia were merged; RDF-3X ran queries in Benchmark 3 against this dataset. Queries ran in ANAPSID were comprised of sub-queries combined using the **agjoin** and **adjoin** operators. To facilitate the execution of queries against the Virtuoso endpoints, the SPONGE parameter was set to *Local*, i.e., the endpoint only considered data locally stored in its database; the rest of the configurations of SPONGE failed, reporting the errors: `server stopped responding` and `proxy error 502`. Table 2 reports on execution times and geometric means for Benchmarks 1, 2 and 3.

We can observe that RDF-3X is able to improve cold cache execution time by a factor of 1.37 in the geometric mean when the Benchmark 1 queries were run in warm cache, by a factor of 1.8 for Benchmark 2, and by a factor of 2.85 for Benchmark 3. This is because RDF-3X exploits compressed index structures and caching techniques to efficiently execute queries in warm cache. ANAPSID accesses remote data and does not implement any caching technique or compressed index structures; however, it is able to reduce the execution time geometric means of the other RDF engines. For queries in Benchmark 1, Virtuoso SPARQL endpoint execution time is reduced by a factor of 19.31, and RDF-3X warm cache execution time is improved by a factor of 3.62; ARQ failed evaluating these queries.

Further, queries in Benchmark 2 timed out in all linkedCT SPARQL endpoints. Similarly, queries q4 to q9 timed out after 12 hours in ARQ. However, ANAPSID was able to run all the Benchmark 2 queries, and overcome RDF-3X in warm cache and ARQ by a factor of 1.1 and 4,160.56, respectively. Finally, for queries in Benchmark 3, which combine data from linkedCT and DBPedia, we observed that RDF-3X did not exhibit a good performance, while the SPARQL endpoints as well as ARQ, failed executing all the queries. Bad performance of RDF-3X may be because, the dataset result of mixing linkedCT and DBPedia has around 18GB, and this size impacts on the aggregated index structures needed to be accessed during both optimization and

[20] http://twistedmatrix.com.
[21] http://sourceforge.net/projects/jena/
[22] http://www.mpi-inf.mpg.de/ neumann/rdf3x/

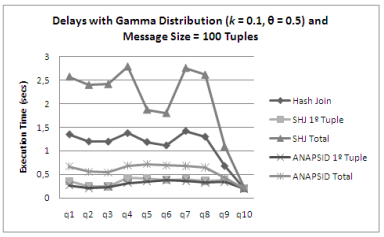
Table 2. Execution Time (secs) Different RDF Engines; Virtuoso Endpoint Sponge Local.

Benchmark 1											
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	Geom. Mean
RDF-3X	Cold Caches										
	7.83	7.12	8.47	7.45	6.36	523.89	551.20	462.77	472.42	473.20	60.60
	Warm Caches										
	4.40	4.14	4.09	4.18	4.05	466.79	465.26	464.65	475.95	463.96	44.10
SPARQL Endpoint	380.71	147.03	129.40	141.06	93.86	374.56	464.02	330.16	466.62	198.86	234.86
ANAPSID	16.60	9.22	9.54	6.80	9.59	21.48	14.34	13.48	11.08	16.19	12.16
Benchmark 2											
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	Geom. Mean
RDF-3X	Cold Caches										
	6.35	3.55	4.13	1,543.82	3.71	4.36	1,381.9	2.75	3.83	0.51	10.62
	Warm Caches										
	2.44	2.28	2.41	1,385.09	2.71	1.75	1,321.05	1.74	1.73	0.14	5.87
SPARQL Endpoint	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
ANAPSID	6.21	6.11	6.67	7.27	6.94	6.24	6.89	6.76	4.28	1.10	5.30
ARQ	21,043.34	17,686.52	18,936.85	43,200+	43,200+	43,200+	43,200+	43,200+	43,200+	593.36	22,051.01+
Benchmark 3											
	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	Geom. Mean
RDF-3X	Cold Caches										
	6.84	4.15	4.12	34,037.8	2,954.76	2,447.02	35,497.11	2,403.11	2,402.71	0.33	268.49
	Warm Caches										
	0.88	0.92	0.90	27,779.41	2,468.83	2,416.54	26,420.77	2,374.60	2,374.51	0.003	94.01
SPARQL Endpoint	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
ANAPSID	12.54	11.66	12.97	18.17	10.41	9.79	12.60	12.87	6.68	7.03	11.03

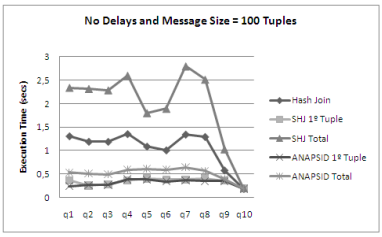
query execution. Furthermore, the endpoints were not able to execute these queries, because they could not dereference the URIs in the queries before meeting the timeout. Finally, ARQ executed all the joins as *Nested Loop joins*, and invoked many times the different endpoints, which failed executing the queries because the maximum number of allowed requests was exceeded. However, ANAPSID showed a stable behavior along all the queries, overcoming RDF-3X in warm caches by a factor of 8.52. ANAPSID performance relies on the operators and the shape of plans; they are composed of small-sized sub-queries that can be executed very fast by the endpoints. These results indicate that even in the best scenarios where data is locally stored and state-of-the-art RDF engines are used to execute the queries, ANAPSID is able to remotely access data and reduce the execution time.

5.2 Adaptivity of ANAPSID Physical Operators

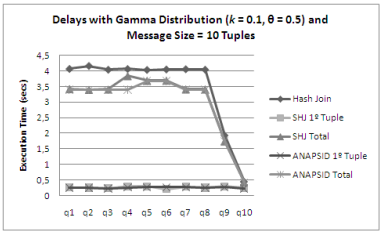
We also conducted an empirical study to analyze adaptivity features of ANAPSID operators in presence of unpredictable data transfers or data availability. We implemented an endpoint simulator, and ran different types of physical join operators to analyze the impact on the query execution time, of different data transfer distributions. We considered three join implementations: (a) Blocking corresponds to a traditional Hash join which produces all the answers at the end of the execution, (b) SHJ implements a Symmetric Hash Join, and (c) the ANAPSID **agjoin** operator. All the operators were implemented



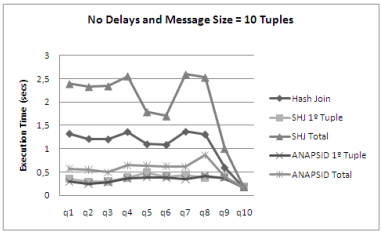
(a) Gamma(k=0.1;θ=0.5), 100 tuples.



(b) No Delays, 100 tuples.



(c) Gamma(k=0.1,θ=0.5), 10 tuples.



(d) No Delays, 10 tuples.

Fig. 6. Execution time (secs.) of Hash Join, Symmetric Hash Join (SHJ), and ANAPSID operators.

in Python 2.6.5. We measured the time to produce the first tuple, and time to completely produce the query answer. To run the simulations, queries of Benchmark 2 were executed and all intermediate results were stored in files, which were accessed by the endpoint simulator server during query execution simulations; five different simulated endpoints were executed. Data transfer rates were configured to respect a Gamma distribution with $k = 0.1$ and $\theta = 0.5$; message sizes were set to 100 and 10 tuples. Finally, the performance of all the operators in an ideal environment with no delays, was also studied.

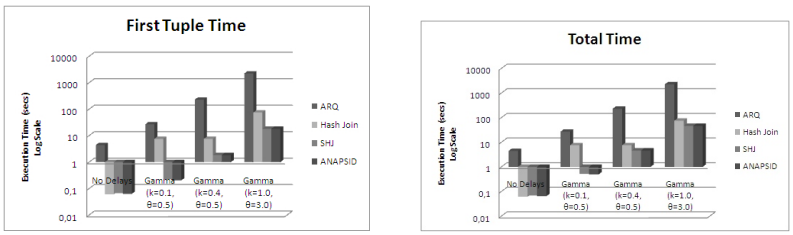
Figure 6 reports on the performance of the proposed operators. We can observe that the usage of RJTs in ANAPSID, benefits a faster generation of the first tuple as well as the output of the complete answer, even considering the cost of managing asynchronous processes in the non-blocking operators. In case that the tuple transfer delays are high (Figure 6 (c)), SHJ and ANAPSID operators exhibit a similar behavior; this is because the savings produced by using the RJTs are insignificant with respect to the time spent in receiving the data. Based on these results, we can conclude that ANAPSID operators overcome blocking operators, and that their performance may be affected by the distribution data transfer rate.

Finally, we ran ARQ, Hash join, SHJ, and ANAPSID against the endpoint simulator, and evaluated their performance in the following SPARQL 1.1. query:

```
SELECT DISTINCT ?fn3 ?fn5 ?C WHERE
{     {SERVICE <http://127.0.0.1:9000> {
          ?A4 <http://data.linkedct.org/resource/linkedct/intervention_name> "Coenzyme Q10" .
          ?A3 <http://data.linkedct.org/resource/linkedct/intervention> ?A4 .
          ?A3 <http://data.linkedct.org/resource/linkedct/condition> ?C .
          ?A3 <http://xmlns.com/foaf/0.1/page> ?fn3 .}} .
```

```
{SERVICE <http://127.0.0.1:9001> {
    ?A6 <http://data.linkedct.org/resource/linkedct/intervention_name> "Niacin" .
    ?A5 <http://data.linkedct.org/resource/linkedct/intervention> ?A6 .
    ?A5 <http://data.linkedct.org/resource/linkedct/condition> ?C .
    ?A5 <http://xmlns.com/foaf/0.1/page> ?fn5 .}} .}
```

Intermediate results to answer the query were loaded in 15 files which were accessed through two simulated endpoints. We considered three types of delay distributions as well as no delays; Figure 7 reports on execution time (secs. log-scale).



(a) Time First Tuple (b) Total Time

Fig. 7. ANAPSID Physical Operators versus state-of-the-art Join Operators. Execution time in (secs. log-scale).

We observe that SHJ and ANAPSID operators are able to produce the first tuple faster than ARQ or Hash join, even in an ideal scenario with no delays; further, ARQ performance is clearly affected by data transfer distribution and its execution time can be almost two orders of magnitude greater than the time of SHJ or ANAPSID. We notice that SHJ and ANAPSID are competitive, this is because the number of intermediate results is very small, and the the benefits of the RJTs cannot be exploited. This suggests that the performance of ANAPSID operators depend on the selectivity of the join operator and the data transfer delays.

6 Related Work

Query optimization has emphasized on searching strategies to select the best sources to answer a query. Harth et al. [7] present a Qtree-based index structure which stores data source statistics that have been collected in a pre-processing stage. A Qtree is a combination of histograms and an R-tree multidimensional structure; histograms are used for source ranking, while regions determine the best sources to answer a join query. Li and Heflin [16] build a tree structure which supports the integration of data from multiple heterogeneous sources. The tree is built in a bottom-up fashion; each triple pattern is rewritten according to the annotations on its corresponding datasets. Kaoudi et al. [13] propose a technique that runs on Atlas, a P2P system for processing RDF distributed data that are stored in hash tables. The purpose of this technique is to minimize the query execution time and the bandwidth consumed; this is done by reducing the cardinality of intermediate results. A dynamic programming algorithm was

implemented that relies on message exchange among sources. None of these approaches use information about the processing capacity of the selected sources; in consequence, they may select endpoints that will time out because the submitted query is too complex.

The XJoin [22] is a non-blocking operator based on the Symmetric Hash Join, and it follows two principles: incremental production of answers as sources become available, and continuous execution including the case when data sources present delays; access to the sources is not done through SPARQL endpoints, and the XJoin operator can only be applied when its arguments are evaluated independently. The Tukwila integration system [5] executes queries through several autonomous and heterogeneous sources. Tukwila decomposes original queries into a number of sub-queries on each source, and uses adaptive techniques to hide delays. We consider dependency between arguments and define operators able to respect binding pattern restrictions while delays are hidden.

Urhan et al. [23] present the algorithm of scrambling query plans that aims to hide delays; in case a source becomes blocked and all the previously gathered data have already been considered, the execution plan is reordered to produce at least partial results. Hartig et al. [9] rely on an adaptive iterator model that is able to detect when a dereferenced dataset stops responding, and submits other query requests to alive datasets; also, heuristics-based techniques are proposed to minimize query intermediate results [8]. Ludwig and Tran [14] propose a mixed query engine; sources are selected using aggregated indexes that keep information about triple patterns and join cardinalities for available sources; these statistics are updated on-the-fly. Execution ends when all relevant sources have been processed or a stop condition given by the user is hold; additionally, the Symmetric Hash Join is implemented to incrementally produce answers; recently, this approach was extended to also process Linked Data locally stored [15]. Avalanche [2] produces the first k results, and sources are interrogated to obtain statistics which are used to decompose queries into sub-queries that are executed based on their selectivity; sub-queries results are sent to the next most selective source until all sub-queries are executed; execution ends when a certain stop condition is reached. Finally, some RDF engines are able to process federated SPARQL queries[4, 18, 20]. Although these approaches are able to access Linked data, none of them provide an adaptive solution to query SPARQL endpoints.

7 Conclusions and Future Work

We have defined ANAPSID, an adaptive query processing engine for RDF Linked Data accessible through SPARQL endpoints. ANAPSID provides a set of physical operators and an execution engine able to adapt the query execution to the availability of the endpoints and to hide delays from users. Reported experimental results suggest that our proposed techniques reduce execution times and are able to produce answers when other engines fail. Also, depending on the selectivity of the join operator and the data transfer delays, ANAPSID operators may overcome state-of-the-art Symmetric Hash Join operators. In the future we plan to extend ANAPSID with more powerful and lightweight operators like *Eddy* and *MJoin* [5], which are able to route received responses through different operators, and adapt the execution to unpredictable delays by changing the order in which each data item is routed.

References

1. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the WWW*, pages 41–50, 2010.
2. C. Basca and A. Bernstein. Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In *The 6th International Workshop on SSWS at ISWC*, 2010.
3. E. Blanco, Y. Cardinale, and M.-E. Vidal. A sampling-based approach to identify qos for web service orchestrations. In *iiWAS*, pages 25–32, 2010.
4. C. Buil-Aranda, M. Arenas, and O. Corcho. Semantics and optimization of the sparql 1.1 federation extension. In *ESWC (2)*, pages 1–15, 2011.
5. A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
6. D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD Conference*, pages 311–322, 1999.
7. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, pages 411–420, 2010.
8. O. Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *ESWC*, pages 154–169, 2011.
9. O. Hartig, C. Bizer, and J. C. Freytag. Executing sparql queries over the web of linked data. In *ISWC*, pages 293–309, 2009.
10. S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD Conference*, pages 297–308, 2009.
11. D. Izquierdo, M.-E. Vidal, and B. Bonet. An expressive and efficient solution to the service selection problem. In *International Semantic Web Conference (1)*, pages 386–401, 2010.
12. Jena TDB. http://jena.hpl.hp.com/wiki/TDB, 2009.
13. Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. Sparql query optimization on top of dhts. In *ISWC*, pages 418–435, 2010.
14. G. Ladwig and T. Tran. Linked data query processing strategies. In *ISWC*, pages 453–469, 2010.
15. G. Ladwig and T. Tran. Sihjoin: Querying remote and local linked data. In *ESWC (1)*, pages 139–153, 2011.
16. Y. Li and J. Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *ISWC*, pages 502–517, 2010.
17. T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD International Conference on Management of Data*, pages 627–640, 2009.
18. B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *ESWC*, pages 524–538, 2008.
19. E. P. Steve Harris, Andy Seaborne. SPARQL 1.1 Query Language, June 2010.
20. M. Stoker, A. Seaborne, A. Bernstein, C. Keifer, and D. Reynolds. SPARQL Basic Graph Pattern Optimizatin Using Selectivity Estimation. In *WWW*, 2008.
21. T. Tran, L. Zhang, and R. Studer. Summary models for routing keywords to linked data sources. In *ISWC*, pages 781–797, 2010.
22. T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
23. T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD Conference*, pages 130–141, 1998.
24. M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martinez, J. Sierra, and A. Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *ESWC*, pages 228–242, 2010.
25. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
26. G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.