Optimizing Approximate Membership Metadata in Triple Pattern Fragments for Clients and Servers

Ruben Taelman, Joachim Van Herwegen, Miel Vander Sande, Ruben Verborgh

IDLab, Department of Electronics and Information Systems, Ghent University – imec

Abstract. Depending on the HTTP interface used for publishing Linked Data, the effort of evaluating a SPARQL query can be redistributed differently between clients and servers. For instance, lower server-side CPU usage can be realized at the expense of higher bandwidth consumption. Previous work has shown that complementing lightweight interfaces such as Triple Pattern Fragments (TPF) with additional metadata can positively impact the performance of clients and servers. Specifically, Approximate Membership Filters (AMFs)—data structures that are small and probabilistic—in the context of TPF were shown to reduce the number of HTTP requests, at the expense of increasing query execution times. In order to mitigate this significant drawback, we have investigated unexplored aspects of AMFs as metadata on TPF interfaces. In this article, we introduce and evaluate alternative approaches for server-side publication and client-side consumption of AMFs within TPF to achieve faster query execution, while maintaining low server-side effort. Our alternative client-side algorithm and the proposed server configurations significantly reduce both the number of HTTP requests and query execution time, with only a small increase in server load, thereby mitigating the major bottleneck of AMFs within TPF. Compared to regular TPF, average query execution is more than 2 times faster and requires only 10% of the number of HTTP requests, at the cost of at most a 10% increase in server load. These findings translate into a set of concrete guidelines for data publishers on how to configure AMF metadata on their servers.

1. Introduction

SPARQL endpoints, that expose Linked Data on the Web through a query-based interface, tend to suffer from availability issues [1]. In comparison to most other HTTP servers, SPARQL endpoints require high-end computational resources due the high complexity of SPARQL queries and can thus be difficult to sustain when a number of concurrent clients request query execution. In order to cope with this problem, the Linked Data Fragments (LDF) effort [2] has been initiated as a conceptual framework to investigate alternative query interfaces to publish Linked Datasets, by redistributing the effort of query evaluation between servers and clients.

LDF interfaces allow some parts of the query to be performed on the server, and some on the client, which leads to a redistribution of effort between server and client. This redistribution requires queries to be decomposed into multiple smaller queries, which typically leads to slower query execution due to the HTTP overhead of these roundtrips, compared to fully server-side query execution. In order to reduce this

number of smaller queries, servers could send a pre-filter to the client, which could potentially eliminate many of these queries. The focus of this work is investigating such pre-filters.
In recent years, different kinds of these LDF interfaces have been introduced, such as Triple Pattern Fragments (TPF) [2], Bindings-Restricted Triple Pattern Fragments [3], SaGe [4], and Smart-KG [5]. Each of these types of interfaces introduce their own trade-offs in terms of server and client effort. Additionally, LDF interfaces can enable feature-based extensibility, which allows servers to optionally expose certain features as metadata through usage of self-descriptive hypermedia [6], which can then be detected automatically by supporting clients to enhance the query evaluation process. Due to the extensibility of TPF, several interface features have already been proposed for TPF [7, 8, 9]. One such feature is Approximate Membership Filter (AMF) [7] metadata, which supporting clients can use to reduce the number of HTTP requests, with only a slight increase in server cost. Unfortunately, this currently comes at the cost of slower query execution, because the individual HTTP requests were larger and more expensive to compute. Since TPF is quickly gaining adoption among publishers [10], we focus on improving the performance of AMF with TPF in this work. AMF could also be useful for other types of LDF interfaces, but we consider this out of scope for this work.
Even though the work on extending TPF with AMFs showed excessive overhead, we claim that these problems can be resolved, and that AMFs can be used to lower overall query execution times without significantly increasing server load. As such, the goal of our work is to investigate what changes are required server-side and client-side to optimize AMFs for TPF. Concretely, we introduce six dimensions through which the AMF approach from Vander Sande et al. [7] can be improved. One of these dimensions involves the introduction of a new client-side algorithm to handle AMFs. The other dimensions are related to the server-side handling of AMFs. The effects and feasibility of each of these dimensions are evaluated and analyzed in detail. In summary, our work brings a deeper understanding of the appliance and benefits of AMF metadata for Linked Data interfaces, so that Linked Data publishers can expose their Linked Datasets in a more efficient manner through TPF interfaces.

2. Related Work

In this section we cover the relevant existing research relating to our work. We start by discussing the TPF interface. After that, we discuss different AMFs, followed by their use in query evaluation, and their use for the TPF interface.

2.1. Triple Pattern Fragments

Linked Data Fragments (LDF) [2] is a conceptual framework to study interfaces for publishing Linked Data, by comparing server and client effort. During query execution, some LDFs may require a low server effort, at the cost of increased client-side querying effort (*e.g. data dumps*). while others require a high server effort, at the cost of minimal client-side effort (*e.g. SPARQL endpoint*). The Triple Pattern Fragments

(TPF) interface [2] was introduced as a trade-off between those extremes, by restricting the server interface to triple pattern queries, and leaving the remainder of query evaluation to the client. Compared to SPARQL endpoints, TPF in general reduces the required server-side capacity and load for query evaluation at the expense of more bandwidth usage and slower query times. Results show that the number of HTTP requests forms the primary bottleneck during querying.
TPF follows the REST architectural style, and aims to be a fully self-descriptive API. TPF achieves this by including *metadata* and declarative *controls* in all of its RDF responses next to the main data. The metadata can contain anything that may be useful for clients during query execution, such as cardinality estimates.

2.2. Approximate Membership Functions

Approximate Membership Functions (AMFs) are probabilistic data structures that efficiently can determine membership of a set, at the cost of false positives. They are typically much smaller than a full dataset, making them a useful pre-filtering method. When selecting among different AMF techniques, we need to take into account trade-offs between filter size and false-positive rate.
Bloom filters [11] and *Golomb-coded sets* (GCS) [12] are examples of AMF techniques. Both approaches guarantee a 100% recall, but not a 100% precision. A Bloom filter is essentially a bitmap filled with the range of a predefined number of hash functions. Elements are added to the filter by applying all hash functions, and `OR`-ing the results into the bitmap. Afterwards, membership tests can be done by applying all hash functions again, and performing a bit-wise `AND` to see if all results are *possibly* present. GCS were introduced as an improvement to Bloom filters by using only a single hash function. Furthermore, the range of the hash function is always a uniformly distributed list instead of a bitmap, which allows for more efficiency compression using geometric distributions [13]. Compared to Bloom filters, GCS achieve a higher compression rate, at the cost of slower decompression.

2.3. Approximate Membership for Query Evaluation

AMFs find their use in many areas related to RDF querying, such as join optimization and source selection.
AMFs have been proven to be a useful tool for improving the performance of *graph pattern joins*. Bloom filters can therefore be used to efficiently group connected triple patterns by frequency [14], to improve the efficiency of merge joins as a way of representing equivalent classes [15], and for joining distributed and stored streams [16].
Furthermore, Bloom filters are also used in the domain of federated querying to optimize the process of *source selection*. Concretely, SPARQL's boolean ASK response can be enhanced with Bloom filters as a way of sharing a concise summary of the matching results [17]. This allows source selection algorithms to identify overlap between different sources, and can either minimize the required number of requests, or it can be used to retrieve as many results as possible.

2.4. Approximate Membership Metadata for TPF

Pure TPF query plans typically produce a large number of *membership requests* [7], checking whether a specific triple (without variables) is present in a dataset. Due to the significant number of HTTP requests that these membership requests require, these can cause unacceptably high query execution times. The authors have shown 50% of all requests are membership requests for 1 in 3 queries, which indicates that optimizing membership queries can have a positive effect on query evaluation.
In the spirit of LDF, servers can combine multiple interface features to assist supporting clients with query evaluation. An interface feature with *approximate membership metadata* for all variables in the requested *triple patterns* considerably reduced the number of membership requests to a server [7]. In order to reduce unneeded data transfer to clients that are unable to handle AMF metadata, the actual binary AMFs are included out-of-band behind a link in the metadata. Client-side query engines can detect this AMF metadata, and use it to test the membership of triples. Clients can skip many membership requests by ruling out true negatives (because of the 100% recall of AMFs), leaving only HTTP requests to distinguish false from true positives (because of the <100% precision). More details on the exact representation of this AMF metadata can be found on https://github.com/comunica/Article-SSWS2020-AMF/wiki/AMF-metadata.
The results of this work show that there is a significant decrease in HTTP requests when AMFs are used, at the cost of only a small increase in server load. However, even though the *number* of HTTP requests was lower (reduction of 33%), the *total execution time increased* for most queries, because of the long server delays when generating AMFs. In this work, we aim to solve this problem of higher execution times.

3. Problem Statement

The goal of our work is to optimize query execution over TPF interfaces using AMFs. We build upon the work from Vander Sande et al. [7], where the authors allowed the number of HTTP requests to be reduced at the cost of slower query execution. Our goal is to mitigate this major drawback, while retaining its advantages.
Vander Sande et al. introduced a number of follow-up questions that we use as a basis for defining our research questions. Concretely, in order to mitigate the earlier mentioned drawbacks, our research questions are defined as follows:

1. **Can query execution time be lowered by combining triple pattern AMFs client-side on larger Basic Graph Patterns (BGPs)?**
   Earlier work focused on using AMF metadata from triple pattern queries to test the membership of materialized triples, while there is potential for exploiting this for other types of patterns in the query as well. For instance, combining multiple AMFs at BGP-level by applying AMFs on triple patterns with shared variables.
2. **To what extent do HTTP caching and AMFs speed up query execution?**
   As Vander Sande et al. suggest that caching of AMFs reduce server delays, we investigate the impact of caching HTTP requests and/or AMFs.

3. **How does selectively enabling AMF impact server load and querying?**
Earlier work introduced AMF as a feature that was always enabled. However, some specific AMFs may be too expensive for servers to calculate on the fly. As such, it may be beneficial to only enable AMF for queries that have a result count lower than a certain threshold.
4. **How does network bandwidth impact query performance with AMFs?**
In experiments by Vander Sande et al., the HTTP bandwidth was set to a realistic 1Mbps. However, there is still an open question as to what extent different rates have an impact on the importance of AMF.
5. **How low can AMF false-positive probabilities become to still achieve decent client-side query performance?**
Based on their results, Vander Sande et al. have suggested that additional experimentation is needed with regards to lower *AMF false-positive probabilities*, as higher probabilities did not have a significant effect on query performance. Note that query correctness is never affected, but rather the number of requests to the server, since every positive match requires a request to verify whether it is a true or false positive.

An answer to these research questions will be formed using the experiments from Section 5.

4. Client-side AMF Algorithms

In this section, we explain the existing triple-based AMF algorithm, and we show where it lacks. Following that, we introduce a new client-side BGP-based AMF algorithm that solves these problems. For the reader's convenience, detailed examples of how these two algorithms work can be found on https://github.com/comunica/Article-SSWS2020-AMF/wiki/AMF-algorithm-examples. Finally, we introduce a heuristic that determines whether or not the BGP-based algorithm is beneficial to use.

4.1. Triple-based AMF Algorithm

```
function hasTriple(triple, context) {
  for position in ['subject', 'predicate', 'object']
    if !context.amf[position].contains(triple[position])
      return false;
  return super.hasTriple(triple, context);
}
```

Listing 1: Triple-based AMF algorithm by Vander Sande et al. [7] as a pre-filtering step for testing the membership of triples.

Vander Sande et al. [7] introduced an algorithm that acts as a cheap pre-processing step for *testing the membership of triples*. This algorithm was used in combination with the streaming greedy client-side TPF algorithm [2] for evaluating SPARQL

queries. Listing 1 depicts this algorithm in pseudo-code.
Concretely, every triple pattern that has all of its variables resolved to constants is run through this function right before a more expensive HTTP request would be performed. This function takes a triple and a query context containing the AMFs that were detected during the last TPF response for that pattern. It will test the AMFs for all triple components, and from the moment that a true negative is found, false will be returned. Once all checks pass, the original HTTP-based membership logic will be invoked.

4.2. BGP-based AMF Algorithm

Following the idea of the *triple-based* algorithm, we introduce an extension that applies this concept for *BGPs*. This makes it possible to use AMFs not only for testing the membership of triples, but also for using AMFs to test partially bound triple patterns that may still have variables. In theory, this should filter (true negative) bindings earlier in the query evaluation process.

```
function getBindings(triplePatterns, context) {
  for ((triplePattern, amf) in (triplePatterns, context.amfs))
    for position in ['subject', 'predicate', 'object']
      if ((!triplePattern[position].isVariable()
          && !amf[position].contains(triplePattern.subject))
        return new EmptyStream();
  return super.getBindings(triplePatterns, context);
}
```

Listing 2: BGP-based AMF algorithm as a pre-filtering step for BGP evaluation.

Listing 2 shows this algorithm in pseudo-code. Just like the triple-based algorithm, it acts as a pre-processing step when BGPs are being processed. It takes a list of triple patterns as input, and query context containing a list of corresponding AMFs that were detected during the last TPF responses for each respective pattern. The algorithm iterates over each pattern, and for each triple component that is not a variable, it will run it through its AMF. Once a true negative is found, it will immediately return an empty stream to indicate that this BGP definitely contains no results. If all checks on the other hand pass, the original BGP logic will be invoked, which will down the line invoke more expensive HTTP requests.

4.3. Heuristic for Enabling the BGP Algorithm

While our BGP-based algorithm may filter out true negative bindings sooner than the the triple-based algorithm, it may lead to larger AMFs being downloaded, possibly incurring a larger HTTP overhead. In some cases, this cost may become too high if the number of bindings that needs to be tested is low, e.g. downloading an AMF of 10MB would be too costly when only a single binding needs to be tested. To cope with these cases, we introduce a heuristic in Listing 3, that will estimate whether or not the BGP-based algorithm will be cheaper in terms of HTTP overhead compared to just executing the HTTP membership requests directly. Concretely, the heuristic checks if the size of an AMF is lower than the size of downloading TPF responses. This heuristic has been designed for fast calculation, with exactness as a lower priority. Based on measurements, we set `AMF_TRIPLE_SIZE` to 2 bytes, and `TPF_BINDING_SIZE` to 1000 bytes by default. In Section 5, we will evaluate the effects for different `TPF_BINDING_SIZE` values. In future work, more exact heuristics should be investigated that take perform live profiling of HTTP requests and delays to avoid the need of these constants.

```
function preferAmfForBgp(bindingsCount, triplePatternsCardinality)
  totalAmfsSize = triplePatternsCardinality.sum() * AMF_TRIPLE_SIZ
  joinRequestData = (bindingsCount * triplePatternsCardinality.len
      * TPF_BINDING_SIZE;
  return totalAmfsSize < joinRequestData;
}
```

Listing 3: Heuristic for checking if the BGP-based AMF algorithm should be executed, where `bindingsCount` is the number of intermediary bindings for the current BGP, and `triplePatternsCardinality` is an array of cardinality estimates for each triple pattern in the BGP. `AMF_TRIPLE_SIZE` is a parameter indicating the number of bytes required to represent a triple inside an AMF, and `TPF_BINDING_SIZE` is the size in bytes of a single TPF response.

5. Evaluation

The goal of this section is to answer the research questions from Section 3. First, we briefly discuss the implementations of our algorithm. After that, we present our experimental setup, and we present our results. All code and results results can be found on GitHub *(https://github.com/comunica/comunica-feature-amf)*.

5.1. Implementation

For implementing the client-side AMF algorithms, we make use of the JavaScript-based Comunica SPARQL querying framework [18]. Since Comunica already fully supports the TPF algorithm, we could implement our algorithms as fully standalone plugins. Our algorithms are implemented in separate Comunica modules, and will be available open-source on GitHub. Concretely, we implemented the original triple-based AMF algorithm, our new BGP-based AMF algorithm (*BGP Simple*), and a variant of this BGP-based algorithm (*BGP Combined*) that pre-fetches AMFs in parallel.
The original TPF server extension in the LDF server software by Vander Sande et al. [7] allowed both Bloom filters and GCS to be created on the fly for any triple pattern. To support our experiments, we extended this implementation with new features. This implementation is available on GitHub *(https://github.com/LinkedDataFragments/Server.js/tree/feature-handlers-amf-2)*. In order to measure the server overhead of large AMFs, we added a config option to dynamically enable AMFs for triple pat-

terns with number of matching triples below a given result count threshold. Next to that, we implemented an optional file-based cache to avoid recomputing AMFs to make pre-computation of AMFs possible.

5.2. Experimental Setup

Based on our LDF server and Comunica implementations that were discussed in Subsection 5.1, we defined five experiments, corresponding to our five research questions from Section 3. These experiments are defined and executed using Comunica Bencher *(https://github.com/comunica/comunica-bencher)*, which is a Docker-based benchmark execution framework for evaluating Linked Data Fragments. This enables reproducibility of these experiments, as they can be re-executed with a single command.
The following experiments execute WatDiv with a dataset scale of 100 and a query count of 5 for the default query templates, leading to a total of 100 queries. We only report results for Bloom filters for experiments where no significant difference was measured with GCS. Each experiment includes a warmup phase, and averages results over 3 separate runs. During this warmup phase, the server caches all generated AMFs. For each query, the client-side timeout was set to 5 minutes and, to enforce a realistic Web bandwidth, the network delay was set to 1024Kbps. All experiments were executed on a 64-bit Ubuntu 14.04 machine with 128 GB of memory and a 24-core 2.40 GHz CPU—each Docker container was limited to one CPU core, behind an NGINX HTTP cache.

1. **Client-side AMF Algorithms**: In this experiment, we compare different client-side algorithms (*None, Triple, BGP Simple, BGP Combined, Triple with BGP Combined*) for using AMF metadata.
2. **Caching**: In this experiment, we evaluate the effects of caching all HTTP requests combined with caching AMF filters server-side, both following the LRU cache replacement strategy. We also compare the effects of using AMF metadata client-side or not. Finally, we test the effects of warm and cold caches.
3. **Dynamically Enabling AMF**: In this experiment, we compare different result count thresholds (*0, 1.000, 10.000, 100.000, 1.000.000*) with each other, with either the server-side AMF filter cache enabled or not. We disable the HTTP cache and warmup phase to evaluate a cold-start.
4. **Network Bandwidths**: Different network bandwidths (*256kbps, 512kbps, 2048kbps, 4096kbps*) are tested for evaluating network speedups, and their effects or different AMF algorithms (*None, Triple, BGP Combined*) are tested.
5. **False-positive Probabilities**: In this final experiment, we compare different AMF false-positive probabilities (*1/4096, 1/1024, 1/64, 1/4, 1/2*).

5.3. Results

In this section, we present the results for each of our experiments separately. We analyzed our results statistically by comparing means using the Kruskal-Wallis test, and report on their p-values (*low values indicate non-equal means*).

Client-side AMF Algorithms

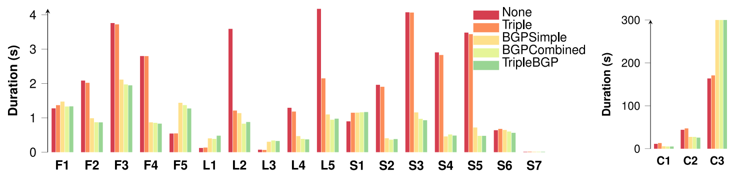

Fig. 1: Query evaluation times for the different client-side algorithms for using AMF metadata, lower is better. BGP-based approaches are mostly faster.

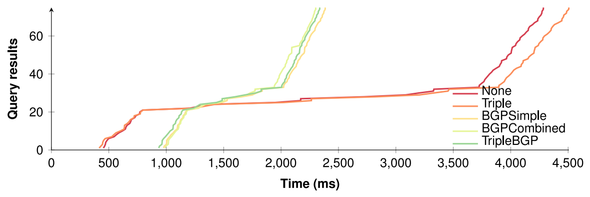

Fig. 2: Query result arrival times for query F3 for the different client-side algorithms. BGP-based algorithms introduce a delay until first result, but produce results at a higher rate after this delay.

Approach	Requests	Relative requests	Cache hits	Cache hit rate
None	1,911,845	100.00%	1,686,889	88.23%
Triple	1,837,886	96.13%	1,626,611	88.50%
BGPSimple	191,764	10.03%	173,617	90.53%
BGPCombined	191,768	10.03%	173,621	90.53%
TripleBGP	191,773	10.03%	173,626	90.53%

Fig. 3: Number of HTTP requests, number of HTTP requests relative to not using AMFs, number of cache hits and cache hit rate for the different client-side algorithms. BGP-based algorithms require significantly fewer HTTP requests.

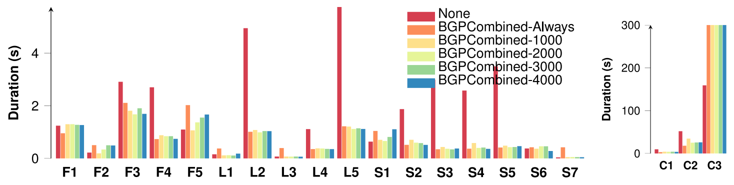

Fig. 4: Query evaluation times when enabling the heuristic in the client-side combined BGP algorithm. The heuristic shows a slight improvement in most cases.

Fig. 1 shows the query evaluation times for our first experiment on the different client-side algorithms for using AMF metadata. In line with what was shown in the first TPF AMF experiments [7], the triple-based algorithm reduces query evaluation times in only 2 of the 20 queries. Our new BGP-based algorithms on the other hand

reduce query evaluation times and outperforms the triple-based algorithm. Only for 5 of the 20 queries, evaluation times are higher or equal. Our combined BGP algorithm is slightly faster than the simple BGP algorithm. By using both the combined BGP-based and the triple-based algorithms, we can reduce evaluation times slightly further. Fig. 2 shows the query result arrival times for query F3, and is similar to the arrival times for other queries. This figure shows that the time-until-first-result is the highest for BGP-based AMF algorithms. However, once this first result comes in, the arrival rate becomes much higher compared to the other algorithms. This delay for the BGP-based algorithms is caused by the higher download times for large AMFs, and explains the higher or equal evaluation times for 5 of the 20 queries.
Fig. 3 shows the BGP-based algorithms significantly lower the number of required HTTP requests, which explains the significant reduction in query execution times. This allows the NGINX cache hit rate to slightly increase compared to the regular and triple-based TPF algorithms, since fewer requests are made, which lowers the number of required cache evictions.
Based on these results, there is *no statistically significant difference* between the evaluation times of the triple-based AMF algorithm, and not using AMF metadata at all (*p-value: 0.93*). The simple and combined BGP algorithms are significantly faster than not using AMF metadata (*p-values < 0.01*). Furthermore, the simple and combined BGP algorithm are on average more than twice as fast as the triple-based algorithm, which make them significantly faster (*p-values < 0.01*). Furthermore, combining our simple and combined BGP algorithm with the triple-based algorithms shows no further statistically significant improvement (*p-values: 0.95, 0.67*).
In Fig. 4, we show the results where we apply the heuristic for dynamically disabling the BGP heuristic based on different parameter values. On average, setting the request size parameter value to 2000 has the lowest average evaluation time for this experiment. This case achieves lower evaluation times for 19 of the 20 queries, which is an improvement compared to not using the heuristic. This improvement by itself however only small, and not statistically significant (*p-value: 0.18*).

Caching

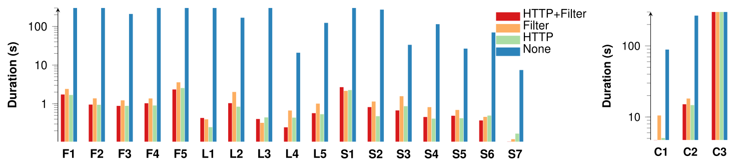

Fig. 5: Logarithmic query evaluation times comparing server-side HTTP and AMF caching. Not caching anything is always slower than caching HTTP responses or AMFs.

Fig. 5 shows that caching either HTTP requests or AMF filters server-side has a significant positive effect on query evaluation times (*p-value: < 0.01*). We observe that caching HTTP requests reduces query evaluation times *more* than just caching AMF filters (*p-value: 0.02*). Furthermore, there is no significant difference between query

evaluation times for caching of both HTTP requests and AMF filters compared to just caching HTTP requests (*p-value: 0.77*). This shows that an HTTP cache achieves the best results, and additionally caching AMF filters server-side is not worth the effort.
If we compare these results with the results for non-AMF-aware querying, we see that if HTTP caching is *disabled*, query evaluation times for non-AMF-aware querying are *significantly lower* than AMF-aware approaches (*p-value: < 0.01*). On the other hand, if HTTP caching is *enabled*, query evaluation times for non-AMF-aware querying are *significantly worse* than with AMF-aware approaches (*p-value: < 0.01*). While caching is already very important for TPF-based querying, these results show that caching becomes *even more important* when AMFs are being used.
Finally, our results show that when our cache is warm, exposing Bloom filters instead of GCS achieves faster query evaluation times. While there are a few outliers where GCS is two to three times slower, the difference is only small in most cases (*p-value: 0.18*).

Dynamically Enabling AMF

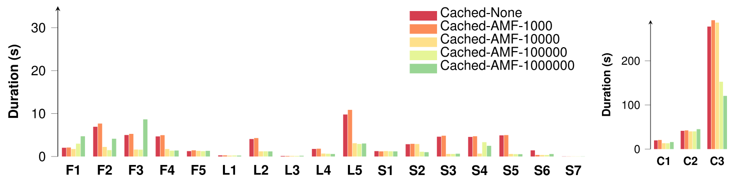

Fig. 6: Query evaluation times for different AMF result count thresholds and AMF algorithms when HTTP caching is enabled. Low result count thresholds slow down query execution.

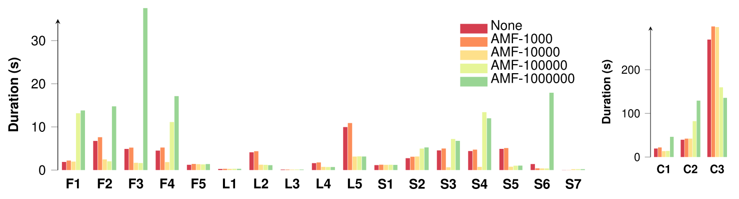

Fig. 7: Query evaluation times for different AMF result count thresholds and AMF algorithms when HTTP caching is disabled. High result count thresholds slow down query execution.

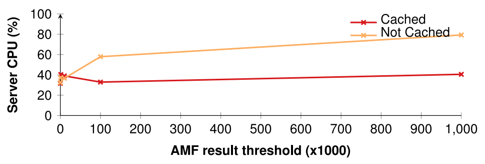

Fig. 8: Average server CPU usage increases when AMF result count thresholds increase when caching is disabled, but much slower if caching is enabled.

Fig. 6 shows lower server-side AMF result count thresholds lead to higher query evaluation times when HTTP caching is enabled (*p-value: < 0.01*). Fig. 7 shows that AMF result count thresholds also have an impact on query evaluation times when HTTP caching is disabled (*p-value: < 0.01*), but it does not necessarily lower it. For this experiment, setting the threshold to 10K leads to the lowest overall query evaluation times.
Fig. 8 shows that lower AMF result count thresholds lead to lower server loads when HTTP caching is disabled (*p-value: 0.03*). On the other hand, if HTTP caching is enabled, there is no correlation (*Pearson*) between AMF result count threshold and server CPU usage (*p-value: 0.46*). This shows that if caching is enabled, dynamically enabling AMFs based on the number of triples is not significantly important, and may therefore be disabled to always expose AMFs.
For this experiment, average CPU usage increased from 31.65% (no AMF) to 40.56% (all AMF) when caching is enabled. Furthermore, when looking at the raw HTTP logs, we observe that by *always* exposing AMFs, we use 28.66% of the total number of HTTP requests compared to not exposing AMFs. As such, AMFs significantly reduce the number of HTTP requests at the cost of ~10% more server load.
Network Bandwidth

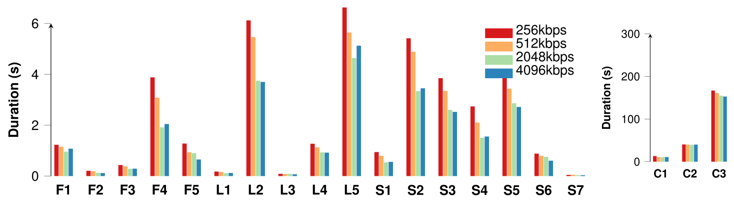

Fig. 9: When AMF is not used, query evaluation times decrease with increased bandwidth up until 2048kbps.

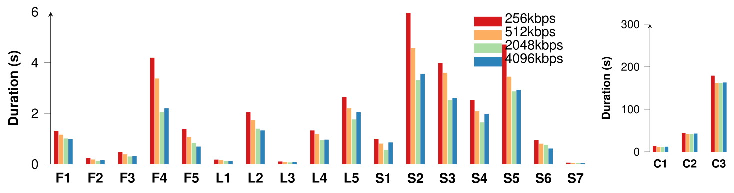

Fig. 10: When the triple-based AMF algorithm is used, query evaluation times also decrease with increased bandwidth up until 2048kbps.

Fig. 9, Fig. 10 and Fig. 11 show the effects of different bandwidths on query evaluation times over different algorithms. We observe that when not using AMF, or using the triple-level AMF algorithm, lower bandwidths lead to higher query evaluation times. However, when bandwidths become much higher, query evaluation times decrease at a lower rate. In contrast, the BGP-level AMF algorithm continuously becomes faster when bandwidth increases. We do not measure any significant impact of bandwidth on both non-AMF usage and triple-level AMF usage (*p-values: 0.29,* *0.23*). For BGP-level AMF, we measure a significant impact (*p-value: < 0.01*). This shows that *if* BGP-level AMF is used, then higher bandwidths can be exploited *more* for faster query evaluation.

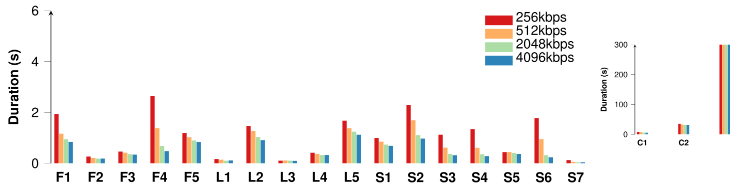

Fig. 11: When the BGP-based AMF algorithm is used, query evaluation times decrease with increased bandwidth, even for more than 2048kbps, showing that this algorithm can make better use of higher bandwidths.

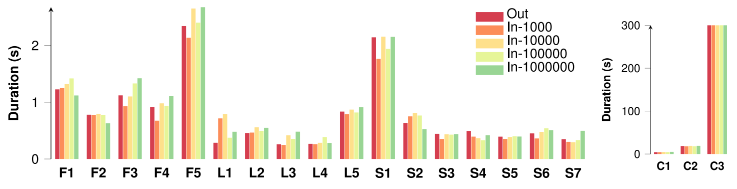

Fig. 12: Query evaluation times comparing out-of-band and in-band based on different AMF triple count threshold show no major differences.

False-positive Probabilities

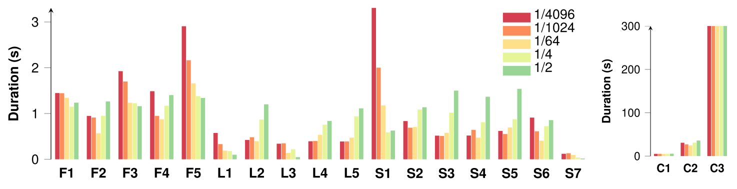

Fig. 13: Query evaluation times comparing different false-positive probabilities for AMFs that are generated server-side. Extremely low and high probabilities show a negative impact.

Fig. 13 shows that different false-positive probabilities have impact on query evaluation times. This impact has however only a weak significance (*p-value: 0.18*). On average, a false-positive probability of 1/64 leads to the lowest overall query evaluation times for this experiment.

6. Conclusions

In this article, we introduced client-side and server-side improvements to the AMF feature for TPF. The experimental results show that our client-side algorithms make average query execution more than two times faster than with regular TPF while only requiring 10% of the number of HTTP requests, at the cost of less than 10% additional server CPU usage.
We offer implementations of these algorithms and server enhancements, which means that they can be used by any of the existing data publishers that are exposing their data through a TPF interface, or any client that aims to query from them.
Hereafter, we conclude our findings with respect to our research questions, based on the evaluation, and we introduce a set of recommendations for data publishers using AMF with TPF.

6.1. Research findings

BGP-based Algorithms Improve Query Efficiency Results show that our new client-side BGP-based algorithms that use AMF metadata significantly reduce query evaluation times (*Research Question 1*). However, the are a few outliers where our new algorithms perform *worse* than the triple-based algorithm. This is because AMFs are sometimes very large, which has a significant impact on query execution times when they have to be downloaded from the server. Our results have shown that a heuristic that can decide whether or not to use the BGP-based algorithm can solve this problem, but further research is needed to come up with a more general heuristic that works in a variety of cases.
BGP-based Algorithms Postpone Time to First Results Even though total query evaluation times for the AMF-aware algorithms are mostly lower, we observe that the time-until-first-result is mostly higher. The reason for this is that the BGP-based algorithms tends to use larger AMFs, which introduces a bottleneck when requesting them over HTTP. Even though we have this overhead, the gains we get from this are typically worth it, as results come in much faster once the AMFs have been downloaded. This finding shows that dynamically switching between different algorithms may be interesting to investigate in future work. Our bandwidth experiment results confirm this bandwidth bottleneck when downloading large AMFs, and show that higher bandwidths lead to even more performance gains for the BGP-level algorithms (*Research Question 4*). This *continuous efficiency* can be investigated further in the future using metrics such as *diefficiency* [19].
Pre-computation and Caching of AMFs is Essential Our results show that AMF-aware querying only has a positive impact on query evaluation times if the server can deliver AMF filters sufficiently fast (*Research Question 2*) by for example caching them. Furthermore, if no cache is active, AMF-aware querying performs *worse* than non-AMF-aware querying. Ideally, all AMFs should be pre-computed, but due to the large number of possible triple patterns in a dataset, this is not feasible. On the other hand, our results have shown that server-side on the fly creation of AMFs only starts to have a significant impact for sizes larger than 10.000 (*Research Question 3*).

On a low-end machine (2,7 GHz Intel Core i5, 8GB RAM), creation of AMFs takes 0.0125 msec per triple, or 0.125 seconds for AMF creation of size 10.000. As such, AMFs of size 10.000 or less can be created on the fly with acceptable durations for Web servers (after which they can still be cached).
Fig. 14 shows that there is only a very small number of triple patterns with a very large number of matches. When setting the WatDiv dataset to a size of 10M triples, there are only 90 triple patterns with a size larger than 10.000. Setting that size to 100M triples, this number increases to 255, so this is not a linear increase. Due to this low number of very large patterns, servers can easily pre-compute these offline before dataset publication time. Since the WatDiv dataset achieves a high diversity of *structuredness*, it is similar to real-world RDF datasets [20]. As such, this behavior can be generalized to other datasets with a similar structuredness.

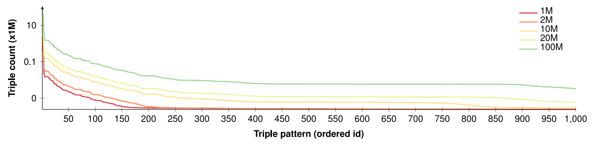

Fig. 14: Logarithmic plot of the number of matches for triple patterns in five datasets of varying sizes, limited to the 1000 patterns with the most matches. Triple patterns are sorted by decreasing number of matches.

Bloom Filters are Preferred over GCS with Active Cache Results show that when AMFs are pre-computed, Bloom filters achieve faster query evaluation times than GCS (*Research Question 2*). This is because Bloom filter creation requires less effort client-side than GCS due to the simpler decompression, at the cost of more server effort. However, this higher server effort is negligible if AMFs can be pre-computed. As such, we recommend Bloom filters to always be preferred over GCS, unless AMFs can not be cached.
A Good Trade-off Between False-positive Probabilities and AMF Size Lowering the false-positive probability of an AMF increases its size. As we have seen that larger AMFs have an impact on query evaluation times, we do not want AMFs to become too large. On the other hand, we do not want the false-positive probabilities to become too high, as that leads to more unneeded HTTP requests. Our results have shown that a probability of 1/64 leads to an optimal balance for our experiments (*Research Question 5*). However, further research is needed to investigate this trade-off for other types of datasets and queries.

6.2. Recommendations for Publishers

Based on the conclusions of our experimental results, we derived the following guidelines for publishers who aim to use the AMF feature:
- Enable **HTTP caching** with a tool such as NGINX.
- **Pre-compute AMFs** (or at least cache) AMFs of size 10.000 or higher.
- If AMFs can be cached, prefer **Bloom filters** over GCS.
- Use a false-positive **probability of 1/64**.

References

1. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.-Y.: SPARQL Web-Querying Infrastructure: Ready for Action? In: The Semantic Web–ISWC 2013. pp. 277–293 (2013).
2. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics. (2016).
3. Hartig, O., Buil-Aranda, C.: Bindings-Restricted Triple Pattern Fragments. In: Proceedings of the 15th International Conference on Ontologies, DataBases, and Applications of Semantics. pp. 762–779 (2016).
4. Minier, T., Skaf-Molli, H., Molli, P.: SaGe: Web preemption for public SPARQL query services. In: The World Wide Web Conference. pp. 1268–1278 (2019).
5. Amr, A., Fernandez Garcia, J.D., Maribel, A., Polleres, A.: SMART-KG: Hybrid Shipping for SPARQL Querying on the Web. (2020).
6. Verborgh, R., Dumontier, M.: A Web API ecosystem through feature-based reuse. Internet Computing. 22, 29–37 (2018).
7. Vander Sande, M., Verborgh, R., Van Herwegen, J., Mannens, E., Van de Walle, R.: Opportunistic Linked Data querying through approximate membership metadata. In: International Semantic Web Conference. pp. 92–110. Springer (2015).
8. Van Herwegen, J., Verborgh, R., Mannens, E., Van de Walle, R.: Query Execution Optimization for Clients of Triple Pattern Fragments. In: The Semantic Web. Latest Advances and New Domains (2015).
9. Taelman, R., Vander Sande, M., Verborgh, R., Mannens, E.: Versioned Triple Pattern Fragments: A Low-cost Linked Data Interface Feature for Web Archives. In: Proceedings of the 3rd Workshop on Managing the Evolution and Preservation of the Data Web (2017).
10. Verborgh, R.: DBpedia's triple pattern fragments: usage patterns and insights. In: European Semantic Web Conference. pp. 431–442. Springer (2015).
11. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM. 13, 422–426 (1970).
12. Putze, F., Sanders, P., Singler, J.: Cache-, hash-, and space-efficient bloom filters. Journal of Experimental Algorithmics (JEA). 14, 4 (2009).
13. Gallager, R., Van Voorhis, D.: Optimal source codes for geometrically distributed integer alphabets (corresp.). IEEE Transactions on Information theory. (1975).
14. Huang, H., Liu, C.: Estimating selectivity for joined RDF triple patterns. In: Proceedings of the 20th ACM international conference on Information and knowledge management. pp. 1435–1444. ACM (2011).
15. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. pp. 627–640. ACM (2009).
16. Dia, A.F., Aoul, Z.K., Boly, A., Métais, E.: Fast SPARQL join processing between distributed streams and stored RDF graphs using bloom filters. In: 12th International Conference on Research Challenges in Information Science (2018).
17. Hose, K., Schenkel, R.: Towards benefit-based RDF source selection for SPARQL queries. In: Proceedings of the 4th International Workshop on Semantic Web Information Management. p. 2. ACM (2012).
18. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a Modular SPARQL Query Engine for the Web. In: Proceedings of the 17th International Semantic Web Conference (2018).
19. Acosta, M., Vidal, M.-E., Sure-Vetter, Y.: Diefficiency metrics: measuring the continuous efficiency of query processing approaches. In: International Semantic Web Conference. pp. 3–19. Springer (2017).
20. Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data