Cacheable autocompletion on the Web

Ruben Dedecker, Pieter Colpaert, Ruben Verborgh

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

Abstract. Forms with an autocompletion feature are typically driven by Web APIs that filter a collection for every given prefix. Given the wide variety of possible prefixes, the resulting highly individualized requests hinder caching, which could otherwise reduce server costs. In order to enable efficient caching, we have investigated file-based fragmentation strategies for Linked Open Datasets optimized for prefix search. We extended the TREE hypermedia API specification with string-specific relations and implemented a Web API inspired by a B-tree. We evaluated this API based on five characteristics: number of requests needed, performance, cache hit ratio, bandwidth consumed and efficiency. Despite the fact the number of HTTP requests follows the scalability of the depth of the specific tree, still the number of requests for a dataset with 73k entries is acceptable. Over this dataset, and for subsequent queries, 15 results per prefix are in within 150ms, while the cache hit ratio is three times better than a query server. With a higher bandwidth (max 25kb per series for our dataset) comes a lower efficiency of ~50%. At the cost of a higher bandwidth consumption, we have shown a file-based search tree Web API can still guarantee the query performance required for autocompletion while shipping more data than strictly necessary. Also non-measurable characteristics can be thought of, such as a better privacy by design or the ability to federate queries over multiple interfaces.

HTML version at http://pieter.pm/icwe2020-autocompletion

1. Introduction

Autocompletion is a user interface feature to select an item from a long list of entities. Given a certain prefix, suggestions are provided that match the prefix to an item in the collection. Examples of autocompletion tasks on the Web include completing addresses, filling out points of interests such as public transport stops, or finding Wikidata entities. A well-established design for autocompletion services is an API that filters a full collection of items on a prefix, such as `https://example.org/{?q}`. However, for every character typed, this will require the server to process a new prefix query. Given the wide variety of possible prefixes, caching becomes hardly possible, and as a result, hosting such an autocompletion service comes at a cost. In this paper we study whether the server costs can be lowered by raising the cacheability.

The idea of Linked Data Fragments shows there are more alternative design options than to ship a datadump of the entire collection to the client. Instead, a fragmentation strategy can be thought of, where the client takes part in the query evaluation. The cacheability will raise, as fragments can be shared among clients and even reused on the same client for the similar queries. This comes with interesting opportunities to

scale up, such as the possibility to use Content Delivery Networks. The first contribution of the paper is that we implemented a public Web API exposing a collection of items in a B-tree structure. In order to evaluate this approach, we used the database of all transport stops in Belgium, for which we also published a real query-set based on an access log. We evaluate the amount of requests needed, the performance, cache hit-rate, bandwidth consumed and efficiency. A secondary contribution is that we extended an in-band hypermedia specification, called the TREE Ontology, for enabling clients to prune their search-space based on a prefix.

2. State of the art

The idea of fragmenting datasets on the Web instead of answering the full query on the server-side is not new. Work on different fragmentation strategies have slowly been advancing since 2014. While it started with Triple Pattern Fragments (TPF) [1], different extensions of the simple graph-based fragmentation have been proposed: speeding up basic graph patterns on top of TPF with approximate membership metadata [2], binding restricted TPF [3], Linked Connections for public tranport routing [4] or extending the server with time-based querying for archives [5]. So far, no fragmentation strategies have been thought of for string search.

In a survey on Query Auto Completion (QAC) [6], the state of the art is introduced. It sketches an elaborate overview of the research trends, from heuristic and learning based approaches to raise the relevance of the suggestions, to the computational complexy (on one machine) or how to present results to a user. No alternate Web API designs are discussed, where clients could take part in the query execution. Furthermore, in order to test the computational complexity, only the complexity of resolving one prefix is considered. Nevertheless, a following QAC query could continue on a previous query, and thus have a lower amortized complexity.

Open source software such as Elastic Search and triple stores such as Virtuoso (`bif:contains`) or Blazegraph (`bds:search`), offer full server-side full-text search querying. This can be used for autocompletion (e.g., as is the case in Linked Open Vocabularies [7]). Specialized autocompletion software stacks can also be found, such as Pelias, which extends Elastic Search for address autocompletion and geocoding. Particularly interesting is that Pelias also publishes their own user experience guidelines for autocompletion. These include: throttling requests, taking into account possible out of order responses and using a pre-written client on the front-end if possible. A limitation that is not discussed is the fact that due to the stateless nature of HTTP, the search algorithm cannot be paused and continued on further user input. Instead, for every new request, the whole query execution needs to be re-evaluated.

The only work we found so far that tried to extend a hypermedia API with text search is by Van Herwegen et al [8]. The TPF interface was extended with substring filtering on objects. Nevertheless, while the knowledge graph itself is fragmented in pages, a server-side feature was added for querying the data in the same way as the set-up with Elastic Search. The hypermedia controls necessary to discover this feature was not developed and the feature was not adopted in the querying SDK Comunica [9].

3. An in-band documented search space

In order to enable clients to prune their search space based on hypermedia descriptions beyond simple pagination, we need to come up with a comprehensible specification, as well as describe the client-side algorithm. In contrast to the state of the art, we want the specification to describe precisely how the index works, so any client-side query algorithm can benefit without a human developer having to explain how to interpret the responses.
We have introduced the TREE hypermedia specification depicted in Fig. 1. Using the ontology, relations can be described, linking other fragments to the current fragment. The description allows for a client to understand that, when following the link, all members of the collection from that page on will respect a certain condition, such as the fact that the name starts with a certain prefix. The TREE specification uses the triple-based RDF as its knowledge representation model, enabling different serialization to be used, as well as compatibility with existing Linked Data domain models. The full specification can be found in the TREE specification.

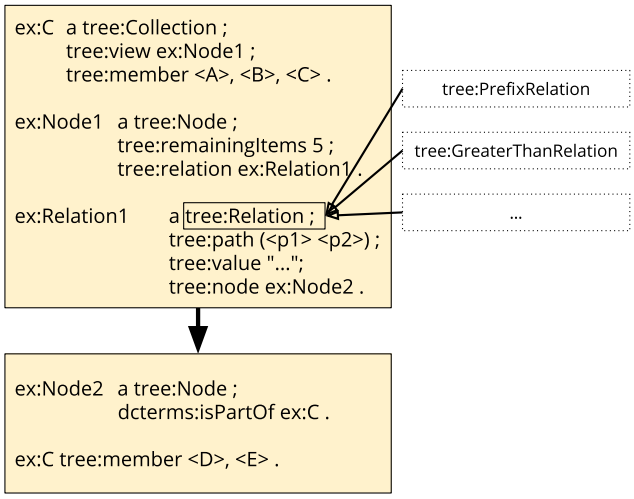

Fig. 1: TREE is a specification to qualify the relation between two pages. It is able to describe to a client whether following a certain page will not be interesting for its current query, and thus whether the client's search space can be pruned.

A `tree:Node` is linked to a `tree:Collection`, which can be found from discovery results from W3C specifications such as DCAT, Activity Streams 2.0, Web of Things, Hydra, SPARQL resultsets and Triple Pattern Fragments. Only the specific predicate `tree:view` guarantees that from this node on, all members of the collection can be found. With the predicate `tree:member`, elements of this collection can be found in the current page. The `tree:Node` describing the current page is linked with the predicate `tree:relation` to other pages. The object is a subclass of `tree:Relation`. For string search, we have defined the type

`tree:PrefixRelation`. A `tree:value` predicate links a specific string to this relation: the string is a prefix of all further members in the collection. The `tree:path` describes the property path (using the Shapes Constraints Language (SHACL) Property Paths specification) on which this `tree:value` applies. By default, unicode order is followed, yet specific flags can change the order for options such as case-sensitivity, as documented in the TREE specification.
Based on the hypermedia specification, **a server** can choose in which way to build up a prefix search space. Both dynamic and static API designs are possible. Making the case that even for large datasets server-effort can be limited, we chose to implement this using static files. The code to create your own fragmentation using a B-tree approach approach can be found on a git repository. The resulting files can be hosted on any file host, yet Cross Origin Resource Sharing (CORS) headers as well as caching headers need to be configured. Our set-up contains uses an nginx reverse proxy cache which enables the necessary headers, compression and CORS headers. When updates in the dataset happen, they can often be done locally in the node storing the data, and only that node's HTTP cache needs to be invalidated. In the case when balance needs to be restored, we refer to the state of the art in search tree design to keep the changes as local as possible.
The **client search algorithm** starts with a URL to a root node that is requested on page load. For this node, and every page that is processed, the relations must be extracted as well as the members itself. From the moment the user types a prefix, two things happen: (i) the list of members is filtered and returned; (ii) the list of relations is filtered and processed in a depth first approach. Depth first is chosen to be able to show the first suggestions as fast as possible. However, when completeness is important during a full-text search, the same API can cater a client doing a breadth first search. The code is available on this git repository. An online demo can be tested, applied to the Flemish address database.

4. Experiments and results

The *query server* is a server handling the full prefix such as in this URL template: `https://example.org/{?query}`. The *tree approach* is the approach introduced in this paper, where a client traverses a search tree for the right prefixes. In order to understand the effectiveness of the tree approach, we want to understand the implications on the amount of HTTP requests necessary, the cacheability of these requests, and how this impacts efficiency and bandwidth, based on the size of a dataset in a real-world environment. Based on this cacheability and the scalability of the depth of the tree, we can deduct what this means for the user-perceived performance.
We define an autocompletion query as the full set of HTTP requests needed to find a target. For the experiments, a real-world query log was used (available online) that we extracted from a server autocompleting Belgian railway stops at `https://irail.be/stations/NMBS/{?q}`. We then also republished a B-tree of Belgian public transport stops (6 triples per entity, and a total of 72 967 entites) with a

page size (m) of 25 members and/or relations. An extracted example `“L” → “Lo” → “Lou” → “Louv” → “Louvai” → “Louvain”` needed 6 HTTP requests in the query server approach to find the target station of Louvain.

4.1. Number of requests

The best-case amount of HTTP requests needed to autocomplete a prefix is 1, as the target may be contained in the root node. When we even consider the fact that the root node is going to be used by *any* autocompletion task, it should be requested on page load, and not during the autocompletion task itself. Therefore, the best-case amortized number of HTTP requests needed to autocomplete a prefix is 0.
In contrast to a query server, where the number of HTTP requests stays the same when a dataset grows in size, the amount of HTTP requests needed with the tree approach is proportional to the amount of elements in the dataset. In case of a B-tree, the number of requests necessary to find elements matching a prefix value is decided based on the height of the tree, which can be calculated theoretically for a dataset of n elements as: $r_{max} = \lfloor log_{\lceil M/2 \rceil}(\frac{n+1}{2}) \rfloor$, with r_{max} the maximum number of requests necessary, without counting the root node which can be prefetched, and with M the maximum number of children a node can have. The depth thus defines the number of HTTP requests needed when only the root node of a tree is found, and this is the first time the auto-completion is being ran, so the client cache is cold. With every depth that is added, it takes an exponential (power of $M/2$) number of elements more to result in an increase in worst-case number of requests necessary. Applied to our dataset: with 25 elements per page for a total of 10^5 elements, a depth of 4 would be needed (h_{max}), and thus 4 HTTP requests would be needed in worst-case to perform an autocompletion. When a new query shortly thereafter is done, the probability of being able to cache one of the higher-level nodes should become higher, which is illustrated by Fig. 3. Our implementation confirms this theoretical analysis. For a full series of queries, the experimental results are depicted in Fig. 4.

4.2. Cache efficiency

Considering a static dataset in an interval, we can statically generate the nodes needed and push them to a server-side cache. In this case, it is straightforward to achieve a *server-side cache hit ratio of 100%* as no fragments are served by a back-end application, they can immediately be served by a Content Delivery Network (CDN). In Fig. 2, we show the cache hit ratio when the nginx cache is 10% the size of the dataset. We tested this for both a query server (baseline) as the TREE approach and notice the TREE approach achieves a three times bigger cache hit rate.

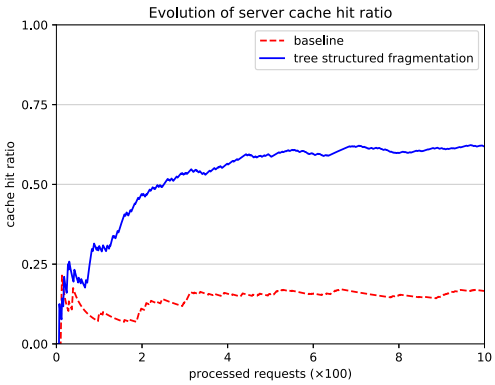

Fig. 2: Average cache hit ratio for applications evaluating series of prefix queries.

4.3. Query performance

As the TREE approach provides incremental results, we want to see how many results we can achieve within 150ms, a timespan which feels instantaneous to end-users. In Fig. 3 we can see only when typing a first charactar, the approach stays below 10 results. For any following character typed, it becomes faster to produce the next results and 15 results can be produced within 150ms. For an average query, it will also already show the first results before 50ms.

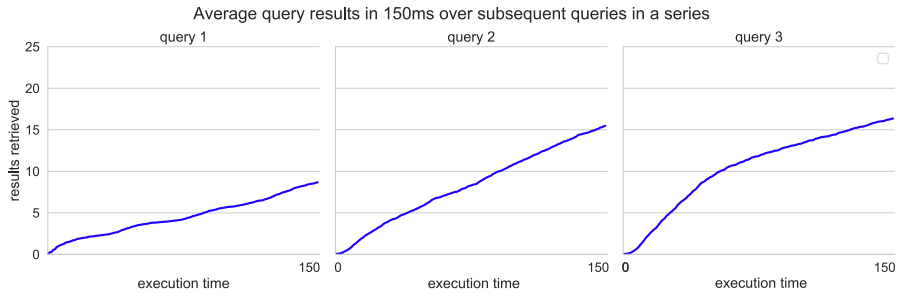

Fig. 3: Increased performance for subsequent prefix queries evaluated over the same data collection.

4.4. Efficiency and bandwidth

We define the efficiency as “the fraction of data retrieved from the server during the execution of a task over the amount of data required to execute that task”. [1]. In a query server approach, developers will aim towards a 100% efficiency. At the cost of efficiency, the TREE approach raises the cacheability. In Fig. 4, we discuss the results for how much efficiency we sacrifice and bandwidth the TREE approach consumes.

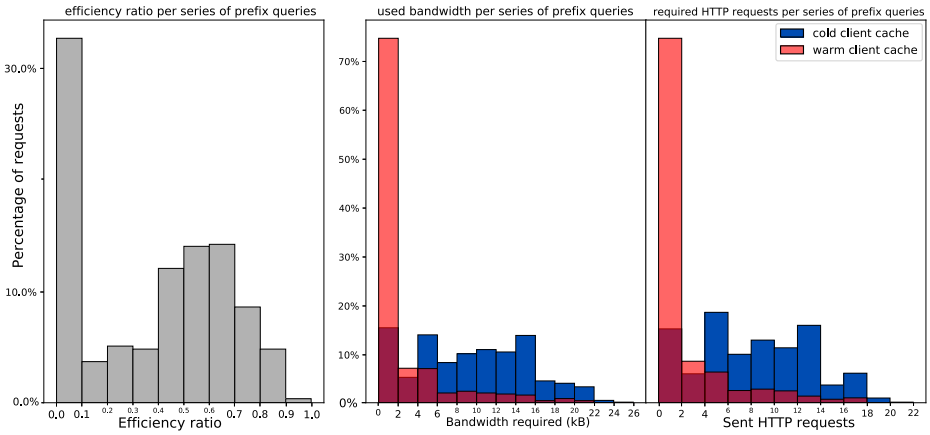

Fig. 4: The efficiency shows a large quantity of 0% queries. This is due to targets that do not result in an answer (not in the collection). For other requests we see the efficiency averages over 50%. A max bandwidth of 25kB is consumed for an autocompletion query for this particular dataset and query log. The number of requests for a full query autocompletion range from 0 to 20 requests in worst-case.

5. Conclusion

A new trade-off has been established, balancing server and client effort, to offer autocompletion over a set of items. At the expense of the client having to take part in the query evaluation and consume more bandwidth, the server is off-loaded and can work fully from cache, archive or CDN. When using a cache that is 10% the size of the dataset, the TREE approach has a cache hit ratio that is ~3 times better, which means we achieved a server that is offloaded and becomes more cost-efficient. We however do not compromise on user-perceived performance. The new client-server relation for prefix search has an effect on the user experience guidelines of Pelias (see Section 2). (i) Throttling requests became unnecessary. When altering the prefix for which no extra HTTP request would be needed, the throttling can be very low. In this research we have tried to stay as close as possible as the state of the art, as judging from its adoption, this performance is widely accepted. In a similar way, there is also no danger of out of order responses. The client-side algorithm always interrupts the current prefix evaluation to emit responses, but forwards its on-going query processing to the next prefix. Last but not least, (iii) using a pre-written client was a guideline when working with the query server design, and remains. Now however, this pre-written client is given more responsibility, and with more responsibility also comes more flexibility to implement the autocompletion or any full-text search feature in just the way you want.

Yet, also other guidelines can be thought of based on these results for server administrators. (i) The most important of them all will be setting caching headers. Both conditional caching with `etag header`, as setting a `cache-control` is of utmost importance. Do not try to reproduce this research without enabling caching: it

is the driver behind the scalability of this approach. Next, for public datasets, also (ii) Cross Origin Resource Sharing (CORS) headers need to be enabled. This will enable application developers to create *serverless* JavaScript applications. While we tested our approach on HTTP/1.1 being the current state of the art protocol, the advent of (iii) HTTP2 and HTTP3 can only work to our advantage, being faster, allowing more parallel requests, and allowing HTTP Push. Some of these features can already be enabled in existing HTTP servers. Upcoming technologies such as 5G, Web Workers, faster RDF parsers, and HTTP/3 should make the trade-off of higher bandwidth consumption in return for cheaper servers easier to accept. Moreover, as a side-effect of a coarser fragmentation strategy, the exact search string of your user is not exposed to a third party server.

References

1. 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. 37–38, 184–206 (2016).
2. Vander Sande, M., Verborgh, R., Van Herwegen, J., Mannens, E., Van de Walle, R.: Opportunistic Linked Data querying through approximate membership metadata. In: ISWC. pp. 92–110. Springer (2015).
3. Hartig, O., Buil-Aranda, C.: Bindings-Restricted Triple Pattern Fragments. In: Debruyne, C., Panetto, H., Meersman, R., Dillon, T., Kühn, eva, O'Sullivan, D., and Ardagna, C.A. (eds.) Proceedings of the 15th International Conference on Ontologies, DataBases, and Applications of Semantics. pp. 762–779. Springer (2016).
4. Colpaert, P., Verborgh, R., Mannens, E.: Public transit route planning through lightweight linked data interfaces. In: ICWE. pp. 403–411. Springer (2017).
5. Vander Sande, M., Verborgh, R., Hochstenbach, P., Van de Sompel, H.: Toward sustainable publishing and querying of distributed Linked Data archives. Journal of Documentation. 74, 195–222 (2018).
6. Cai, F., De Rijke, M., others: A survey of query auto completion in information retrieval. Foundations and Trends in Information Retrieval. 10, 273–363 (2016).
7. Vandenbussche, P.-Y., Atemezing, G.A., Poveda-Villalón, M., Vatant, B.: Linked Open Vocabularies (LOV): a gateway to reusable semantic vocabularies on the Web. Semantic Web. 8, 437–452 (2017).
8. Van Herwegen, J., De Vocht, L., Verborgh, R., Mannens, E., Van de Walle, R.: Substring Filtering for Low-Cost Linked Data Interfaces. In: Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., and Staab, S. (eds.) Proceedings of the 14th ISWC. pp. 128–143. Springer (2015).
9. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular SPARQL query engine for the web. In: ISWC. pp. 239–255. Springer (2018).