A File-based Linked Data Fragments Approach To Prefix Search

Ruben Dedecker[1 (✉) [0000−0002−3257−3394]], Harm Delva[1 [0000−0001−8272−0754]], Pieter Colpaert[1 [0000−0001−6917−2167]], and Ruben Verborgh[1 [0000−0002−8596−222X]]

[1]IDLab, Department of Electronics and Information Systems,
Ghent University–imec, Technologiepark-Zwijnaarde 122, 9052 Ghent, Belgium
Ruben.Dedecker@Ugent.be

Abstract. Text-fields that need to look up specific entities in a dataset can be equipped with autocompletion functionality. When a dataset becomes too large to be embedded in the page, setting up a full-text search API is not the only alternative. Alternate API designs that balance different trade-offs such as archivability, cacheability and privacy, may not require setting up a new back-end architecture. In this paper, we propose to perform prefix search over a fragmentation of the dataset, enabling the client to take part in the query execution by navigating through the fragmented dataset. Our proposal consists of (i) a self-describing fragmentation strategy, (ii) a client search algorithm, and (iii) an evaluation of the proposed solution, based on a small dataset of 73k entities and a large dataset of 3.87m entities. We found that the server cache hit ratio is three times higher compared to a server-side prefix search API, at the cost of a higher bandwidth consumption. Nevertheless, an acceptable user-perceived performance has been measured: assuming 150 ms as an acceptable waiting time between keystrokes, this approach allows 15 entities per prefix to be retrieved in this interval. We conclude that an alternate set of trade-offs has been established for specific prefix search use cases: having added more choice to the spectrum of Web APIs for autocompletion, a file-based approach enables more datasets to afford prefix search.

Keywords: Prefix Search · Query Evaluation · Linked Data Fragments · Web APIs

1 Introduction

Prefix autocompletion is a common user interface feature in forms. Given a certain prefix, suggestions are provided that match the prefix to an item in a collection. This way, a user can recognize the item they are looking for, rather than recalling the exact identifier it may have in the underlying data. To provide such functionality, an API can be provided that filters the dataset entities on the server-side illustrated by the URL template `https://example.org/{?query}`. Such a service requires the server to process all queries for every typed character

of every connected client. Another solution is to ship the collection of entities to the client for processing. While feasible for small collections, this quickly becomes problematic when the dataset grows.

Where some data publishers manage to publicly provide such an API for prefix autocompletion for their datasets, others leave this feature up to third parties reusing the data. An in-between solution could add more choice to the spectrum for cases where prefix query evaluation entirely on the server is less desirable. For example,

- for a website builder, shipping a collection of a couple of thousand entities would make a page too heavy, yet setting up a website with a full-text search API requires the maintenance of dynamic server-side functionality;
- for specialized cases where additional information can be incorporated in the autocompletion client, e.g. adding error correction using a list of common mistakes or filtering of geographically irrelevant results.
- for users that do not want to leak their search queries via query logs.

In this paper, we present **a file-based architecture with an acceptable user-perceived performance that enables clients to take control of the prefix query evaluation process**. The contributions are as follows: (i) a hypermedia specification that can describe fragmentation strategies for string search, (ii) a tailored implementation of a B-tree fragmentation described with this hypermedia, (iii) a client search algorithm able to traverse the hypermedia search space, and (iv) an evaluation discussing query performance, cache hit ratio, bandwidth and efficiency.

In Section 2 we provide an overview on related work that inspired our work. Evaluating the approach introduced in Section 3, in Section 4 we used the database of all public transport stops in Belgium, for which we also published a real query-set based on an access log, as well as a subset of OSMNames[1], for which we generated a random query-set. We measure whether clients evaluating prefix queries over the proposed fragmentation strategy experience an acceptable user-perceived performance by analyzing the performance, cache hit ratio, bandwidth consumed and efficiency.

2 Related Work

Research on full-text search, prefix search or autocompletion on one machine has a large history [1]. These techniques have profited from that prior work, resulting in powerful open-source tools such as ElasticSearch. Today, for example, ElasticSearch is the engine behind the autocompletion of Linked Open Vocabularies [7], offering a search engine through all indexed Linked Data vocabularies[2]. Another

[1] `https://osmnames.org/download/`

[2] The service can be used via the URL template `https://lov.linkeddata.es/dataset/lov/api/v2/term/autocompleteLabels{?q}`

example of a reconciliation tool using ElasticSearch is Pelias[3]. It offers worldwide address autocompletion and geocoding by combining different datasets such as Geonames[4], OpenStreetMap[5], Whos on first[6] and openaddresses.io. There is however no public instance and a user is required to self-host it, or rely on software as a service solutions that come at a pay per use cost. Furthermore, when using the API, there are user experience guidelines to take into account, such as (i) throttling requests, (ii) taking into account possible out of order responses and (iii) using a pre-written client on the front-end if possible.

Triple Pattern Fragments (TPF) [8] is a Linked Data API specification for solving queries using Basic Graph Patterns, introduced as an alternative to hosting a SPARQL endpoint. Instead of answering a full SPARQL query on the server-side, it requires the client to take part in the query execution. The client retrieves the fragments of the dataset required to evaluate the query from the server by requesting Triple Patterns, and evaluates the query over the retrieved fragments. Approximate counts of the specific triple patterns in the full dataset are provided in the retrieved fragments to optimize client query evaluation based on selectivity of certain triple patterns.

Van Herwegen et al [6] extended the TPF interface with substring filtering on objects using different indexes, such as ElasticSearch or an FM-index. For this part of the query, the client thus relies on the server to fully filter the triple pattern fragments response and does not explore in-between solutions. These initiatives follow the idea of Linked Data Fragments (LDF)[7] [5].

Finally, in a survey on Query Auto Completion (QAC) [1] the state of the art is discussed. It sketches an elaborate overview of the research trends, among others, heuristic and learning based approaches to raising the relevance of the suggestions, analysis of the computational complexity – yet only on one machine – of different algorithms, or the state of the art in QAC user experience. 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, despite the fact that a consecutive QAC query may continue querying from where a previous query left off, and thus have a lower amortized complexity.

3 Dataset fragmentation and traversal

Client participation in query evaluation can be easily achieved by sending all data to the client. However, as datasets grow larger, this approach leads to increased bandwidth requirement and application response times, which is undesirable for cases such as mobile applications where bandwidth caps are in place, and stable network reception cannot be guaranteed.

[3] `https://pelias.io/`
[4] `https://www.geonames.org/`
[5] `https://www.openstreetmap.org/`
[6] `https://whosonfirst.org/`
[7] `https://linkeddatafragments.org`

In this section, we propose our strategy to publish datasets by fragmenting the data using search tree structures. With this approach to data publishing, clients are enabled to evaluate prefix queries over remote datasets by only retrieving fragments of the dataset relevant to the client query. In Section 3.1 we introduce preliminaries, which we use in Section 3.2 to introduce a self-describing fragmentation strategy. Finally, in Section 3.3 a generic client-side traversal algorithm is introduced.

3.1 Preliminaries

Given a dataset D, an autocompletion interface provides autocompletion functionality over all entities in D. These entities can have different properties over which autocompletion can be offered, such as a person entity having a first and last name property. To enable fast prefix search lookups in a dataset for a given property, an index can be constructed for that property using a data structure that enables lookups to only retrieve the parts of the dataset relevant to the query. Clients use such an indexing structure to more efficiently find entities in the dataset matching a given prefix value for the indexed properties. Our approach explores generating such indexing data structures, and using them to fragment the dataset into smaller files (fragments). By embedding this tree structure as hypermedia controls in the generated fragments, clients are enabled to only retrieve fragments relevant to the evaluated prefix query from the dataset.

3.2 A self-describing fragmentation strategy

Instead of publishing a dataset as a query interface, or publishing it as a single data dump, an in-between solution was chosen. To enable clients to participate in the prefix query evaluation, clients should be able to retrieve only the data relevant to the evaluated query from the dataset. This requires the dataset to be fragmented, and for the fragments to be structured in a way that enables clients to traverse and prune the search space. In the interest of improving query performance by limiting the amount of HTTP requests necessary for a client to autocomplete a prefix, we took our inspiration from the design of balanced tree structures such as a B-tree [2]. The implementation of the creation algorithm used in this paper makes use of B-tree structures to fragment the dataset. It can be found at `https://github.com/Dexagod/linked_data_tree`.

To create fragmentations of a dataset, the data publisher first has to deciding the properties over which the dataset entities should be indexed. For each chosen property, a separate fragmentation of the dataset is created.

To create a fragmentation, first an indexing search tree data structure is generated, adding all entities in the dataset using the value of the chosen property as key to add to the data structure (the data publisher decides the extent of a data entity). Upon adding all dataset entities, for each node in the tree structure a dataset fragment is generated, stored as a separate file. Such a fragment contains the node information, its relations to other nodes in the data structure, and the

data entities present in the node. To enable the client to traverse the tree structure in the generated fragments, the node and relation data in the fragments are defined in a semantic way as hypermedia controls, using the TREE hypermedia descriptions[8], as depicted in listing 1.1.

To enable clients to find a dataset and its available fragmentations, this dataset information is published semantically as a **collection** object, as seen in in listing 1.1. The different created fragmentations of the dataset are defined as *views* on this collection object. These view properties point to the root nodes of used tree structure and its containing fragment. An optional *shape* property can be provided, defining the base shape (structure) of all entities in the collection. On publishing this collection object on the Web, a client can discover all available fragmentations of the dataset through the view properties present in the object.

```
1  {
2    "@context": {
3      "tree": "https://w3id.org/tree#"
4    },
5    "@id": "#Dataset",                              ///D
6    "@type": "tree:Collection",
7    "tree:shape": "shape.shacl",
8    "tree:view": {
9      "@id": "node1.jsonld",                        ///n
10     "tree:relation": [
11       {
12         "@type": "tree:GreaterThanRelation",     ///defines χ
13         "tree:path": "foaf:name",                ///p
14         "tree:value": "Alice",                   ///v
15         "tree:node": "node2.jsonld",             ///c
16       },
17       ...
18     ]
19   },
20   "tree:member": [ ... ]                          ///array of e
21 }
```

Listing 1.1. An example of the metadata of a response in JSON-LD. A client encountering this relation knows that all data found following the link to `node2.jsonld` will result in a value that is greater than `Alice` for the `foaf:name` property.

For each node in the created tree structure of the dataset, the generated fragment for that node defines a **node** object. This object stores the relations to its child nodes (and their containing fragments). The dataset entities present in the node of the tree structure are stored in the generated fragment as members of the collection object (dataset) as seen on line 20 of listing 1.1.

The relations to the child nodes are defined as **relation** objects, as seen on line 10 of listing 1.1. These semantically define the data found in the subtree of the referenced child node, by specifying the following properties:

1. The relation type, being `LessThanRelation`, `LessThanOrEqualToRelation`, `GreaterThanRelation` or `GreaterThanOrEqualToRelation`. This relation type specifies a comparison operator χ, to to which all entities e_c in the subtree of child node c are evaluated in comparison to the relation value $v_{relation}$;

[8] `https://treecg.github.io/specification`

2. a node property, being the hypermedia link to the child node c and its containing dataset fragment.
3. an optional path p, that is the property path over which all entities e_c in the subtree of child node c are evaluated, and
4. a value $v_{relation}$.

This relation object semantically defines the entities that can be found in the subtree of child node c. E.g. for a path p of *firstName*, a value $v_{relation}$ of *Alice*, and a relation type of `GreaterThanRelation`, the relation defines that all data entities e_c in the subtree of c have value greater than *Alice* for the value of their *firstname* property. With this information, a client evaluating a query over the dataset fragmentation can process the available relations, and prune the ones that do not lead to relevant data entities for the evaluated query. Note that multiple relation objects can defined in a node referencing the same child node, further specifying the entities found in the child node and its subtree.

In order to express the property paths, the design of property paths in the Shapes Constraint Language (SHACL) [4] is reused. As the ordering of characters is important for comparing string based values, unicode ordering is used as a default, as defined by the TREE specification. Flags are available to indicate other orderings used to generate the fragmentation, and have to be followed by clients.

As datasets can contain entities with non-unique values for a given property, the tree structure used to fragment the dataset needs to support duplicate key values. In Modern B-tree techniques [3], duplicate key values are stored once, and subsequent entities with the same value are added in a (paged) array-structure. Since however we are not limited to predefined semantics for relations, we adapted the B-tree splitting algorithm to allow entities with duplicate key values to just be stored in the tree structure. In case a node overflows during the creation of the tree structure, and is split between duplicate key values, this is resolved by having the parent node reference the two new nodes using the relation types `LessThanOrEqualToRelation` and `GreaterThanOrEqualToRelation`. The client is not required to be aware of this adaptation, as it just requires an understanding of the relation semantics to prune the search space.

3.3 Client algorithm

In this section we describe the client algorithm used for the evaluation in Section 4. A client can be asked to evaluate prefix search queries for a given prefix value v_{query} and property path p_{query} over a dataset D (e.g. the client searches for entities in D where the property p_{query} value of *firstName* matches the prefix v_{query} of *Ali*). For this, the client requires a reference to the collection object of D. Upon retrieving this collection object, the client dereferences the root nodes of the available fragmentations of the dataset through the views defined on the collection. This operation can be done at page load times, and should not slow down lookups when used in Web applications.

At the start of the query evaluation process, the client decides the best fragmentation of the dataset to query over. For all fragmentations, the client checks if

the available relations specify a property path $p_{relation}$ that matches the queried property path p_{query} (e.g. the relation specifies it stores relations for the *firstName* property, and the query searches entities matching a prefix value for the *firstName* property). If a fragmentation is found containing such relations, the client continues to evaluate its query over this fragmentation. If no such fragmentation can be discovered, the client will not be able to prune relations of any of the fragmentations (e.g. when the relations provide information over the stored entities *firstName* property, but the client is querying for entities with a *lastName* property matching the prefix *Bob*). In this case the client can retrieve fragments of a random fragmentation without pruning, until the required amount of results is retrieved for the query.

Now that the client has chosen a fragmentation to evaluate the query over, the **recursive traversal process** is started. On retrieval of a new fragment of the dataset (initially the one containing the root node), the client starts by extracting all tree metadata from the fragment. First, the client emits all data entities in the fragment matching the client query. In case the desired amount of results is retrieved, the client is stopped. This design of incremental results contrasts with evaluating the full query on the server-side, where traditionally results are only emitted when the desired amount of results have been found.

If more results are needed, the relations available in the node of the fragment are evaluated, as seen in listing 1.1 on line 10. As multiple relations may reference the same child node, and provide additional constraints to the entities stored in the subtree of that child node, the relations are grouped on the child nodes they point to. If any of the relations pointing to a child node can be pruned, all the relations to that child node can be pruned, as the referenced node and its subtree cannot contain results for the client query

A relation is evaluated by matching the relation path $p_{relation}$ to the query path p_{query}. In the case that these paths do not match (e.g. the relations stores entities for the *firstName* property, where the client queries entities based on their *lastName* property), the client can make no assumptions about the entities stored in the node referenced by the relation, and the relation cannot be pruned.

In the case of matching path properties, the client tries to prune the search space. This is done by comparing the queried prefix value v_{query} to the relation value $v_{relation}$ and the comparison operator χ specified by the relation type (e.g. the `GreaterThanRelation` specifies the $>$ comparison operator). In the case of prefix search, the client now evaluates if the queried prefix value v_{query} is contained in the range specified by the prefix of the same length of the relation value $v_{relation}$ and the comparison operator χ defined by the relation type. A query for the prefix *Car* evaluated over a relation of type `GreaterThanRelation` with a value $v_{relation}$ of *Alice*, requires the client to check if $Car > Ali$. If this comparison holds, the client may retrieve entities relevant for the evaluated query by dereferencing the relation child node. For prefix search, the edge cases where the queried prefix and the prefix of the relation value are equal, or where the relation value is smaller prefix of the queried prefix value, the referenced child node may also contain entities relevant to the evaluated query. When all relations

referencing a child node cannot be pruned by the algorithm, the child node may contain useful data and the **recursive traversal process** is repeated for the child node fragment.

As the client is in control of the query evaluation process, subsequent evaluated queries for incremental prefix values (e.g. `A` `Al` `Ali`) can continue the previous query evaluation, for an updated prefix value, as the results of the updated query are a subset of the results of the previous query. Additional rules can be implemented, such as deciding to not prune relations that provide results within a certain Levenshtein distance of the queried prefix if only few results are found. Different traversal strategies such as breadth-first or depth-first can be implemented and changed during query evaluation depending on the situation. Multithreading can be implemented using a queue system, where a set number of relations can be processed in parallel.

The client implementation used for evaluating the approach can be found at `https://github.com/Dexagod/ldtreeBrowser`.

4 Experiments and results

We define a prefix query as the set of the requests that are performed to retrieve 25 results (if available) that start with that prefix. The *query server* approach consists of a client using a query server that returns a page with 25 results from the dataset (if available) for the queried prefix value and path. The *search tree* is the approach introduced in this paper, where a client traverses a published search tree for entities matching the queried prefix value and path. For this evaluation, we assume that a fragmentation is available for the property over which the prefix query is evaluated. In order to understand the effectiveness of the tree approach, we will measure 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 overall user-perceived performance.

For the experiments, we republished 2 datasets using our approach, and setup a query server interface for these datasets: A dataset of Belgian public transport stops (6 triples per entity, and a total of 72,967 entities), and a subset (BE, FR, NE, LU, DE) of the OSMNames dataset[9] (32 triples per entity, and a total of 3.87m entities), both published using a fragment size (m) of 25 members (a fragment contains 25 data entities, and relations to the 26 child nodes and their containing fragments). For the Belgian public transport stops dataset, a real-world query log was extracted from a server autocompleting Belgian railway stops. For the OSMnames subset, we did not have access to a real-world query set, so a randomized query set was generated. This randomized query set was generated for 1000 target values distributed over 50 simulated user clients, where for each target a series of prefix queries was created, starting at random

[9] `https://osmnames.org/download/`

length prefix, for a randomized amount of subsequent prefixes (e.g. `Lou` `Louv` `Louvai`). In the case of the query server, the client sends a separate request for every prefix in the query log. The used datasets and query logs are made available on Github[10].

The evaluation consists of a server providing the search tree fragmentation, a server providing the server-sided prefix search query interface and a proxy cache in front of these servers on the same network, and a laptop using Wi-Fi with an average ping of ±20 ms. All servers are dedicated machines with a 2x Dual Core AMD Opteron 270 (2GHz) CPU, 4GB Ram and a 80Gb Hdd. All queries are evaluated on a laptop with a Intel M4800MQ CPU and 16GB Ram.

4.1 Cache efficiency

In Fig. 1, we show the server cache hit ratio for a client evaluating the randomized prefix query set over the OSMNames dataset, using an nginx cache at 10% the size of the original dataset. This evaluation was done for the larger dataset, to provide a better overview of the caching behavior. We tested this for both a query server (baseline) as the search tree approach and notice the search tree approach achieves a three times bigger cache hit rate on the server.

As the baseline query server returns results for a specific prefix value, it can only return a cached request in the case of an exact match in requested prefix. In contrast, our approach uses a tree structure to fragment the dataset. A client evaluating a prefix query over a tree structure requires multiple requests for nodes in the tree in contrast to the single request when using the baseline query server. As the client evaluates queries by traversing the tree structure starting from the root node, nodes closer to the root of the tree structure are fewer and therefore have a higher probability of being retrieved for a random prefix query. Because of this, they have a higher probability of being present in the server cache. This explains the higher server cache hit ratio for our approach of evaluating prefix queries compared to the baseline query server.

4.2 Query performance

As the proposed search tree querying approach enables the autocompletion client to emit results during the traversal process of the tree structure, this experiment was setup to see how many results can be achieved within 150 ms, a time span which feels instantaneous to end-users. As query server interfaces are capable of this, and do not provide incremental results (all results are transmitted at once), we focus on the query performance of the proposed *search tree* approach.

Prefix requests are not isolated events, where often the value of the previous request is a prefix of the current request. Because of this, the performance evaluation is done separately for the first three queries (if available) in each series of prefix queries in the used query sets.

[10] `https://github.com/Dexagod/Paper_metadata/tree/master/ISWC2021`

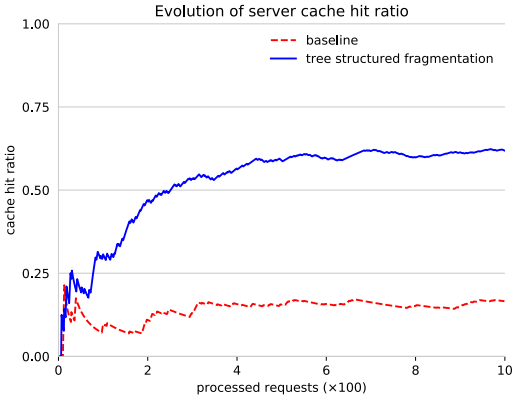

Fig. 1. Average server cache hit ratio evaluating the randomized query set over the OSMNames dataset. The query server stagnates at around 20%, where our proposed approach provides cached result for more than half the requests to the server after an initial warmup period.

The performance is measured as the amount of results the query emits for the evaluation of a prefix query within a 150 ms interval of the client receiving a new prefix value. The results are averaged for all first, second and third requests in all series of subsequent prefix queries in the query logs. For the experiment, the server cache size is set to 10% of the dataset size, and a client cache is set per user that stores all previously retrieved fragments.

In Fig. 2 we can see that for the public transport stops dataset (73k entities), the first evaluated prefix query in a series returns on average just below 15 results in within the 150 ms period. For the subsequent second and third prefix queries, the results can be retrieved faster, producing on average 15 results within a 150 ms interval. For an average query, the first 5 results are shown in a 25 ms interval as a result of server caching, and cached results from previous queries. Subsequent queries in a series returning less results can be explained by the dataset containing less than 15 entities matching the queried prefix.

In Fig. 3, we see that for the subset of the OSMNames[11] dataset (3.87 m entities), the proposed approach results in a slower start for the evaluation of the first prefix queries in a series. This is caused by the randomized nature of the query set used for this dataset. As the first prefix query has a randomized length, it may have results stored deeper in the tree structure of the published dataset fragmentation, requiring a more expensive initial lookup. For subsequent queries in the series, the performance normalizes because of previously cached data, with the second evaluated prefix query returning on average 15 results in the 150 ms interval. A slower average results in the first 25 ms of the query compared to the smaller dataset is caused by the deeper tree structure of the fragmentation.

[11] `https://osmnames.org`

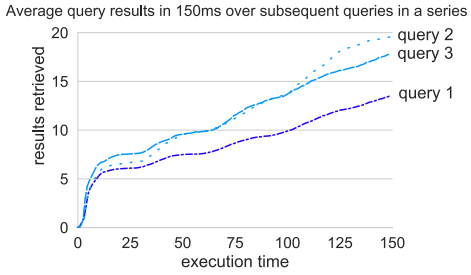

Fig. 2. Increased performance in a series of subsequent prefix queries (e.g. query1: "Lou", query2: "Louv", query3: "Louva") over the transport stops dataset (73k entries) for the proposed approach. First 5 results are shown in a 25 ms as a result of caching and results available from previous queries. Later queries may not have 15 results in the dataset, leading to a lower amount of retrieved results.

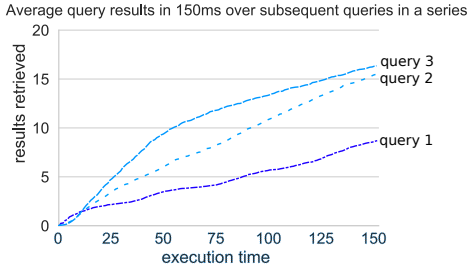

Fig. 3. Slower retrieval of results for the first evaluated prefix query in a series because of randomized length of the first query in the generated query set for the OSMNames dataset (3.87m entities). Subsequent queries provide 15 results in the 150 ms interval, with a slower start because of the deeper tree structure of the dataset fragmentation.

4.3 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 [8]. In a query server approach, developers will aim towards a 100% efficiency. At the cost of efficiency, the search tree approach raises the cacheability. In Fig. 4, we discuss the results for how much efficiency we sacrifice. In Fig. 5 we discuss the bandwidth consumption and the number of HTTP requests that this adheres to.

In the implementation for our approach, we made the decision to allow the data publisher to decide how many data entities can be stored per dataset fragment (identical to the amount of values that can be stored in a node of a generic B-tree implementation). The consequence of this is that the size of a single fragment scales both with the amount of data entities stored in the fragment, as well as with the individual sizes of each of these entities. Because of this, at publication time the average entity size has to be taken into consideration when creating a fragmentation, as this will influence the bandwidth requirement of the interface.

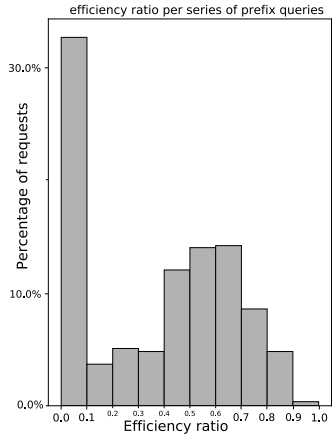

Fig. 4. The efficiency shows a large quantity of 0% queries. This is due to targets that do not result in an answer (not in the collection). For other requests we see the efficiency averages over 50%.

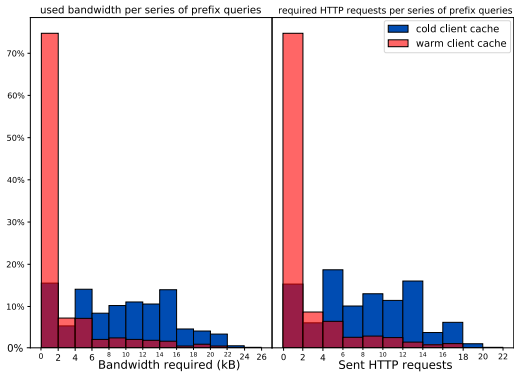

Fig. 5. A max bandwidth of 25kB is consumed for an autocompletion query for this particular dataset and query log. The number of requests for a full query autocompletion range from 0 to 20 requests in worst-case. The y-axis is the % of queries.

4.4 Fragment sizes

In the prior experiments, we always used a fragment size of 25 entities per page. We selected this page size when comparing the query performance of 25, 50, 75 to 100 entities per page. The results are provided in Fig. 6. We notice longer startup times for the initial queries over larger dataset, and that larger fragment sizes perform better for small datasets, but in turn perform worse for larger datasets. This can be attributed to the trade-off between traversal speed and data locality, where for larger datasets traversal speed becomes increasingly important.

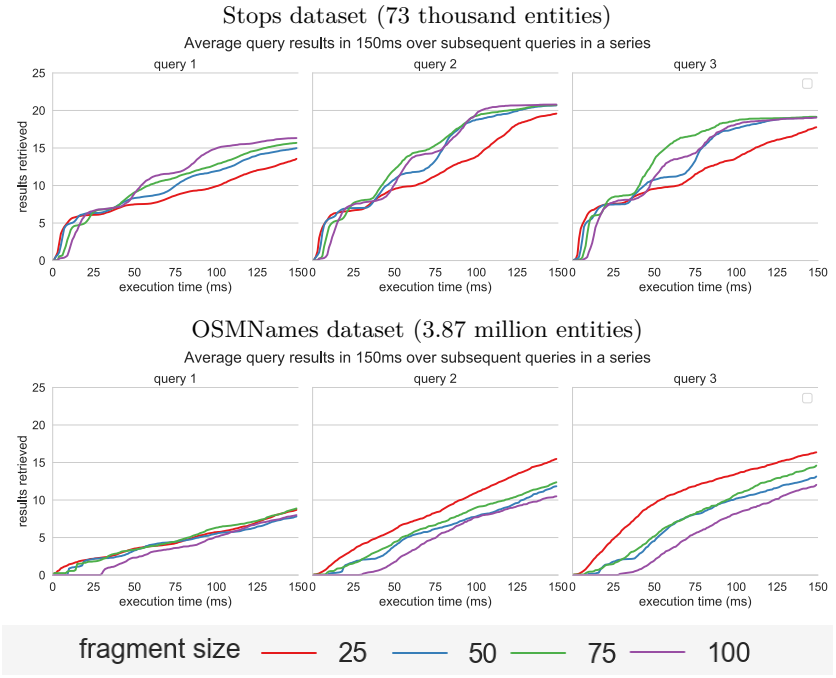

Fig. 6. This figure compares the performance for clients evaluating prefix queries over (Top) the public transport stops dataset (73k entities, 6 triples per entity) and (Bottom) the OSMNames dataset (3.87m entities, 32 triples per entity).

5 Discussion

In contrast to a query server, where the number of HTTP requests stays constant when a dataset grows in size, the amount of HTTP requests needed with the tree approach is proportional to the amount of entities in the dataset. In case of a fragmentation based on a theoretical B-tree, the number of requests necessary to find entities matching a prefix value is decided based on the height of the tree, which can be calculated theoretically for a dataset of n entities 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 entities more to result in an increase in worst-case number of requests necessary. Applied to our dataset: with 25 entities per fragment for a total of 10^5 entities, a depth of 4 would be needed (h_{max}), and thus 4 HTTP requests would be needed in worst-case to show results. 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. 2. Our implementation and experiment confirms this theoretical analysis: for a full series of queries, the experimental results are depicted in Fig. 5.

This is illustrated by the OSMNames dataset, with a maximum depth of 6 for 3.87 million entities, vs. the public transport stops dataset with a maximum depth of 4 for 73k entities. Fig. 3 thus also shows that even for larger datasets, the approach returns timely results.

6 Conclusion

Given that a sufficient number ($\pm$15) of results will be retrieved in a timely fashion ($\pm$150 ms), we can conclude that our approach of fragmenting a dataset as static files can be a viable alternative to a query service, given a dataset fragmentation is published for the queried data property. At the expense of the client having to take part in the query evaluation and consume more bandwidth, the server may work even fully from cache, archive or CDN. The results show using a cache that is 10% the size of the dataset, the search tree approach implemented in this paper reaches a server cache hit ratio that is $\pm$3 times better. Thanks to the TREE hypermedia specification, any search space design that uses the specified hypermedia controls can be used by a generic autocompletion client. The downside however is a larger bandwidth consumption, meaning query response times will be easier impacted by a bad internet connectivity. While we designed this approach for datasets for which setting up a tool like ElasticSearch or a SPARQL endpoint is not worth the effort, the approach can return results in a timely fashion even for large datasets with millions of entities.

The fragment size itself however is a difficult decision to make, and we do not have a silver bullet approach to decide what the best number per dataset fragment would be. In this paper we tested the approach for one specific use case of prefix autocompletion and came to the conclusion that a size of 25 entities per fragment gave the best response times. However, depending on the dataset, the query set used for the benchmark, the level of privacy you want to guarantee and type of text search query, we believe other fragment sizes may be more interesting. Furthermore, the ideal fragment size will also depend on the type of hypermedia search space one implements. In this paper we chose a B-tree approach to prove that file-based fragmentation strategies can produce and acceptable user-perceived performance, yet we certainly do not rule out other search space designs. Future work will be to come up with specific search space designs such as faster querying by adding important entities higher up in the tree, for substring search with automata, for fuzzy matches by clustering by string distance, with a geospatial bias by first adding a geospatial fragmentation to your dataset, etc.

The new client-server relation for prefix search has an effect on the user experience guidelines of Pelias (cfr. Section 2)). (i) Throttling requests can happen differently, as a large amount of requests can be handled from server cache. In a similar way, there is also no danger of out of order responses (ii). As the client

controls the the query evaluation process, subsequent request can filter the previously retrieved results, and continue the on-going query processing to the next prefix. Finally, (iii) using a pre-written client was a guideline when working with the query server design, and remains our guideline here as well. This pre-written client is given more responsibility for the query evaluation process, giving it more flexibility to implement the autocompletion or any text search feature in the way a developer wants. In the same spirit of the Pelias user experience guidelines, we formulate two additional guidelines for publishing a fragmented interface. A caching is the driver behind the scalability of this approach, probably the most important of these guidelines will be (iv) to set caching headers. Both conditional caching with `etag header`, as setting a `cache-control` header are possibilities in different designs. Next, for public datasets, also (v) Cross Origin Resource Sharing (CORS) headers need to be enabled. This will enable application developers to reuse the dataset from a different domain than where the dataset itself is hosted.

References

1. Cai, F., De Rijke, M., et al.: A survey of query auto completion in information retrieval. Foundations and Trends in Information Retrieval **10**(4), 273–363 (2016), https://staff.fnwi.uva.nl/m.derijke/wp-content/papercite-data/pdf/cai-survey-2016.pdf
2. Comer, D.: Ubiquitous b-tree. ACM Computing Surveys (CSUR) **11**(2), 121–137 (1979), https://dl.acm.org/doi/10.1145/356770.356776
3. Graefe, G., Kuno, H.: Modern b-tree techniques. In: 2011 IEEE 27th International Conference on Data Engineering. pp. 1370–1373. IEEE (2011). https://doi.org/10.1561/1900000028, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.7269&rep=rep1&type=pdf
4. Knublauch, H., Kontokostas, D.: Shapes constraint language (shacl).(2017). W3C recommendation (2017), https://www.w3.org/TR/shacl/#property-paths
5. 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), http://comunica.linkeddatafragments.org/
6. 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., Staab, S. (eds.) Proceedings of the 14th ISWC. Lecture Notes in Computer Science, vol. 9366, pp. 128–143. Springer (Oct 2015), https://linkeddatafragments.org/publications/iswc2015-substring.pdf
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**(3), 437–452 (2017)
8. 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 (Mar 2016). https://doi.org/doi:10.1016/j.websem.2016.03.003, http://linkeddatafragments.org/publications/jws2016.pdf