Vrije Universiteit Amsterdam

Bachelor Thesis

Deployment and Evaluation of a New Recommender System for Wikidata

Author: Marta Anna Jansone (2641123)

supervisor: Michael Cochez
2nd reader: Ilaria Tiddi

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 3, 2021

Abstract

Wikidata is a collaborative central storage repository that collects structured data and provides support for other wikis of the Wikimedia movement. The data on the site is represented in terms of Items and Property statements, which can be edited by humans or machines. To support the editing of a Wikidata Item in terms of adding additional Properties, an association rule-based extension *PropertySuggester* is used. A paper published in May of 2020 introduced the *SchemaTreeRecommender*, which is an alternative approach of assigning properties to an Item that were previously not attributed to it. The *SchemaTreeRecommender* assigns properties to an Item employing the maximum-likelihood property recommendation approach. The evaluation of the two recommendation systems showed that the *SchemaTreeRecommender* outperformed the *PropertySuggester* in all performance metrics. In this work the cache efficiency of the *SchemaTreeRecommender* is improved leading to a 7% decrease in request latency and no significant difference in CPU usage. Further, the *SchemaTreeRecommender* is deployed to Wikidata and the evaluation between the two recommendation systems is carried out in the production environment of Wikidata.

1 Introduction

Wikidata is a collaborative central storage repository collecting structured data. The leading purpose of Wikidata is to provide support for other wikis of the Wikimedia movement (Vrandečić, 2012). Wikidata provides a shared knowledge base that can be extended and reused. This is enabled through open editing, which implies that the information provided on the site can be edited by any user. The data available on Wikidata is multilingual. Other wikis (e.g. Wikipedia) might have independent editions for each of the languages, however, as the data stored by Wikidata is universal, it is necessary that all values have direct translations (Vrandečić and Krötzsch, 2014). The repository consists of Items represented by a Q that is followed by a number. Each Item has a label, a description and any amount of aliases associated with it.

An Item can be described by statements, which are built from a pair of a property and a value. Properties are represented by a P that is followed by a number. For instance, given an Item 'Douglas Adams' represented by an entity code Q42 it can be described by a statement pair where the property is 'educated at' (P69) and the value is 'St. John's College' (Q691283). Moreover, each statement can be expanded through the use of qualifiers. Qualifiers are applied to statement pairs in order to further describe the value of a property given in a statement. By using qualifiers, the property 'educated at' (P69) can be further elaborated to include the start time, end time, academic degree and major. The visual representation of this relationship and its' layout on a Wikidata entity page can be seen in Figure 1.

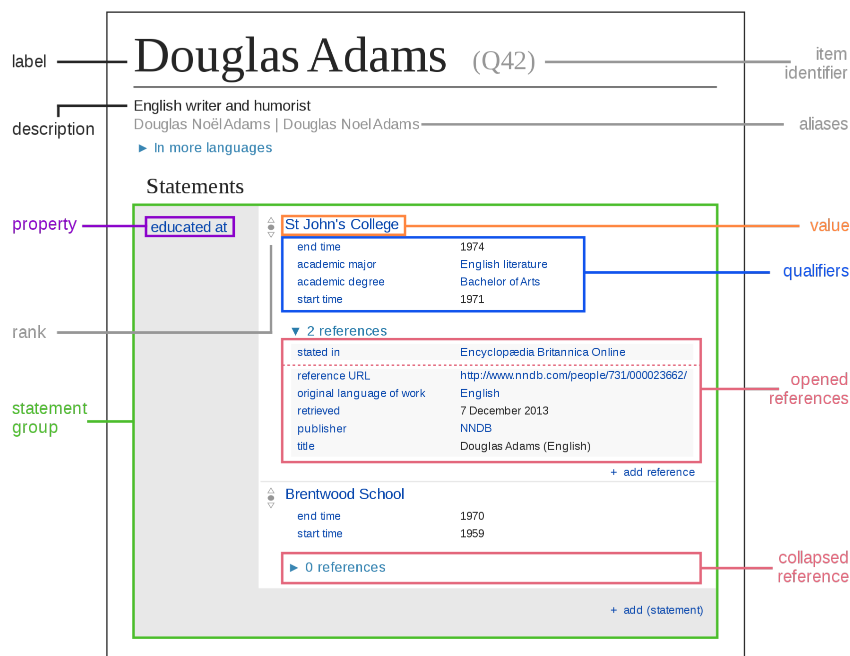


Figure 1: Wikidata entity page representation[a].

[a]Image source: https://www.wikidata.org/wiki/Wikidata:Introduction

The Wikidata database can be edited by the supporting community, therefore, both

new Items and Property statement pairs can be added by either humans or machines designed for the task. However, given the growing amount of properties available, it is a challenging task to assign relevant properties to entities. The current software behind recommending new properties to users is the *PropertySuggester*[1]. *PropertySuggester* recommends properties based on the frequency of other entities that have the same 'instance of' (P31) value and the properties that are present on the item. This approach is based on association rules. The leading goal of such rules is to locate co-occurring items within a database (Zangerle et al., 2016). A paper by Gleim et al. (2020) introduced the *SchemaTreeRecommender*[2], which is an alternative approach to recommend additional properties to an entity.

The *SchemaTreeRecommender* uses the maximum-likelihood of properties to suggest additional properties that were not previously attributed to the entity. The *SchemaTreeRecommender* employs a data structure *SchemaTree*, which is a compact trie-based representation of property and type co-ocurrences. The structure is stored in an adapted trie construction frequent pattern tree (FP-tree) for probability calculations and retrieval of the property pairs. The *SchemaTreeRecommender* was evaluated against the state-of-the-art Wikidata *PropertySuggester* in regards to the performance and the quality of the recommendations. When evaluating the different variations of the *SchemaTree* approach against the state-of-the-art Wikidata *PropertySuggester*, the *SchemaTreeRecommender* outperformed the current system in all performance metrics (Gleim et al., 2020).

The *SchemaTreeRecommender* outperformed the *PropertySuggester* in the experiments conducted by Gleim et al. (2020), which suggests that the *SchemaTreeRecommender* would provide greater support for property recommendations within Wikidata. To further investigate this assumption, the two recommendation systems have to be evaluated in the Wikidata production environment. This paper aims to integrate the *SchemaTreeRecommender* in the existing Wikibase, the knowledge base software driving Wikidata, in order to evaluate and compare the performance of the existing *PropertySuggester* against the performance of the *SchemaTreeRecommender*.

First the changes made to the existing code base of the *SchemaTreeRecommender* will be discussed followed by the description of the integration process of the recommender within Wikibase and, thus Wikidata. The evaluation of the two systems will be carried out through A/B testing and additional further work will be discussed, which will be the topics concerning the last sections of the paper.

2 *SchemaTree* Adaption

In this section the preliminary work and adjustments made to the original *SchemaTree* code will be discussed in detail. The data-types and structures used to represent the tree in Golang will be introduced. Along this, the experiments that were carried out in order to locate potential areas for improvement in regard to CPU usage and request duration will be presented.

[1] https://www.wikidata.org/wiki/Q86989962

[2] https://github.com/lgleim/SchemaTreeRecommender

2.1 Introduction to the *SchemaTree*

The *SchemaTree* is constructed using the full RDF Dumps of Wikidata. The use of RDF was introduced to Wikidata due to the necessity for an exchange format for Wikidata with SPARQL query functionality. The information available on the site closely corresponds with the RDF model as each entity can be seen as the subject of a triple and the property statements associated with that entity can be linked to predicates and objects associated with the subject (Hernández et al., 2015). Within the RDF file each entity is represented as a combination of a data node and an entity node. The data node describes metadata about the given entity record and the entity node describes the entity data. The data contained within the entity node provides information that concerns the labels, aliases, the description and the statements related to the entity. An example of a data node is shown in Figure 2 and an example of an entity node is provided in Figure 3.

```
wdata:Q2 schema:version "59"^^xsd:integer ;
    schema:dateModified "2015-03-18T22:38:36Z"^^xsd:dateTime ;
    a schema:Dataset ;
    schema:about wd:Q2 .
```

Figure 2: RDF schema used by Wikidata for the data node [a].

[a]Image source: https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format

```
wd:Q3 a wikibase:Item ;
    rdfs:label "The Universe"@en ;
    skos:prefLabel "The Universe"@en ;
    schema:name "The Universe"@en ;
    schema:description "The Universe is big"@en ;
    skos:altLabel "everything"@en ;
    wdt:P2 wd:Q3 ;
    wdt:P7 "value1", "value2" ;
    p:P2 wds:Q3-4cc1f2d1-490e-c9c7-4560-46c3cce05bb7 ;
    p:P7 wds:Q3-24bf3704-4c5d-083a-9b59-1881f82b6b37,
         wds:Q3-45abf5ca-4ebf-eb52-ca26-811152eb067c .
```

Figure 3: RDF schema used by Wikidata for the entity node [a].

[a]Image source: https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format

Further, the *SchemaTree* structure will be introduced. The description is directly adapted from Gleim et al. (2020).

The *SchemaTree* is constructed to contain the data structure which is further used to serve the property recommendations. These recommendations are calculated given the maximum-likelihood of properties. Given an entity E with properties S such that $S = \{s_1, ..., s_n\} \subseteq A$ in a knowledge graph where A is set of all available properties, the task of recommending maximum-likelihood properties is described as finding the property $\hat{a} \in A \setminus S$ such that

[2]Image source: https://www.wikidata.org/wiki/Wikidata:Introduction

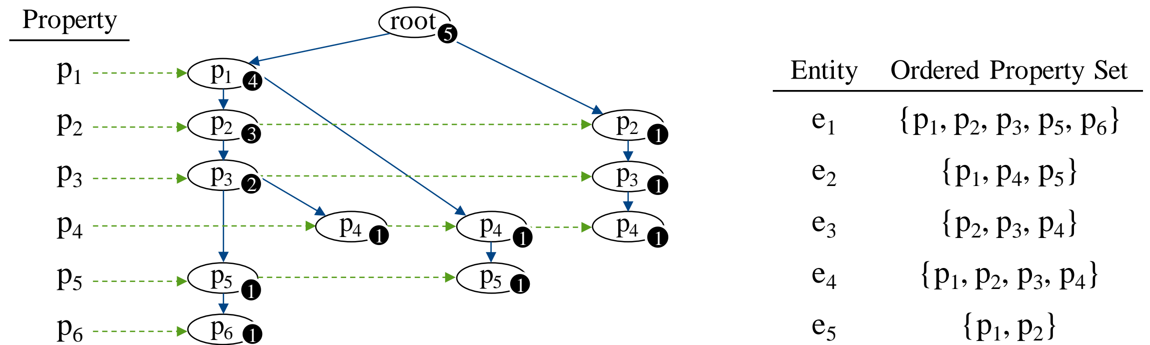
$$\hat{a} = \underset{a \in (A \setminus S)}{\operatorname{argmax}} P(a|\{s_1, ..., s_n\}) = \underset{a \in (A \setminus S)}{\operatorname{argmax}} \frac{P(\{a, s_1, ..., s_n\})}{P(\{s_1, ..., s_n\})} \tag{1}$$

where $P(\{t_1, ..., t_m\})$ is the probability that a selected entity has at least the properties $t_1, ..., t_m$. Following this, the properties that are recommended are the ones which most often occur together with the properties that the given entity already has. (Gleim et al., 2020)

Through frequentist probability interpretation and using an approach of grouping RDF triples by subject, the joint probabilities are approximated by their relative frequency of occurrence. The absolute frequency of a set of properties is then denoted as $supp(A)$ and through reformulating Eq. 1 the most likely property recommendation can be estimated as follows:

$$\hat{a} \simeq \underset{a \in (A \setminus S)}{\operatorname{argmax}} \frac{supp(a, s_1, ..., s_n)}{supp(s_1, ..., s_n)} \tag{2}$$

The *SchemaTree* structure is introduced in order to reduce the time consumption required when computing the recommendations given the large amounts of data contained by Wikidata. The *SchemaTree* is an adapted version of the trie construction FP-tree that was originally introduced by Han et al. (2000). The structure is created using the 2-pass tree construction approach and the support descending property ordering together with lexicographical ordering as proposed by Gyorodi et al. (2003).



Entity	Ordered Property Set
e_1	$\{p_1, p_2, p_3, p_5, p_6\}$
e_2	$\{p_1, p_4, p_5\}$
e_3	$\{p_2, p_3, p_4\}$
e_4	$\{p_1, p_2, p_3, p_4\}$
e_5	$\{p_1, p_2\}$

Figure 4: The SchemaTree displayed on the left side of the figure constructed from the property sets on the right side (Gleim et al., 2020).

Figure 4 depicts the tree which is constructed given the property sets on the right side of the figure. The property sets are ordered by the support each of the properties has (e.g. within all of the sets p_1 and p_2 occur a total of four times each and p_3 occurs a total of three times, therefore, p_1 and p_2 will be ordered before p_3 within the sets). The tree is then constructed by inserting all of the properties within the sets starting at the root node.

Additionally, in order to improve recommendations in instances where the input set size is large, the *SchemaTree* employs backoff strategies, which either reduce the input set size or split it into two separate sets. The available backoff strategies are:

SplitPropertySet splits the input property set into two smaller input sets. This is done through sorting the properties supplied in the request based on the global property support ordering and split into two subsets. A recommendation is performed on both subsets. The properties which appeared in the original request are removed and the two recommendation lists are merged.

DeleteLowFrequency deletes the properties that have the lowest property support from the properties supplied in the request and perform a recommendation using the remaining properties.

The backoff strategies can be triggered by one of the two backoff conditions, which are:

TooFewRecommendations initiates the execution of a backoff strategy if a threshold of T1 for the number of returned properties is not satisfied.

TooUnlikelyRecommendations initiates the execution of a backoff strategy if a threshold of T2 for the average probability of the top 10 recommendations returned is not satisfied.

2.2 Code base

The original *SchemaTreeRecommender*[3] code consists of ten packages total, required for creating a new *SchemaTree* from an RDF file and loading the tree from an encoded file for the purpose of serving it over an HTTP connection. The modules and their usages are as follows:

schematree contains the data structures required by the tree and the standard recommendation algorithm

strategy module responsible for executing the required recommendation procedure given an input assessment

preparation used to split the input dataset, prepare the dataset in N-Triples format. Additionally, enables the possability to configure how the split and the filtering of the dataset is carried out

io methods used to assist with parsing and writing of the N-Triples files

glossary module used to map the property URLs with the corresponding labels and descriptions in each of the available languages

configuration module used to read the configuration workflow files, used to set the strategy of the recommendation procedure

backoff contains the possible backoff strategies (*DeleteLowFrequency* and *SplitPropertySet*)

assessment constructs the recommendations from a user-provided input

server module used to set up an API endpoint using a HTTP server to provide communication with the *SchemaTreeRecommender*

[3] The original *SchemaTreeRecommender* code: https://github.com/lgleim/SchemaTreeRecommender

evaluation module used carry out the experiments and evaluate the results described by Gleim et al. (2020)

The deployment of the *SchemaTreeRecommender* to Wikidata was coordinated with a team of people working at Wikidata. Due to security concerns regarding the parsing of files within the same code base that would interact with Wikidata and for better integration of the *SchemaTreeRecommender* with the existing *PropertySuggester* code[4], the code for *SchemaTree* was separated into two parts. A single repository was created for the code required to serve the tree and the recommendations and a repository was created for the code necessary for creating the *SchemaTree* from a RDF file. The modules available in the recommender server code base (*RecommenderServer*[5]) are 1. *schematree*, 2. *strategy*, 3. *configuration*, 4. *assessment*, 5. *backoff*, 6. *server*. The modules remaining in the code base used for creating the *SchemaTree* from a RDF file (*SchemaTreeBuilder*[6]) are 1. *schematree*, 2. *preparation*, 3. *io*. *Glossary*, the package responsible for mapping the property URLs with the correct labels and descriptions in each of the corresponding languages, and the *evaluation* package were not necessary for the purposes of deploying *SchemaTree* to the *PropertySuggester* extension.

2.3 Performance improvements

The code used for the tree is written in Golang and the *SchemaTree* structure used for calculating the recommendations is constructed using user-defined data structures *SchemaTree* and *SchemaNode*. The *SchemaTree* structure holds the root node of the tree, which is of *SchemaNode* data structure. Within the *SchemaNode* the parent, ID, traversal pointer and the total frequency of each of the nodes is held. Additionally, each node contains the pointers of the *SchemaNode* structures that are the children of the given node. In the original *SchemaTreeRecommender* code, all children of a single node are stored in distinct memory objects that are related to each other with pointers. The cache performance of the program could be improved by mitigating the performance bottlenecks between the CPU and the RAM through allocating data structures in a manner that increases the reference locality (Chilimbi et al., 1999). Thus, to improve the efficiency of the *SchemaTreeRecommender* further, the following hypothesis was proposed:

1. The cache efficiency could be improved and the input-output incurred wait times could be reduced by storing child nodes directly within an array structure once the tree has been statically constructed, thus, reducing the latency and CPU usage when executing a request.

In order to evaluate the hypothesis, additional variants of the *RecommenderServer* were created. Within the modified serving code of the *SchemaTree*, the *SchemaNode* structure was adjusted to store a set number of pointers to child nodes directly within the node itself while the reminder of the children remain within the memory object related to each other with pointers. The changes made to the *SchemaNode* data structure are shown in Figure 5. In the right side of Figure 5 the constant N denotes the number of child nodes to be stored directly within the parent node.

[4] https://gerrit.wikimedia.org/r/admin/repos/mediawiki/extensions/PropertySuggester

[5] https://github.com/martaannaj/RecommenderServer

[6] https://github.com/martaannaj/SchemaTreeBuilder

```
                                               const N = 1
type SchemaNode struct {                       type SchemaNode struct {
   ID         *IItem                              ID            *IItem
   parent     *SchemaNode                         parent        *SchemaNode
   Children   []*SchemaNode      ------>          FirstChildren [N]*SchemaNode
   nextSameID *SchemaNode                         Children      []*SchemaNode
   Support    uint32                              nextSameID    *SchemaNode
}                                                 Support       uint32
                                               }
```

Figure 5: Changes made to the *SchemaNode* data structure in order to store children of the node directly within the node.

All of the tests were conducted on a Intel Xeon Silver 4210R processor (2.40 GHz) and 264 GB of RAM. When comparing the different variants of the *SchemaTree* the full Dumps of Wikidata acquired as of May 12, 2021 were used. The dataset was split into a training set and a testing set using a 1 in 10000 split. Further, the training set was used to construct the *SchemaTree* and the testing set was used to evaluate the performance in regards to latency (ms) and CPU usage. For all different variants of the *SchemaTree* evaluated, the training and testing sets were kept constant. When conducting the performance experiments the backoff approach used was the *SplitPropertySet* together with the *everySecondItem*[7] splitter and the average merging strategy. The backoff strategy was triggered by the backoff condition *TooFewRecommendations* with a T1 threshold of one. This decision was made as the evaluation results in Gleim et al. (2020) paper demonstrated that this specific set-up of the *SchemaTree* proved to outperform all other variants.

The experiments were carried out on the original version of the *SchemaTree* and on the additional variants with storing a single, two and three child nodes directly within the parent node (setting the constant N depicted in Figure 5 to $1, 2$ and 3). Figure 6 depicts the average request duration for the different variants of the *RecommenderServer*. In Figure 6 (a) the request duration is recorded per set size, which indicates the number of properties and types that were used when issuing the request. In Figure 6 (b) the request duration is recorded on the basis of the number of non-types, which is the number of properties that were supplied in the request. Figure 6 (a) indicates that all versions of storing pointers directly within the node outperform the original version of an array of pointers by a slight fraction up till the set size of roughly 10. However, with the increase of set size, only the *RecommenderServer* with a single pointer stored directly in the node outperforms the original structure of the service. Similarly, when the request duration is compared in terms of number of non-types, only the *RecommenderServer* with a single pointer stored within the node outperforms all other setups.

	Original	3 pointers	2 pointers	1 pointer
Latency (ms)	64.87	65.48	66.32	60.32
%CPU used	3030%	3007%	2979%	2956%

Table 1: Average latency of requests and percentage of CPU used during the tests.

[7] *everySecondItem* is a method of splitting the properties provided in the request employed during the backoff strategy of SplitPropertySet. Each property at an even position is ordered in subset P1 while each property of an odd position is ordered in subset P2.

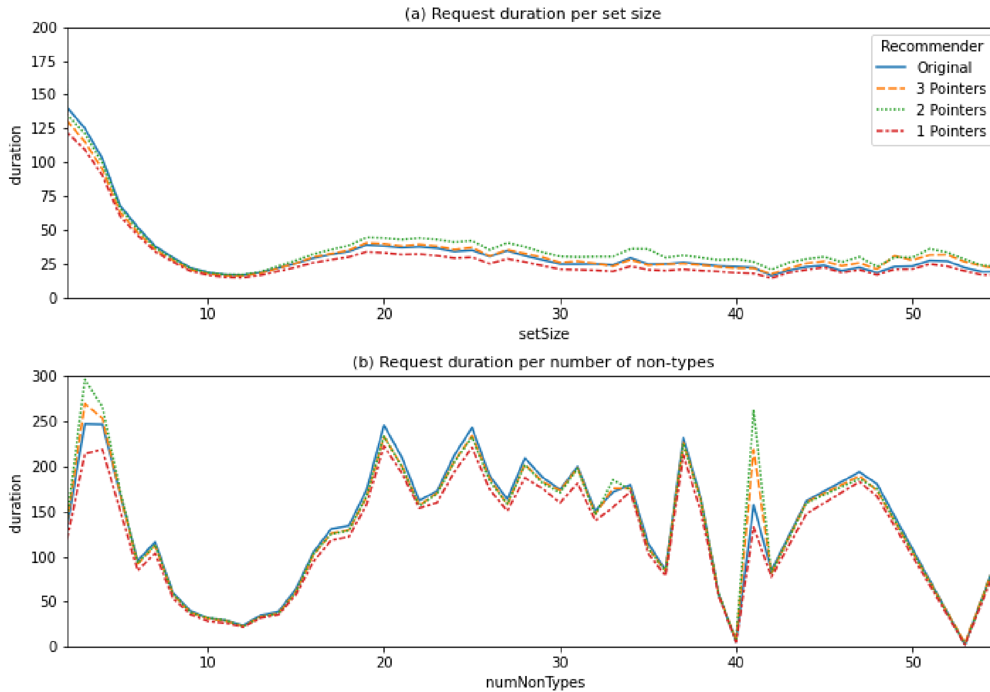


Figure 6: Latency of the request for different set-ups of the SchemaTree (ms).

The results depicted in Table 1 display the average latency in milliseconds and the average percentage of CPU usage for all four versions of the *RecommenderServer* evaluated. When storing a single child node directly within the parent node, the *RecommenderServer* saw an improvement of approximately 7% in regards to request latency, while there was no significant impact on the CPU usage.

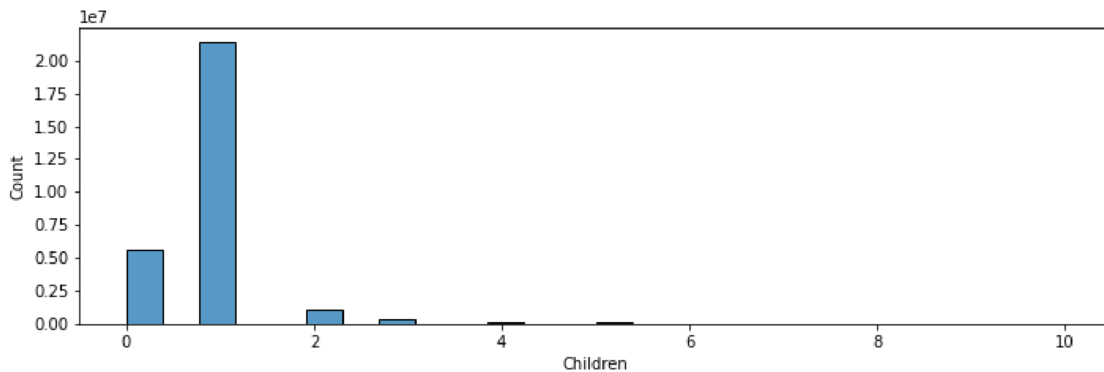


Figure 7: Frequency of children per node in the *SchemaTree*.

The behaviour visible in Figure 6 and Table 1 can be explained by the frequency of child nodes per node relationship depicted in Figure 7. The x-axis of the figure portray the number of children a node has while the y-axis show the total count of such nodes

within the tree structure. From Figure 7 it can be estimated that roughly 74% of nodes in the *SchemaTree* structure have a single child.

Given the experiment results the *RecommenderServer* version deployed to Wikidata contains a single pointer to a child node stored directly within the parent node. The API call used to serve the property recommendations to Wikidata handles approximately 0.5 - 1.5 requests each second[8]. Therefore, the small performance improvement would still have a significant impact given the scale of Wikidata, which currently contains approximately 94000000 items[9].

3 Deploying to Wikidata

This section concerns the process of integrating the *SchemaTree* code within the existing Wikidata infrastructure. Alongside, the original implementation and the necessary adjustments to the original *PropertySuggester* extension will be explored.

3.1 Current *PropertySuggester* implementation

The recommendations for additional properties in the current setup of Wikidata are served by the *PropertySuggester*, which is an extension for Wikibase. The request for property recommendations gets triggered when the user attempts to add new statements to a given Item. The current implementation of the *PropertySuggester* extension requires that the *wbs_propertypairs* SQL table is present within the database of the wiki. When the API call to *'wbsgetsuggestions'* is triggered from within Wikidata, the *PropertySuggester* extension is executed. The properties which are then suggested are acquired by using the association rule approach and querying the information from the *wbs_propertypairs* table. The broad overview of this process can be viewed in Figure 8 where the API call gets triggered from within Wikidata, which in turn executes the *PropertySuggester* extension and leads to an SQL query to the database associated with the extension.

The API call receives the following parameters[10]:

entity a string that represents the Q code of the current entity page

properties alternatively, if an entity is not provided the parameter 'properties' should be used with a list of P codes

limit the maximum number of suggestions to return

continue the offset from which to return the suggestions

language a string representing the language in which the suggestions should be returned

context a string representing the context of the suggestions to return (either item, qualifier or reference)

include a string representing which suggestions should be included (e.g. whether depreciated properties should be provided)

search a string representing the query the user has entered

[8] https://grafana.wikimedia.org/d/000000559/api-requests-breakdown?orgId=1&refresh=5m&var-metric=p99&var-module=wbsgetsuggestions&from=now-30d&to=now

[9] https://www.wikidata.org/wiki/Wikidata:Statistics

[10] https://www.wikidata.org/w/api.php?action=help&modules=wbsgetsuggestions

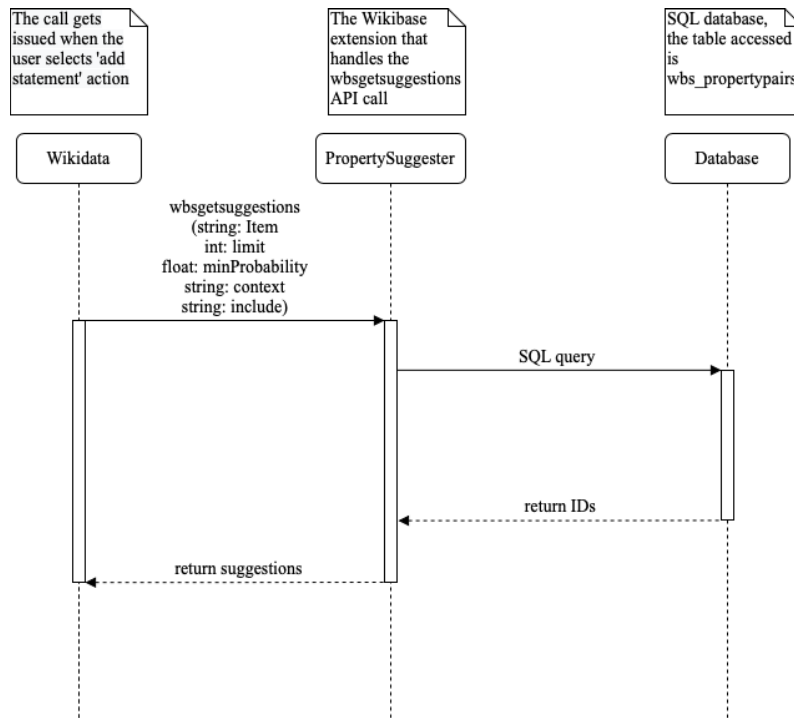


Figure 8: Sequence diagram for a general overview of the execution of the original *PropertySuggester* code.

3.2 Adapting the *PropertySuggester* extension

In order to integrate the *SchemaTreeRecommender* within the *PropertySuggester* extension it is necessary that the *RecommenderServer* is deployed as a service to Wikimedia Cloud Services. For integrating the *RecommenderServer*, the service selected was Cloud VPS [11], which is a cloud computing infrastructure for projects related to the Wikimedia movement. To further make the *RecommenderServer* accessible to requests from outside the cloud infrastructure, a docker image was created using Blubber[12] and it was deployed as a *systemd* service within the Cloud VPS instance. HTTP and HTTPS proxies were set up to make the port used by *RecommenderServer* publicly available. This integration will remain active for the purposes of carrying out A/B testing and evaluation of the two recommendation systems. In the instance where the *SchemaTreeRecommender* outperforms the *PropertySuggester*, the *RecommenderServer* has to be deployed to the production cluster of Wikidata to ensure that the correct security protocols are obeyed.

Due to the evaluation purposes of this work, it was decided that A/B testing has to be implemented. A/B testing is a controlled experiment, which can be conducted in an online environment, therefore, allowing for data collection. Such tests have shown to create a more accurate understanding of what customers (users of Wikidata) value (Fabijan et al., 2017). In regard to this work the decision to implement the data collection process as an A/B test was to provide unbiased results about the usability and quality

[11] https://wikitech.wikimedia.org/wiki/Portal:Cloud_VPS
[12] https://wikitech.wikimedia.org/wiki/Blubber

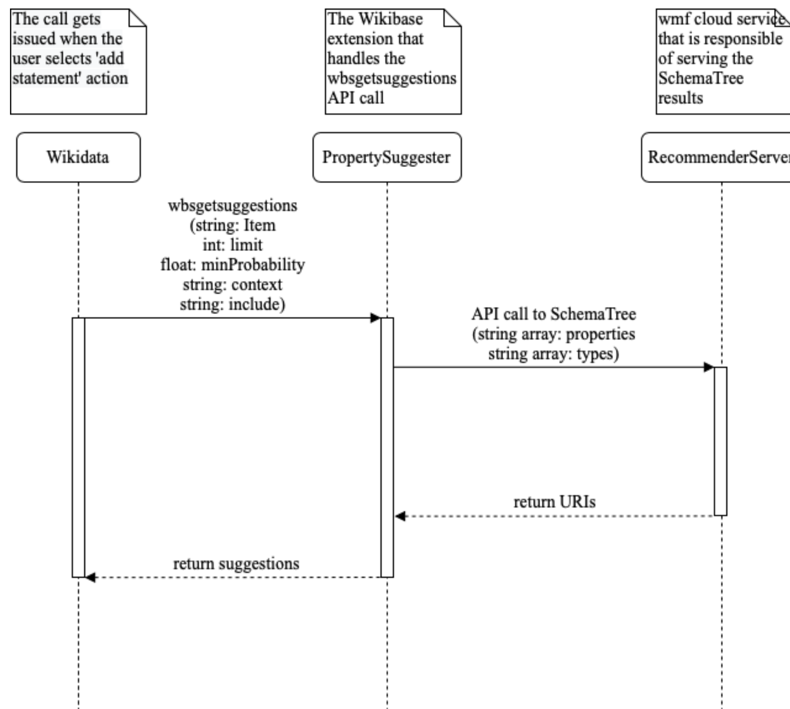


Figure 9: Sequence diagram for a general overview of the execution of the *SchemaTreeRecommender* within the *PropertySuggester* code.

of both recommendation systems. The existing *PropertySuggester* extension was further adapted to handle responses using both the original rule based suggester and the updated maximum-likelihood *SchemaTree* recommender[13]. By employing this approach it was also ensured that A/B testing can be integrated within the same API (*'wbsgetsuggestions'*) call without making it noticeable within the user interface. The general structure of handling the request with the *SchemaTree* recommender can be seen in Figure 9, while Figure 10 depicts a lower level description of how both recommenders are integrated within the *PropertySuggester* extension.

Important to note is that each API call to *'wbsgetsuggestions'* contains the *context* parameter. While the *PropertySuggester* is capable of handling all three possible contexts, the *SchemaTreeRecommender* issues responses only if the context provided is *'item'*. Hence, A/B testing and data collection occurs only in the instances where the context supplied in the request is *'item'*. In addition, the implementation allows for a default recommender to be set and in the instance where the request for the *SchemaTreeRecommender* times out, a fallback to the original *PropertySuggester* will occur.

Moreover, when introducing the *SchemaTreeRecommender* and A/B testing to the *PropertySuggester* extension the list of possible API call parameters was extended to include:

[13]Code patch for integrating the *SchemaTreeRecommender*: https://gerrit.wikimedia.org/r/c/mediawiki/extensions/PropertySuggester/+/689161

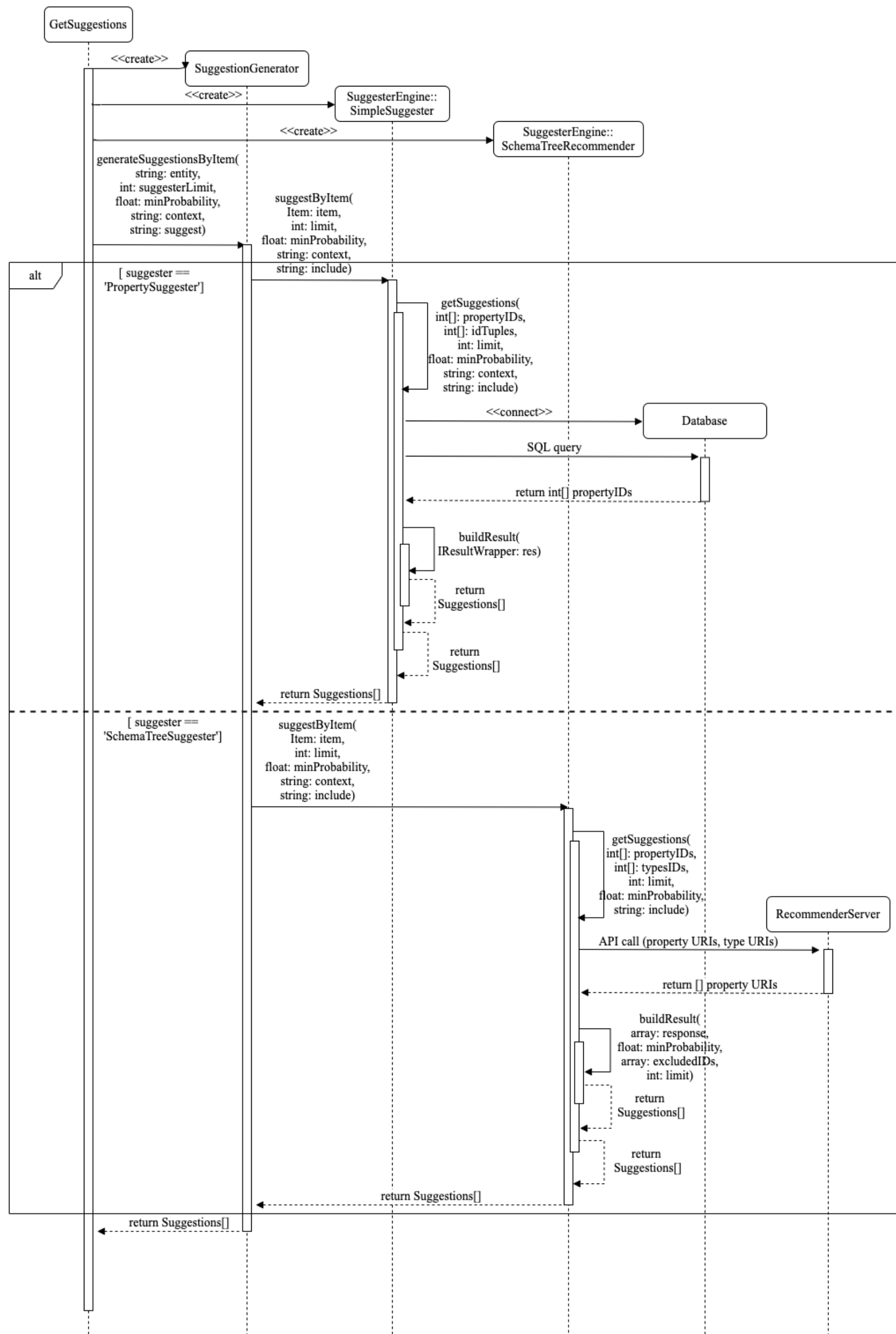


Figure 10: Sequence diagram for the execution of both of the recommenders[a].

[a]The alternative block represents the A/B testing execution of only one of the recommenders. Additionally, the sequence diagram does not represent the full extent of the operations carried out by the *PropertySuggester* extension. For readability purposes, the initialization order of the instances has been changed in the diagram.

types if an entity ID is not provided, it is possible to provide a list of types, this parameter

can be used in combination with the ***properties*** parameter

event if A/B testing is enabled this parameter will contain the event identifier which is used to link events logged client-side with the events logged server-side

4 Evaluation

This section concerns the evaluation of the *SchemaTreeRecommender* in comparison to the association rule based *PropertySuggester* currently used by Wikidata. The further analysis regard the results concerning the quality and the performance of the two recommender systems.

The existing association rule based *PropertySuggester* and the *SchemaTreeRecommender* are evaluated under three hypotheses. These hypotheses are aimed to explore whether objectives concerning user experience and information quality when using the property suggestion field within Wikidata are improved.

1. The *SchemaTreeRecommender* leads to reduced time consumption when a new property is added
2. The quality of the recommendations served by the *SchemaTreeRecommender* is higher than the quality of those served by the original association rule based recommender
3. The *SchemaTreeRecommender* provides greater support for users who are less experienced with the Wikidata site

4.1 Preparation

In order to prepare for the evaluation of the two recommender systems, an additional patch[14] for the *PropertySuggester* extension was created employing the *EventLogging* extension[15]. Moreover, a patch[16] containing the schemas required for event logging both client-side and server-side were created. To carry out the evaluation between the two recommender systems the following values are logged:

propertysuggester_name a string representing the title of the suggester that handled the request (*'PropertySuggester'* or *'SchemaTreeSuggester'*)

entity_id a string representing the Q code of the entity page being edited

existing_properties an array of strings containing the P codes of the existing properties the entity has

existing_types an array of strings containing the Q codes of the existing types the entity has

request_duration_ms the latency of the request in milliseconds

[14]Code patch for event logging: https://gerrit.wikimedia.org/r/c/mediawiki/extensions/PropertySuggester/+/694496

[15]https://www.mediawiki.org/wiki/Extension:EventLogging

[16]Code patch containing the evnet logging schemas: https://gerrit.wikimedia.org/r/c/schemas/event/secondary/+/689152

add_suggestions_made an array containing the P codes of the additional suggestions the suggester recommends

language_code a string representing the language in which the request was made (provided in the API call)

rank_selected the rank of the property which the user selects

recommendation_selected a string representing the URI of the property the user selects

num_characters the number of characters the user looks up before selecting a property

user_id a string representing a hashed user ID or an indicator that the user was not logged in when making the request

Values concerning the *event_id* and the *session_id* are also logged in order to link the events occurring client-side with the requests executed server-side. The *session_id* is used for the purpose of linking the event occurring in the instance of the user selecting a property with the events leading up to it. Additionally, all events logged contain a timestamp representing the moment when the specific event was issued.

4.2 Evaluation procedure

To evaluate the two recommender systems the data described in the section regarding evaluation preparation is aggregated with regard to the set size (the total number of existing properties and types the entities have). This ensures that the two recommenders used can be objectively evaluated given that the speed and the quality of the recommendations returned is dependent on the amount of properties supplied to the request.

Further, the *PropertySuggester* and the *SchemaTreeRecommender* will be evaluated with regard to the performance and the quality of the responses served when the requests are issued by users who are logged in and by users who are not logged in. The data acquired will be gathered in two groups each corresponding to user status of being a registered member of Wikidata or not.

4.3 Metrics

In order to objectively compare the performance of the two recommenders, the following metrics are employed regarding all grouping of entities described in the section of evaluation procedure:

Rank the average position of the property the user selects in the list of properties suggested by the recommender

Stddev the standard deviation of the ranks

Latency the average time to receive the list of recommendations (in milliseconds)

Number of characters (NumOfChar) the average number of characters the user queried before selecting a property

TopX the percentage of all recommendations issued where the suggested property selected by the user was in the top X recommendations

Time the average time it took the user to select a property to add to the entity page in seconds (measured as the time between the timestamps of the first event logged and the event of selecting the property logged)

4.4 Evaluation results

The results further discussed will address each of the three hypotheses concerning the evaluation of the two recommender systems. The results visible in Tables 2, 3, 4 and 5 are grouped by the total set size.

4.4.1 Time expenditure

Recommender	Latency (ms)	Time (s)
SchemaTreeSuggester	Results	pending
PropertySuggester		

Table 2: Results describing the latency for executing the requests and the average time it took a user to add an additional property to the entity.

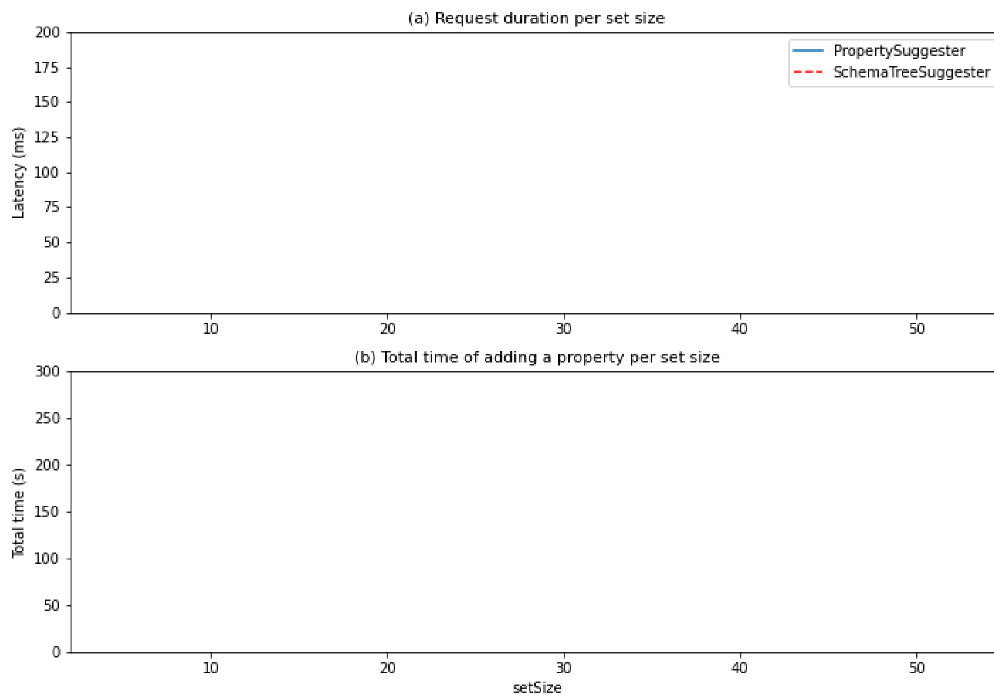


Figure 11: The average request duration and the total time taken to add a new property to a Wikidata entity page. The results are grouped by set size.

4.4.2 Recommendation quality

Recommender	Rank	Stddev	Top1	Top3	Top5	NumChar
SchemaTreeSuggester			Results	pending		
PropertySuggester						

Table 3: Results describing the quality of the recommendations provided by the *SchemaTreeRecommender* and the *PropertySuggester*.

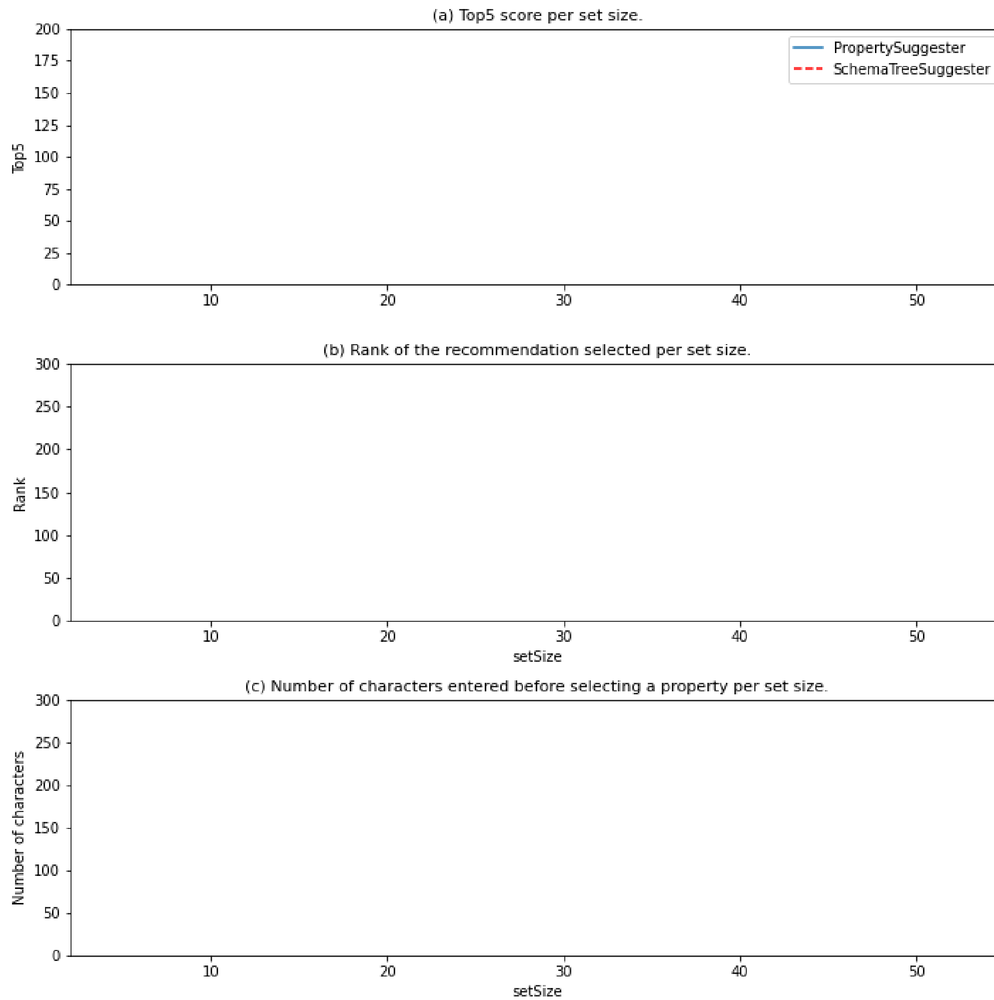


Figure 12: The average Top5 score, rank and the number of characters entered by a user before selecting a property. The results are grouped by set size.

4.4.3 User guidance

Recommender	Rank	Stddev	NumChar	Top1	Top3	Top5	Latency (ms)	Time (s)
SchemaTreeSuggester			Results					
PropertySuggester			pending					

Table 4: Results describing the statistics for the *SchemaTreeRecommender* and the *PropertySuggester* for logged in users.

Recommender	Rank	Stddev	NumChar	Top1	Top3	Top5	Latency (ms)	Time (s)
SchemaTreeSuggester			Results					
PropertySuggester			pending					

Table 5: Results describing the statistics for the *SchemaTreeRecommender* and the *PropertySuggester* for users who are not logged in.

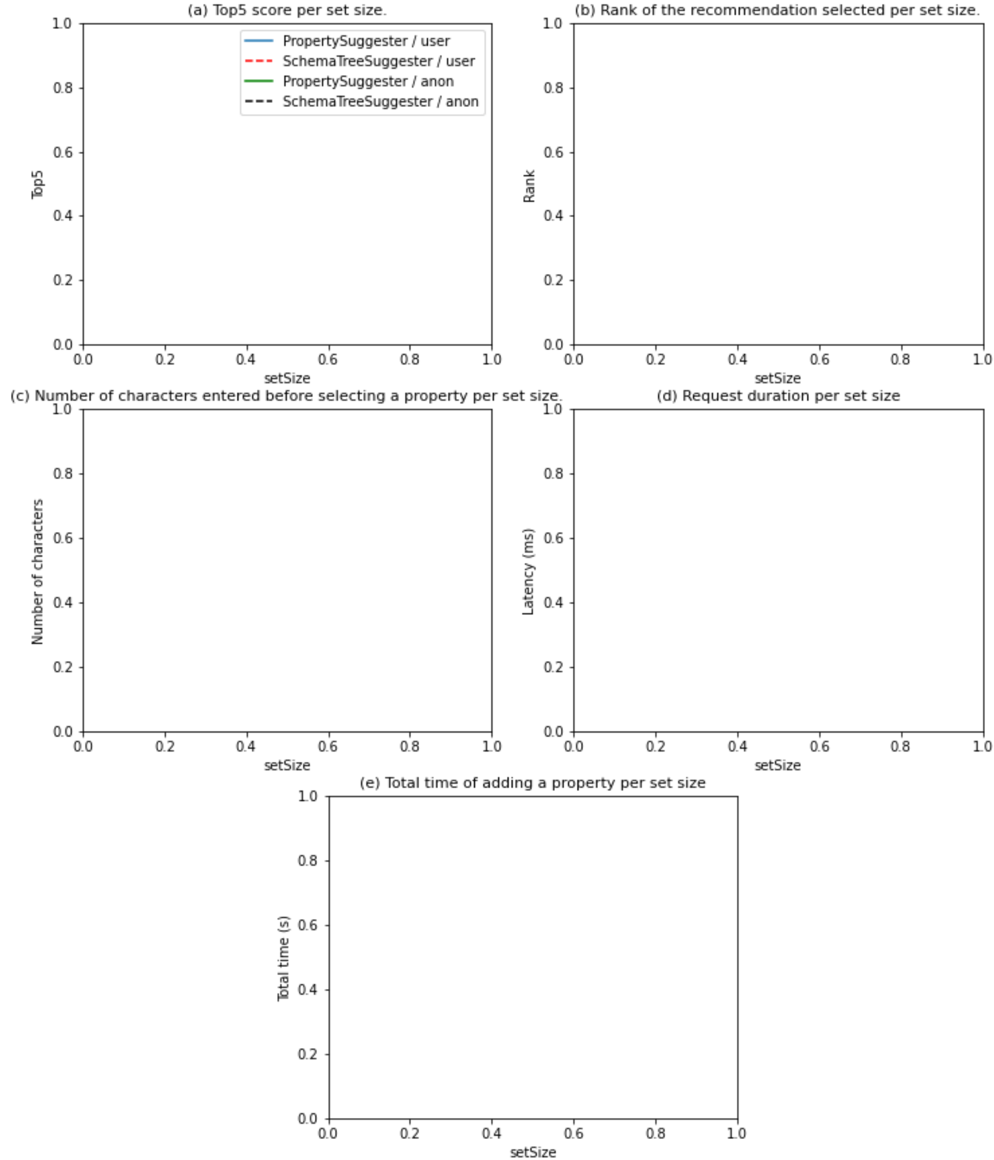


Figure 13: The average Top5 score, rank, number of characters the user entered before selecting a property, the average request duration and the total time spent editing. The results are grouped by set size and displayed separately for users who are logged in and users who are not logged in.

5 Conclusions and Future Work

In this paper the *SchemaTree* trie-based data structure previously introduced by Gleim et al. (2020) was deployed within the Wikidata environment. Leading up to the deployment process the original *SchemaTreeRecommender* code base was manipulated to decrease the request duration and the CPU usage even further. Additionally, the code was separated

into two repositories in order to assist with the deployment process to Wikidata. Within the production setting it was further evaluated against the existing association rule based *PropertySuggester*, which thus far has served the purpose of suggesting new properties to an item. The two recommendation systems were evaluated on the basis of their performance and quality in regards to the three hypotheses that support core goals of Wikidata. The *SchemaTreeRecommender* and the *PropertySuggester* were contrasted in terms of the time consumption they require when a new property is added, the overall quality of the recommendations which they serve and the support they provide to users with different expirience levels of Wikidata.

The current implementation of the *SchemaTree* is limited to only suggestions of additional properties. Therefore, further work to improve the usability of the recommendation systems of Wikidata would be to extend the *SchemaTree* to support the recommendations of qualifiers and references, which currently remain handled by the original *PropertySuggester*. The current implementation of qualifier and reference recommendations are frequency based and do not take take into consideration the qualifiers and references that are already added to the property statement. In order to adapt the *SchemaTreeRecommender* to recommend additional qualifiers to a property statement, the code base for constructing the *SchemaTree* and for serving the recommendations could be adjusted to return a ranked list of qualifiers given their maximum-likelihood.

Additionally, it is worth investigating to what extent the current implementation of the *SchemaTreeRecommender* complies to the property constraints set within Wikidata. Property constraints are the rules which denote what properties specific entities can be described by. An example of such a constraint is that the property 'head of government' (P6) should be assigned to only entities which are people. Property constraints are currently defined for over 8000 items on Wikidata, however, they remain rather guidelines and not strict rules (Ahmadi and Papotti, 2021). Therefore, conducting further analysis of how well the *SchemaTreeRecommender* complies would provide additional information on the quality of the recommendations served. Proposals of how to integrate the property constraints within the current *SchemaTreeRecommender* include adding these constraints directly within the *SchemaTreeRecommender* code base or to supply them within the API request body. Additionally, a table could be added to the Wikidata SQL database containing these property constraints, therefore, allowing for the removal of unsuitable properties within the *PropertySuggester* extension.

Moreover, the *SchemaTreeRecommender* is capable of returning types as well, however, they are not used for the purposes of adding additional properties to an item. It would be beneficial to include further investigation of how accurate the type recommendations provided by the *SchemaTreeRecommender* are. Given results of such an experiment the *SchemaTreeRecommender* could be further adapted to provide type recommendations as well.

References

Ahmadi, N. and Papotti, P. (2021). *Wikidata Logical Rules and Where to Find Them*, page 580–581. Association for Computing Machinery, New York, NY, USA.

Chilimbi, T. M., Davidson, B., and Larus, J. R. (1999). Cache-conscious structure definition. *SIGPLAN Not.*, 34(5):13–24.

Fabijan, A., Dmitriev, P., Olsson, H., and Bosch, J. (2017). The benefits of controlled experimentation at scale.

Gleim, L. C., Schimassek, R., Hüser, D., Peters, M., Krämer, C., Cochez, M., and Decker, S. (2020). Schematree: Maximum-likelihood property recommendation for wikidata. In Harth, A., Kirrane, S., Ngonga Ngomo, A.-C., Paulheim, H., Rula, A., Gentile, A. L., Haase, P., and Cochez, M., editors, *The Semantic Web*, pages 179–195, Cham. Springer International Publishing.

Gyorodi, C., Gyorodi, R., Cofeey, T., and Holban, S. (2003). Mining association rules using dynamic fp-trees. In *Proceedings of irish signals and systems conference*, pages 76–81.

Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12.

Hernández, D., Hogan, A., and Krötzsch, M. (2015). Reifying rdf: What works well with wikidata? 1457:32–47.

Vrandečić, D. (2012). Wikidata: A new platform for collaborative data collection. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, page 1063–1064, New York, NY, USA. Association for Computing Machinery.

Vrandečić, D. and Krötzsch, M. (2014). Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85.

Zangerle, E., Gassler, W., Pichl, M., Steinhauser, S., and Specht, G. (2016). An empirical evaluation of property recommender systems for wikidata and collaborative knowledge bases. In *Proceedings of the 12th International Symposium on Open Collaboration*, OpenSym '16, New York, NY, USA. Association for Computing Machinery.