Fixing Wikipedia Interlinks Using Revision History Patterns

Tova Milo
Tel Aviv University
milo@post.tau.ac.il

Slava Novgorodov
eBay Research
snovgorodov@ebay.com

Kathy Razmadze
Tel Aviv University
kathyr@mail.tau.ac.il

ABSTRACT

Wikipedia, the web-based free content encyclopedia project, is one of the most popular websites on the Web. Its “open-door" policy, allowing anyone to edit, has made Wikipedia the largest and possibly the best encyclopedia in the world. At the same time, the continuously evolving content, constantly updated by a large number of uncoordinated users, renders the maintenance of a clean, consistent encyclopedia an extremely challenging task.

The goal of the WICLEAN (WC) system presented in this paper is to assist Wikipedia editors in this difficult task. Specifically, we focus on the correctness of Wikipedia *inter-links* that point from one article (entity) to another. Such inter-links form a key component of the structured part of Wikipedia and their correctness is critical for coherent browsing. Given an entity type of interest, our highly parallelizable algorithm identifies relevant edit patterns across revision histories of Wikipedia entities of related types, along with time windows in which partial edits are tolerable. The discovered patterns/windows are then used by WC to alert Wikipedia editors on past edits that appear to be incomplete, as well as to provide users with on-line assistance as they update the encyclopedia. Our experiments with real-life Wikipedia data demonstrate the efficiency and effectiveness of WC in identifying actual errors in a variety of Wikipedia entity types.

1 INTRODUCTION

Wikipedia, the free-content web encyclopedia, is one of the most popular websites on the Web. Per Time magazine, Wikipedia’s "open-door" policy of allowing anyone to edit the data, has made it the largest, and possibly best, encyclopedia in the world [2]. Nonetheless, the continuously evolving content, constantly updated by a large number of uncoordinated users, renders the maintenance of a clean, consistent encyclopedia an extremely challenging task. To understand the volume of the updates, the English Wikipedia in 2018 consisted of 6 million articles, with an average of 3.4 million edits per month, by roughly 30*K* active editors [4].

The goal of our work is to assist Wikipedia editors in this difficult task. Specifically, we focus here on the correctness of *inter-links* that point from one article to another in the *structured* sections of Wikipedia (such as infoboxes and tables), which is critical for coherent browsing. Maintaining the integrity of these links is challenging, as illustrated by the following example.

Example 1.1. Consider the Wikipedia page of the soccer player Neymar. The links in its infobox point to the page of his current club, Paris Saint Germain F.C. (PSG), his place of birth, and so on. When Neymar moved to PSG in 2017, leaving his previous team, Barcelona F.C., the three related pages, *Neymar*, *PSG*, and *Barcelona F.C.* had to be updated.

There are three typical causes for inconsistently updating these links. First, Wikipedia editors are not provided with a comprehensive list of links that need to be updated as a result of such an event. A typical error related to player transfers is updating only the page of the new club and neglecting to update the page of the old club, which still incorrectly links to the player.

Second, different pages are often edited by different people, typically, in an uncoordinated manner. It could be that, e.g., the page of the club is updated by one dedicated editor, whereas no editor has taken up the responsibility of updating Neymar’s page or even noticed the absence of a corrected link. Moreover, no mechanism alerts the active editors of Neymar’s page of a related update, that may require action on their part.

Third, it is often impractical to correct all links simultaneously. For example, player transfers occur during predetermined periods, referred to as transfer windows, and tend to take a long time to be officially confirmed. In the meantime, many rumors regarding conflicting transfer destinations are posted in various media outlets. Consequently, in that span, there may be hundreds of edits of player pages, adding/removing new/old links, and reverting previous edits, whereas the club pages are commonly updated only once the transfer is officially approved.

More generally, Wikipedia contains very noisy data, as it could be edited by anyone, including bots[1], inexperienced editors, and opposite-agenda editors[2], resulting in editing conflicts[3] and dispute resolution[4]. This process of frequent conflicting edits, culminating in a consistent state, is a naturally evolving mechanism to mitigate noise, due to the distributed and asynchronous nature of Wikipedia edits. Thus, enforcing immediate corresponding updates to all relevant links during the dispute period is impractical and counterproductive. Moreover, the existence of a time window, that may range from hours to months (depending on the context of the update and the involved entities), during which partial inconsistent edits are tolerable, beyond serving as a necessary trigger for the dispute resolution process, also has the advantage of providing users with the most up-to-date, albeit tentative, information.

Previous work. Much research has been devoted to aspects of this problem in the more general context of detecting errors in *knowledge bases* (KBs) [22]. Some of these works [27, 30] also evaluated their solutions over Wikipedia, representing a snapshot of it as a KB, with pages as entities, and entity relations derived from inter-links. Over this representation and an input set of *integrity constraints*, pertaining to entity relations, the objective is to detect all their violations. While these works provide satisfactory solutions for the intended problem over KBs, casting the special case of inconsistencies in the constantly-evolving Wikipedia’s links into this generic framework, omits important practical considerations specific to the operation of Wikipedia.

To illustrate, continuing with our example of player transfers, a possible constraint over the corresponding KB may state that if player A links to club B, then club B also links to player A and vice versa. If there exists only one link or two contradictory links, then a violation of this constraint is detected. There are several drawbacks to applying this approach as a comprehensive solution.

[1] https://www.bbc.com/news/magazine-18892510
[2] https://en.wikipedia.org/wiki/Wikipedia:Lamest_edit_wars
[3] https://en.wikipedia.org/wiki/Edit_conflict
[4] https://en.wikipedia.org/wiki/Wikipedia:Dispute_resolution

First, the most crucial drawback relates to the fact that the constraints are static and lack any temporal dimension. Concretely, the constraint does not account for the time window, discussed in Example 1.1, during which partial edits are acceptable (and, in some cases, practically unavoidable). An inconsistency should be resolved at the earliest appropriate moment but not earlier. In this case, this earliest moment is arguably the end of the transfer window. However, detecting the constraint violation right after Neymar's page is linked to PSG, without a link in the other direction, is treated the same as detecting it long after the end of the transfer window. Consequently, one is uncertain whether to take immediate action or allow the unsupervised process of sequential refining edits to run its course and converge into a consistent state. We note that solutions suggested in previous works (e.g., [27]), discussed above, can successfully identify 'window-less' edits' combinations. That is edits that should all be applied simultaneously and are distributed uniformly across the timeline of Wikipedia's revision history. However, our empirical analysis identified many edit patterns associated with specific time windows, such as in Example 1.1. Our work is, thus, complementary to the above works, as it aims to address specifically these patterns.

Second, most works assume the existence of a set of constraints as input for the solution framework. This is not realistic in the case of Wikipedia. Wikipedia entries encompass a wide array of domains and sub-domains, each with its own set of constraints. While there are broad similarities across related domains, each domain may be infinitely nuanced. Given the volume of domains, entity types, and case-specific subtleties, the task of comprising a nearly exhaustive list of important constraints is impractical, particularly if one is also interested in complex relations (where, e.g., a combination of 10 pages must be consistently updated).

In this line of research, closest to us, is the recent work of [36], where the focus is on Wikipedia, and on top of detecting violations of the given constraints, the solution produces corresponding *correction rules*, that dictate how one can resolve partial edits. This is inferred by examining the revision history, identifying the most common patterns of revision actions for completing each type of partial editing. Nevertheless, this work also does not aim to identify tolerable time windows and targets the scenario where the list of constraints is provided as input (a more detailed comparison to this and previous works is presented in Section 2).

Our approach. To address the above limitations, we present in this paper WiClean (`WC`), a system that automatically infers common edit patterns (combinations of edits), along with a time window for allowing partial edits of each pattern, alerts editors of inconsistencies, and suggests concrete corrections.

The thesis underlying `WC` is that the majority of Wikipedia updates follow desirable patterns and lead to consistent states. `WC`, thus, mines revision logs to identify common update patterns and the time windows in which they occur. Potential errors are then detected by updates that deviate from the patterns and are not completed within the corresponding window. For such partial patterns, `WC` suggests all completions to known patterns, providing statistical metadata to facilitate an informed course of action.

Before describing our solution techniques, we illustrate the format of the revision history. Figure 1 depicts excerpts from the revision histories of players and clubs merged into a single timeline. The Subject column identifies the article where the addition/removal of a link occurred, the Object column identifies the article to which the added/deleted links point, and the Relation column describes the link type (the column R will be explained later). One can see that, after several edits and reverts, the transfer of Neymar is reflected in his and the teams' pages.

#	+/-	Subject	Relation	Object	Time	R
1	-	Neymar	current_club	Barcelona F.C	...1531	0
2	-	Gianluigi Buffon	current_club	Juventus F.C.	...1534	1
3	-	Neymar	in_league	La Liga	...8711	1
4	-	Barcelona F.C	squad	Neymar	...2804	0
5	+	Neymar	current_club	PSG F.C.	...3321	0
6	+	PSG F.C.	squad	Neymar	...8263	1
7	+	Barcelona F.C.	squad	Neymar	...4040	0
8	+	PSG F.C.	squad	Gianluigi Buffon	...4051	1
9	+	Gianluigi Buffon	in_league	Ligue 1	...3330	1
10	+	Neymar	in_league	Ligue 1	...8711	1
11	-	Juventus F.C.	squad	Gianluigi Buffon	...4058	1
12	+	Neymar	current_club	Barcelona F.C	...5861	0
13	-	Kylian Mbappe	current_club	Monaco F.C.	...9459	1
14	-	Neymar	current_club	PSG F.C.	...3732	0
15	-	Gianluigi Buffon	in_league	Serie A	...3380	1
16	-	Neymar	current_club	Barcelona F.C	...6109	1
17	+	Neymar	current_club	PSG F.C	...7694	1
18	-	Barcelona F.C.	squad	Neymar	...8001	1
19	+	Kylian Mbappe	current_club	PSG F.C.	...9589	1
20	-	Monaco F.C.	squad	Kylian Mbappe	...9451	1
21	+	PSG F.C.	squad	Kylian Mbappe	...9885	1

Figure 1: Actions from revision history of several articles

Methods. Formally, we model Wikipedia entities (articles) and the links between them as a graph. Nodes and edges are labeled by type names. Intuitively, the revision history of each article records the edits made to the outgoing links of the corresponding graph node. Given an entity type of interest, our algorithm identifies meaningful relevant edit patterns across revision histories, along with time windows in which partial edits are acceptable. By making an analogy between link edits (resp. edit patterns) and graphs (graph patterns), we can harness conventional graph mining algorithms to our context. However, some important adaptations must be made to account for (1) the Wikipedia type hierarchy[5] that requires the examination of a larger number of potential patterns, and (2) the distributed nature of the edits across revision histories of multiple entities, that makes the construction of the full edits graph prohibitively expensive. For the former we introduce a join-based computation (optimized by the underlying SQL engine) to quickly prune infrequent patterns; for the latter we use incremental graph construction that considers only relevant entity types.

The discovered windows and patterns are then used by WC to assist Wikipedia editors in correcting/updating Wikipedia links. Here again, we employ an optimized join-based computation to quickly identify potential errors. `WC` both alerts Wikipedia editors on past edits that appear to be incomplete as well as provides users with on-line assistance as they update Wikipedia.

Our contributions can be summarized as follows:

- **Model.** We formulate and present a simple, natural model for capturing time windows and update patterns of interest. Given an entity type t, our goal is to find related and common update patterns across the Wikipedia graph. Such updates may involve entities of the same or other types. We first introduce the notion of *abstract* update actions that generalize a set of actions involving specific entities to general patterns over the corresponding entity types. We then define the notion of *connected patterns* which include abstract actions that are related (possibly transitively) to entities of the input type of interest. The *frequency* of a pattern, within a time frame w, is then naturally defined as the fraction of entities of type t that participate in a pattern that occurs within the time frame w (Section 3).
- **Identifying windows and patterns.** Building on algorithms for graph mining, we devise a scalable, highly parallelizable algorithm, based on the following three points. (1) We represent

[5]Typically around eight hierarchy levels.

the identified patterns by relational tables, incrementally computed by dedicated relational queries. This allows harnessing the effective optimizations of the SQL engine underlying WC. (2) Unlike conventional graph mining algorithms that assume that the entire graph is given as input, our focus on connected patterns allows WC to incrementally consider only the entity types (and their corresponding revision histories) that may potentially be related to the input type via frequent edit patterns, thereby significantly saving on graph construction. (3) We focus on non-overlapping time windows and split the revision histories accordingly. This reduces the number of actions (edits) to be considered for each window (and resp. the size of the edits graph) and allows parallelizing the processing of the action sets in the different windows (Section 4).

- **Using Windows and patterns.** An immediate application of the discovered patterns is to alert Wikipedia editors on partial edits performed in past windows, as well as to assist users in current edits. For that, we examine the discovered windows and then signal, for each window and pattern, partial edits that may be extended to a full pattern occurrence. Our algorithm builds on the previously mentioned relational representation of patterns and employs dedicated outer-join queries to identify partial pattern occurrences (Section 5).
- **Implementation and experiments.** We have implemented our solution and employed it over real Wikipedia data. We considered a variety of Wikipedia entities, identifying a multitude of interesting time frames and corresponding relevant frequent edit patterns, and signals of updates that deviate from the mined patterns. Our experiments demonstrate the effectiveness of our approach for identifying real-life errors. The experiments further demonstrate the efficiency and scalability of our algorithms, compared to competing baselines (Section 6).

To complete the outline of the paper, we overview related work in Section 2 and discuss future work in Section 7.

Finally, we note that the prototype of WC was demonstrated in [20]. The short paper accompanying the demonstration provided only a high-level overview of its capabilities and user interface whereas the present paper details the model and algorithms underlying our solution as well as their experimental evaluation.

2 RELATED WORK

We overview related work from several related fields.

Wikipedia Cleaning. Much effort has been devoted over the past years to the cleaning and correction of errors in Wikipedia. Our work, which focuses on link correction, is complementary to works on entity resolution, completeness prediction, and vandalism detection [9, 33]. Similarly to our work, [13] also aims to improve inconsistencies in Wikipedia's infoboxes, representing it as an RDF database. However, [13] does not take the revision history into account and instead uses user interaction as the main tool. In contrast, our algorithm requires no user assistance, other than setting the initial parameters.

Revision history as a tool. Revision histories have been used in multiple areas, e.g., in program repairing, in recording provenance in knowledge bases and assisting query answering [10]. In Wikipedia, revision histories have been leveraged for various purposes, such as the discovery of controversial topics, the estimation of an article's translation quality and the detection of vandalism [23]. Other lines of work attempt to learn how to use the edits to enrich Wikipedia, e.g. to edit infoboxes with news extracted from tweets, or to connect Wikipedia edits to recent news articles [17]. Our work is complementary to these efforts, considering the consistency/completeness of edits to multiple related entities.

As mentioned in the introduction, particularly close to our work is [36], which, similarly to WC, infers from edit histories in Wikidata knowledge bases how to correct inconsistencies/violations. Nevertheless, this problem is formalized over a different model with similar but different objectives. One important difference is that in the setting of [36], *the constraints are provided in advance*, and the focus is on correction rules (for violations of these constraints) mining from relevant past edits. Whereas, one of the key contributions of our work is the derivation of such constraints (edit patterns, in our context). Moreover, the setting of [36] does not take into account the time frames in which a given constraint should or should not be enforced. Another key difference is that [36] do not harness the Wikipedia type hierarchy as a means to enrich their constraints of correction rules.

Other works that use the Infobox revision history focus on cleaning tasks. These include refining infobox titles by locating duplicate attributes within each entity type, predicting when a given infobox is likely to be updated and by whom, and identification of vandalizing editors [8].

Constraints inference and enforcement. The patterns we identify can be viewed as a form of *integrity constraints*. There is a large body of work devoted to inferring and enforcing such constraints. Two recent examples are [27, 30]. In [30], both positive and negative examples are used to infer the constraints. Their approach consists of greedily identifying, at each step, the most promising rule, in terms of the coverage of the positive examples. As [30] focus on identifying rules that make good predictions, some of the rules that exceed the confidence threshold will not be found. An alternative approach is taken by [27], where rules are mined via an exhaustive, breadth-first search method. They devise sophisticated pruning strategies and optimizations that enable their solution to efficiently run on large KBs, such as Wikidata.

Many other approaches to KB correction have been explored in the literature, e.g., discovering *denial constraints* [14] and error detection via the few-shot learning framework (e.g., [22]).

A key difference, in our setting, is that the constraints need to be enforced only outside the time frames in which inconsistencies are acceptable. Thus, we focus on a different objective, where in addition to the update patterns, we also identify the corresponding time window for each pattern. Another difference is that we identify patterns from the sequence of actions in the revision log, and not from a static snapshot of the knowledge base. Moreover, the above works (barring [36], which we discussed separately above), are concerned with detecting the rules/constraints, while one additional objective, in our setting, is to also compute correction suggestions for violations (partial patterns) of these rules. Lastly, to our knowledge, we are the first to leverage the type hierarchy to consider more nuanced rules, at varying levels of abstraction.

The importance of considering consistency, w.r.t. a sequence of actions, has recently been emphasized in the vision paper of [12]. Our work matches their motivating use-case, which advocates the usage of Wikipedia revision logs for data cleaning. Another related, complementary line of work deals with optimizing the corrections procedure over the detected constraint violations [11, 19]. It would be interesting to examine whether their techniques may be employed in our setting, to further optimize WC.

(Sequential) itemset/association rule mining. Algorithms for frequent itemset/association rules mining have been the focus of many works, including contexts where the mined items belong to a type hierarchy [32].As we seek connected patterns, conventional

a-priori style algorithms for frequent itemsets mining [7] inapplicable to our setting. Such algorithms recursively assemble larger frequent itemsets from smaller ones, but arbitrary sub-patterns of a connected pattern may not be connected, w.r.t. the input type. Consequently, our solution exploits principles from graph mining algorithms rather than general frequent itemsets.

Another closely related line of work deals with *sequential* itemset/association rules mining, where the pattern is mined from a sequence of items [34, 41] and [5] which discovers temporal rules for web data cleaning. In these works, the focus is also typically on arbitrary items set, rather than connected patterns. More importantly, the order of the items in the sequence is important in these works. In contrast, as explained in Section 3, in our case, only the co-occurrence of items within the given window matters, whereas their relative positioning within the window does not.

An interesting set of works that deal with sequential patterns mining studies probabilistic or uncertain databases [18, 29]. In our setting, there is inherent uncertainly, w.r.t. the (in)correctness of the identified partial updates, and thus examining the connection to such works is an interesting direction for future research.

Graph mining. Graph mining algorithms (see survey in [24]) can be roughly divided into two categories: algorithms that mine patterns in a set of graphs (e.g. [39]) and algorithms that are provided with a single large graph (e.g. [26, 28]). Our context is the latter. Multiple notions of graph pattern frequency have been proposed in the literature, many of which consider the number of distinct isomorphisms from the given pattern graph to the input graph [15, 24]. However, as our goal is to characterize how frequent a pattern is *relative to a particular entity type of interest*, we employ here the notion of frequency, inspired by [16], that counts the number of nodes, out of all nodes of the given type, that are involved in some pattern occurrence. Our notion can also be viewed as a special case of the MNI support in [15], where the isomorphism count focuses only on the given entity type.

As discussed above for association rules mining, since our focus is on connected patterns (and the corresponding frequency notion), algorithms that consider arbitrary sub-graphs (e.g. [21]) are unsuitable for our setting. We follow instead the "grow and store" approach of [26], that iteratively expands previously identified (connected) patterns. However, two issues must be addressed when adapting such a scheme to our context. First, the need to support the Wikipedia type hierarchy entails a richer order relation among patterns (see Section 3), which, to our knowledge, is not supported by any of the existing algorithms for mining connected patterns in graphs. Second, [26] (and all other comparable works), assume that the algorithm receives as input the entire graph. This is impractical in our context. Specifically, as our experiments demonstrate, materializing the complete edits graph from a massive number of entity revision histories is infeasible. Our dedicated algorithm addresses both these issues. In general, modifying solutions that expect the entire graph as input, is, arguably, not trivial. For instance, the work of [40], which leverages the embedding of the nodes to mine patterns, cannot be straightforwardly integrated into our approach of gradually examining larger subgraphs, as the embedding loses its utility if the underlying graph changes.

Another related line of work is Link Prediction [35, 38] that discover missing links within Wikipedia. However, these works do not detect incorrect links that should be removed.

Wikipedia information extraction. To conclude, we note that one may think of the patterns/time windows that we derive as a particular type of information, extracted from Wikipedia revision logs. Much previous work has been devoted to information extraction from Wikipedia *articles* (e.g. [31]) rather than their edit history. As mentioned, some works consider the revision logs, but for other purposes, and could be useful information or tool for us. In particular, [37] devise optimization methods for processing Wikipedia's revision history, as it is a massive and complicated dataset, and [25] infers the level of expertise of a specific editor from statistics of conflicts with other editors.

3 PRELIMINARIES

We start by presenting the data model underlying the system.

Wikipedia Graph. We model the relations between entities at a given point in time using a graph $G(V, E)$. Each node represents an entity and is labeled by a unique name (e.g. Neymar) and a type (e.g. *soccer player*). Each edge represents a relationship between two entities and is labeled accordingly (e.g. $current\_club$).

We use an alignment from Wikipedia entities to DBPedia [1] to derive the entity types. The link labels (relationship names) are derived directly from Wikipedia. In general, the types belong to type taxonomy - the higher the type is in the taxonomy the more general it is - and an entity may have multiple types. For two types t, t' we use $t' \leq t$ to denote the fact that t either equals to t' or generalizes it. For example, $Soccer\_Player \leq Athlete \leq Person$. We assume that each entity e has one most specific type to which it belongs and use it as its label, denoted $type(e)$. For a type t we use $entities(t)$ to refer to all entities labeled by a type $t' \leq t$.

Actions and inverse actions. The revision history of Wikipedia entities contains edits to the graph edges. We particularly consider two types of actions: adding *new* edges and deleting *existing* ones. Our model associates each action with a time stamp. We use a triplet of the form $a = (+, (u, l, v), t)$ (resp. $a = (-, (u, l, v), t)$) to denote the addition (rep. deletion) of edge from u to v with label l at time t. We use $source(a) = u$ and $target(a) = v$ to denote the source and target entities, resp., of the added/deleted edge. We say that an action a' is the *inverse* of a preceding action a, denoted $a' = Inv(a)$ if applying a' after a leaves the graph unchanged.

For example, in row #1 in Figure 1 we see an update to Neymar's Wikipedia entry, when a user removed $(-)$ the Barcelona $(= v)$ team that Neymar $(= u)$ was playing at $(= l)$, at a certain time (=t), and, action #12 is an inverse action of action #1.

Note that, in Wikipedia, each action appears at the revision history of the *source* node of the edge. Intuitively, this is because the revision history of each article records the edits made to the outgoing links of the corresponding graph node. Updates of other incoming links are recorded in the revision logs of these other pointing entities. Continuing the above example, the two actions #1 and #3 will appear in the revision history of Neymar's page, and the gray action set in Figure 1 is the set of actions taken from entities of the same type (soccer_player).

(Reduced) set of actions. Given a Wikipedia graph $G(V, E)$, a set of entities $S \subseteq V$, and a time frame (referred to as *window*), we consider the set of all actions (denoted as A) that were recorded in the revision history of the entities in S, within the given window.

For instance, Figure 1 shows the set of actions recorded in the revision histories of the entities S = *{Neymar, Kyian_Mbappe, Barcelona_F.C., Gianluigi_Buffon, PSG_F.C., Monaco_F.C., Juvetus_F.C.}* at a given time frame. Observe that all the updated links are *outgoing* links from the entities in S.

In the update processes, some edits may naturally be reversed. To consider only the final effect we focus on *reduced actions sets* that do not include action and its inverse. More formally, given a graph G, we say that two action sets are *equivalent* if, when the

actions are applied on G in the order of their timestamps, they yield the same graph. The reduced set of actions, that remain by removing the rows that their value in column R equals to 0 in the table of Figure 1. We denote it as reduced actions from Figure 1.

Note that up to possibly different timestamps, the reduced version obtained through this iterative removal process is unique, as it contains the same set of graph update operations. Furthermore, the timestamps are no longer important as any permutation of the actions yields the same output graph. We thus consider from now on only reduced sets of actions and ignore the timestamps, referring to actions as pairs $a = (op, (u, l, v))$ where $op \in \{+, -\}$.

Abstract actions. Since we are trying to find general update patterns across the Wikipedia graph, we want to generalize a set of actions involving specific entities to general patterns over the corresponding entity types. For that we define the notion of *abstract actions*. We associate with each entity type t an infinite set of variables $t_1, t_2, \ldots$. Then, an *abstract action* is the pair of the form $a = (op, (t', l, t''))$ where $op \in \{+, -\}$, t' and t'' are type variables, l is an edge label.

Patterns. We define a *pattern* as a set of abstract actions. We consider two patterns identical if they are the same up to isomorphism on the variable names of the same type. We refer to a pattern that contains only a single action as a *singleton* pattern. Given a pattern p we say that a set A' of concrete actions is a *realization* of p (resp. that p is an abstraction of A') if A' may be obtained from p by replacing each variable of type t by a some Wikipedia graph node in $entities(t)$, s.t. distinct variables are assigned different Wikipedia graph nodes.

An observation that will be useful in the sequel is that for a given action a, the set of its possible abstractions can be easily computed by traversing the type hierarchy and replacing $source(a)$ (resp. $target(a)$) by some variable of type $\geq type(source(a))$ ($\geq type(target(a))$).

To illustrate the above notions, in the reduced actions in Figure 1 lines #2 and #13 are both realization of the singleton pattern {-, (player1, current_club, team1)} (which we consider identical, e.g., to the isomorphic pattern {-, (player2, current_club, team2)}). On the other hand, the reduced actions in Figure 1 contain no realization of the pattern
$[\{-, (player1, current\_club, team1)\},$
$\{-, (player1, current\_club, team2)\}]$
as the assigned team nodes have to be distinct in the realization, but all players in the table were removed from a single team.

(Abstract) actions graph. It is useful to make an analogy between action sets and graphs. Given a set of concrete (resp. abstract) actions A (p), consider the directed labeled graph g_A (g_p) with a node per each entity in A (variable in p), labeled by the entity (variable) type name, and where there exists an edge from node v_1 to v_2, labeled $[op, l]$, iff A (p) includes an (abstract) action of the form $(op, (v_1, l, v_2))$. We refer to these graphs as abstract graphs as the actual entity identities (resp. the variable names) that the nodes represent are insignificant.

With this graph view, a realization of a pattern p in a set of actions A corresponds to an isomorphism from g_p to a subgraph of g_A, where the type of each node in p either equals or is more general than the type of its corresponding node in g_A. Given a type t, we say that the pattern p is **connected** (w.r.t. t) iff g_p contains a node variable of type t from which all other nodes are reachable.

Connected patterns. Given an entity type t, we are interested in entities' updates that are related (possibly transitively) to entities of type t. We thus focus on *connected* patterns, where the updated edges are related.

Definition 3.1. For an update pattern p, let g_p be its corresponding abstract graph. Given a type t, we say that the pattern p is **connected** (w.r.t. t) iff g_p contains a node of type t from which all other nodes are reachable.

In the discussion below we refer to such a node (variable) as the pattern's **source** (w.r.t. t). If multiple such nodes exist in the graph, we arbitrarily pick one to serve as the pattern's distinguished source and we use below the term source to refer to this single distinguished node, and denote is as $source_t$.

For example, the pattern shown in Figure 3 is connected w.r.t. to the type $player$. Its corresponding graph g_p appears in Figure 2(a) where all nodes are reachable from the source node $player\_1$. But if we replace the variable $player_1$ in lines 11 and 13 of Figure 3 by a new variable $player_2$, then the pattern becomes disconnected, see Figure 2(b), and composed of two smaller, connected patterns - the abstract actions in lines 10, 5, 2, 7 (with source $player\_1$) and the abstract actions in lines 11, 13 (with source $player\_2$).

For a type t we only consider patterns that are connected w.r.t. t. Thus, for brevity, we use below the term *pattern* to refer to a connected pattern, and omit the type t when clear from the context.

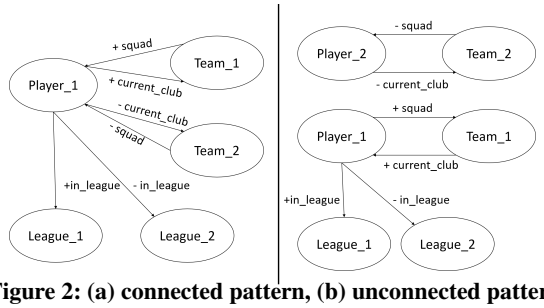

Figure 2: (a) connected pattern, (b) unconnected pattern

Frequent patterns. To define that a pattern is frequent we would like to measure the amount of support that a pattern has, regarding the seed type entities.

Many notions of patterns frequency have been considered in the graph mining literature. Common notions consider the number of distinct isomorphisms from the given pattern graph to the input graph (e.g. occurrence-based support [24] and MNI support [15]. However, as our goal is to characterize how frequent a pattern is *in the context a particular seed type of interest*, we employ here a notion of frequency inspired by the [16] that counts the number of nodes (out of all nodes of the given seed type) that are involved in some pattern. For readers familiar with the *MNI*-based support in [15], we note that our notion of frequency can also be viewed as a special case where the isomorphism count focuses only on the seed entity type node.

Given a type t, a pattern p and a set A of actions, we define the *frequency* of p (w.r.t. to t and A) as the fraction of entities of t that participate as source nodes in a realization of p is A.

Definition 3.2. The *frequency* of a pattern p in a set of actions A, w.r.t. to a type t in p, is defined as $frequency(p, A, t) =$

$$\frac{|\{e \in entities(t) \mid e \text{ appears in realization } entities(t)|}{|entities(t)|}$$

To continue with our running example, consider the actions in Figure 1 and the pattern in Figure 3, and assume there are overall five players in Wikipedia. The frequency of this pattern in the given actions set, w.r.t. to the type $player$, is 0.2 because there is only one player (Neymar) that the patterns hold for (with Neymar mapped to $player\_1$), out of the five existing players. However, the frequency of the partial pattern displayed in figure 3 in lines 1

#	Edit type	Subject	Relation	Object
10	+	$player_1$	current_club	$team_1$
11	-	$player_1$	current_club	$team_2$
5	+	$team_1$	squad	$player_1$
13	-	$team_2$	squad	$player_1$
2	+	$player_1$	in_league	$league_1$
7	-	$player_1$	in_league	$league_2$

Figure 3: Pattern found from set of action in Figure 1

and 2 (gray lines) in this actions set (again w.r.t. the type *player*), is 0.4 because there are 2 players for which that pattern holds.

Partial Order of Patterns. Given a type t, a set A of actions and frequency threshold τ we will be interested in finding patterns whose frequency in A (w.r.t. the given type) is above the threshold. To avoid redundancy, we would like to consider only the most *specific* such patterns. Formally, we say that a pattern p is more specific than a pattern p' (alternatively, p' is more general than p), denoted $p \prec p'$, if p' may be obtained from p by removing some abstract actions, replacing some type variables in p by corresponding variables of a more general type, or both. An alternative definition is close frequent sub-graph, as defined in [39]. To illustrate, for the patterns:

$p_1 = \{(+, (player_1, current\_club, team_1)),$
$(-, (player_1, current\_club, team_2))\}$
$p_2 = \{(+, (athlete_1, current\_club, team_1)),$
$(-, (athlete_1, current\_club, team_2))\}$
$p_3 = \{(+, (athlete_1, current\_club, team_1))\}$

we have that $p_1 \prec p_2 \prec p_3$.

Thus, given a type t and a set A of actions our goal will be to find the most specific patterns with a frequency above a given threshold. Our formal definition refines the closed frequent graph pattern notion of [39], taking the type hierarchy also into consideration when ordering patterns.

Definition 3.3. Given a set of actions A, a type t, and a frequency threshold τ, we say that a pattern p is a *most specific frequent pattern* in A (w.r.t. t and τ), if $frequency(p, A, t) \geq \tau$ and there is no pattern $\hat{p} \prec p$ where $frequency(\hat{p}, A, t_i) \geq \tau$.

Relatively frequent patterns. Finally, note that in the discussion so far, the frequency of patterns p was measured w.r.t. a given type, as the percentage of entities of the given type that serve as a pattern source. In some cases, it is interesting to further explore what percentage of these entities adhere to a more specific pattern p'. For example, what percentage of players among the ones that move to a new team also, change the league. For that we define the notions of *relative* frequency and *relative* frequent patterns.

Definition 3.4. For two patterns p, p' s.t. $p' \prec p$, the *relative frequency* of p' w.r.t. p in a set of actions A (for a given type variable t), is defined as

$rel\_frequency(p', p, A, t) = \frac{frequency(p', A, t)}{frequency(p, A, t)}$.

Definition 3.5. Given a set of actions A, a type t, a pattern p and a relative frequency threshold τ_{rel}, we say that a pattern p' is a *most specific relative frequent pattern* in A, w.r.t. t and p, if $rel\_frequency(p', p, A, t) \geq \tau_{rel}$ and there is no more specific pattern $\hat{p} \prec p'$ where $frequency(\hat{p}, p, A, t) \geq \tau$.

4 FINDING WINDOWS AND PATTERNS

Intuitively, given an entity type t of interest, we wish to signal out significant time frames and identify the most specific frequent patterns in them.

We will first explain how, given a specific window w and frequency threshold τ, the most specific frequent patterns in w (w.r.t. type t), are efficiently identified. The extraction of relative frequent patterns is similar. Finally, we will explain how the windows and thresholds to examine are selected.

As noted in Section 3, the (reduced) set of edit actions performed over Wikipedia entities in the time window w may be viewed as a graph. Thus one may harness graph mining algorithms, such as the ones presented in [24] to identify frequent connected patterns. Such algorithms work roughly as follows. Starting from patterns consisting of a single edge, they incrementally expend the patterns with new edges. At each iteration, they check which of the extended obtained patterns are frequent, prune all the others, and iteratively continue expending the frequent ones.

There are, however, two important issues that one has to address when adapting such a scheme to our context.

1. Supporting the Wikipedia type hierarchy entails a richer order relation among patterns (as defined in Section 3), which to our knowledge is not supported by any of the existing algorithms for mining connected patterns in graphs. While the modifications to the algorithms, to support this, are rather immediate, the number of patterns that now need to be examined becomes larger, and thus the patterns' frequency test must be performed efficiently. For that, we represent each graph's type of relation as a relational table, containing its pattern realizations. That allows us to utilize a join-based computation (optimized by the underlying SQL engine) to quickly prune infrequent patterns.

2. Observe that common graph mining algorithms assume that a full graph is given as input to the algorithm. In our setting, the revision histories are distributed across all Wikipedia entities, and (even when restricted to the time window w) their overall size can be very large. Thus, as our experiments show, materializing the full graph that represents them may be prohibitively expensive. To avoid this, we embed into the discovery of the incremental patterns an analogous incremental graph construction, that materializes only revision histories of entity types that may potentially be related to the input type t via frequent edit patterns. Our pattern mining algorithms is detailed in Algorithm 1. For better understanding the pseudo-code, we first outline the data structures and notations that we make use of. For space constraints, an illustrative example appears in our technical report [3].

4.1 Data Structures and Notation

For each considered time window w, the algorithm incrementally extracts, from the revision histories determined to be relevant, the set of actions performed within the time frame. The actions are abstracted and stored in a dictionary called $abstract\_actions$ whose keys are the time windows. Thus, $abstract\_actions[w]$ denotes a set of abstract actions with realizations in the window w. The corresponding realizations of each such abstract action are stored in a dictionary called $realizations$ whose keys are the time frame and abstract action. Thus, $realizations[w][a]$ denotes the set of realizations of abstract action a within window w.

The identified (relative) patterns, for each time window w (and pattern p in w), are stored in a dictionary named $patterns$ (resp. $rel\_patterns$) whose keys, again, are the time frames (and related patterns). Thus $patterns[w]$ (resp. $rel\_patterns[w][p]$) denotes the set of (relative) patterns computed for time window w (and pattern p in w). We overload notation and also use below $realizations[w][p]$ (resp. $realizations[w][p][p']$ to denote the realizations of the (relative) pattern p (p') within time window w. As mentioned above, the pattern realizations are implemented as relational tables. We will explain this point in details below. Finally, we use an auxiliary data structure $tested[w]$, whose keys are the time frames, to record partial patterns that have already been examined in the computation for the window w.

4.2 Pattern mining

We are now ready to present the Algorithm 1. As mentioned above, the algorithm follows the line of graph mining algorithms such as [15], starting from singleton patterns and incrementally expending them. While doing so it incorporates into the processing the two optimizations mentioned above, to ensure efficient processing in our particular setting. We note that several additional optimization techniques have been introduced in [24, 39], e.g. to minimize the used storage and search space. These are orthogonal to ours and thus, for simplicity of presentation, we follow below the basic scheme of the incremental pattern construction (to which these orthogonal optimizations can later be applied if desired).

Initialization. Our initial entity set S contains the entities of input type t. First, we extract for the given window w edit actions performed on entities in S in time window w. We reduce the set of actions, eliminating redundant edits and computing the possible action abstractions (as explained in Section 3) and store them in $abstract\_actions[w]$ and their corresponding realizations in $realization\_table[w]$. This is performed using the function $reduced\_and\_abstract\_actions(S, w)$ (line 1). $patterns[w]$ stores only abstract actions (singleton patterns) whose source is the seed type t and their frequency in w exceeds the threshold (line 2). We explain below how the frequency is efficiently computed.

Interleaving graph and patterns expansion. We next interleave the extension of considered entity set (and, correspondingly, the considered subgraph representing their respective revision histories), with the extension of the patterns.

To determine which other related entities (and, respectively, entity revision histories) should be considered, we examine the frequent patterns identified so far, to see which additional entity types appear in them, if any (line 4). Correspondingly, we add their (reduced) revision histories within w to the set of considered actions. For that, we employ again the function $reduced\_and\_abstract\_actions(S, w)$ (line 8) that reduces the revision histories and adds the actions abstraction (and their corresponding realizations) to $abstract\_actions[w]$ (resp. $realization\_table[w]$).

Next, we iteratively consider for each previously discovered frequent pattern $p \in patterns[w]$, its graph g_p and attempt to extend it with additional edges (abstract action) $a \in abstract\_actions[w]$, that has not been considered for it yet (lines 9-14). The procedure uses the auxiliary global variable $tested[w]$, (initially the empty set) to record pairs of patterns and actions that have already been examined. It is important to note that by considering all action abstractions (rather than just their base type) we can construct patterns at all abstraction levels.

Extended patterns whose frequency exceeds the threshold are added to $patterns[w]$ (line 14). We will explain later how the pattern realizations and frequency are efficiently computed. When the frequent patterns can no longer be extended *w.r.t. the current set of abstract actions/action realizations*, we check again whether the discovered patterns contain new types whose actions have not yet been considered (line 4). If so, we repeat the graph and patterns extension (lines 5 - 15). Observe that the incremental nature of the patterns' construction allows refining the previously derived patterns with the newly added abstract actions, rather than computing frequent patterns from scratch. In other words, the extension of the actions graph, and the extension of the patterns (w.r.t. the extended graph), interleave well.

Note that, in the presentation so far we keep in $patterns[w]$ *all* the discovered frequent patterns and not just the most-specific ones. This is because such general patterns may still be useful, in later iterations, being expended to other, different most-specific patterns. However, an optimization that we can still employ here is the removal of these (not most-specific) patterns whose expansions have been fully examined, e.g. where all the entities types occurring in them have been thoroughly processed (line 15). Another optimization that we employ (omitted from the pseudo-code) is the cashing of the computed frequencies/realization tables, to be reused if the same patterns are later re-examined with different thresholds. When all patterns have been discovered, we select the most specific ones and return them (line 16).

Computing patterns realization and frequency. To complete the picture we need to explain how the patterns realizations and frequency are computed in lines 12-13 of the algorithm. To efficiently compute (and extend) pattern realizations, we represent each pattern realization in $realizations[w][p]$ by a *relational table* whose attribute names correspond to the pattern variables names, and whose tuples capture the different realizations of the pattern in the given time window (namely the qualifying assignments of concrete Wikipedia graph nodes to the pattern variables).

Now, note that given a pattern p and an abstract action a, there may be several ways to extend the graph g_p with a. First, a's source may be "glued" to any of the nodes (variables) in p of the same type as a (if such exist). Second, for each such possible gluing, a's target may either be added to the pattern as new pattern node (in which case g_p is extended by both a new edge and a new node) or the target may also be glued to an existing same type node (in which case g_p is extended by only a new edge).

We process each such possible extension as follows. Let p' be such an extended pattern. An important observation is that, using the relational representation discussed above, the realization table of the extended pattern p' can be easily computed, from $realizations[w][p]$ and $realizations[w][a]$, via a join-based query. For the glued pattern/action nodes we use equijoin on the corresponding attributes, whereas for the new node (if such exists), we require inequality to all same type attributes. Finally, we only need to project a single column for each pattern attribute. Then, the frequency of a pattern p w.r.t. a type t can be easily computed from the relation, by an SQL count operator that counts the number of distinct nodes appearing in the column corresponding to the pattern's source variable, (then dividing the count by the cardinality of $entities(t)$).

Mining Relative Patterns. To conclude, we note that the computation of *relative* frequent patterns proceeds in a similar manner. The only difference is that each pattern p we begin the expansion process starting from p itself, and relative frequency (rather than just frequency) is computed similarly, but using the formula in Definition 3.4. We omit the details for space constraints.

4.3 Finding Windows and Thresholds

So far we assumed that we are given a window w and a threshold τ, and our goal was to identify the (relative) frequent patterns in w, w.r.t. the seed type t. To identify windows and thresholds of potential interest, we use a simple heuristic, which our experiments show to be extremely effective.

We restrict our attention to non-overlapping time windows and split the revision histories accordingly. This allows parallelizing the processing of the action sets in the different windows. Our analysis of real Wikipedia data indicates this to be a reasonable design choice. For an input type t there are very few meaningful (update-wise) time frames that overlap and those can be merged into a somewhat longer window that includes both update patterns.

Algorithm 1: Mine connected patterns

Input: entity set S,Wikipedia type t, window w, frequency threshold τ, relative threshold τ_{rel}
Output: (relative) patterns and their time frames: $patterns[w]$, rel_patterns[w][p]

1 call reduced_and_abstract_actions(S, w) to create abstract_actions[w] and realizations[w];
2 patterns[w] = $\{\{a\} \mid a \in$ abstract_actions[w] $\wedge type(source(a)) = t \wedge$ frequency($\{a\}$) $\geq \tau$ $\}$;
3 tested[w]={};
4 **while** *new type names found in patterns[w]* **do**
5 **foreach** $p \in$ *patterns[w]* **do**
6 **foreach** *new type name* $t \in p$ **do**
7 S = get_entities(t);
8 call reduced_and_abstract_actions[S, w] to expand abstract_actions[w] and realizations[w];
9 **while** *there exists* $p \in$ *patterns[w],* $a \in abstract\_actions[w]$, *s.t.* $(p, a) \notin$ *tested[w]* **do**
10 tested[w]=tested[w] $\cup \{(p, a_i)\}$;
11 **foreach** *pattern* p' *obtained by expending* p *with* a_i **do**
12 compute realizations[w][p'] from realizations[w][p] and realizations[w][a_i];
13 frequency(p') = $\frac{|distinct\ entities\ of\ type\ t\ in\ realizations[w][p']|}{|entities(t)|}$; **if** *frequency*($p'$) $\geq \tau$ **then**
14 patterns[w]=patterns[w] $\cup \{p'\}$;
15 $prune(patterns[w]), realizations[w]$
16 patterns[w]=most_specific_patterns(patterns[w]);
17 **Return**($patterns$)

Our algorithm is initialized with minimal window size (the system default is two weeks) and frequency thresholds (default 0.7), which are iteratively refined: The window size is extended (resp. the threshold it lowered) if no qualified patterns were found, or if the refinement leads to the discovery of additional patterns. The extension granularity (resp. frequency bound reduction) may be determined by the user. Otherwise, the default refinement policy is to alternate between multiplying the window size by two (retaining the threshold as it) and reducing the frequency thresholds by 20% (retaining the window size). This is repeated as long as the refinement leads to new patterns, up to a maximal window size of one year, and a minimum threshold value of 0.2 (All experiments were run with this setting). We chose the above heuristic by examining several alternatives, as elaborated in Section 6, and chose the one with the lowest running time among all heuristics that performed best in terms of $F1$ score evaluations.

We now present the full algorithm, depicted in Algorithm 2. As mentioned above, given an entity type t, our initial entity set S contains all entities of the input type. Users not familiar with the type hierarchy may provide a seed entity e and the system will use $type(e)$ as an input (lines 1-3). To derive $type(e)$ we use an alignment from Wikipedia entities to DBPedia [1]. Then to find all entities of type t we employ a corresponding inverse index.

We first split the timeline into consecutive time frames of size W_{min} (line 7). Next we call (possibly in parallel) the procedure $Mine\_connected\_patterns$, described in Algorithm 1 in Section 4, for all windows (line 9). We iteratively refine the considered windows width (W_{min}) and frequency threshold (τ) (following the heuristics described above), and until a stable result is obtained (lines 10-11). Finally, for each discovered pattern p in window w, its relative frequent patterns are mined as well (lines 14).

Algorithm 2: Find windows and patterns

Input: Wikipedia type t or seed entity e, min. window width W_{min}, frequency threshold τ, relative threshold τ_{rel}
Output: (relative) patterns and their time frames

1 **if** *t is not given* **then**
2 t = type(e);
3 S = get_entities(t);
4 patterns = [];
5 rel_patterns = [];
6 **Frequent patterns Stage;**
7 split the timeline into a set W of consecutive time frames of size W_{min};
8 **foreach** w *in* W **do**
9 patterns[w]= Mine_connected_patterns(S, t, w, τ, τ_{rel})
10 **if** *patterns==[] or refine?*(W_{min},τ,*patterns*)==*True* **then**
11 go to line 7 with the updated W_{min}, τ;
12 **Relative frequent patterns Stage;**
13 **foreach** $w \in W$ **do**
14 rel_patterns[w] = Mine_rel_connected_patterns (patterns[w], rel_patterns[w], abstract_actions[w], realizations[w],τ_{rel});
15 **Return**($patterns, rel\_patterns$)

5 USING WINDOWS AND PATTERNS

We employ the discovered windows and patterns to clean and correct Wikipedia entries, as well as to assist users in editing.

Cleaning. An immediate application of the discovered patterns is to alert Wikipedia editors on partial edits from past windows. For that, we examine the discovered windows and identify for each window and pattern (using an efficient outer-join based algorithm, described below, parallelly processed) partial sets of actions that may be extended to a full pattern occurrence. To assist the editor in determining how (if) the partial edit should be completed (or reversed), we present examples of other full patterns.

To explain how the algorithm works, recall from Section 4.2 that, to discover patterns, we iteratively expand the pattern's graph, joining corresponding action relations to form a relation table that captures the pattern realizations. In each such join, the left-hand side (LHS) relation represents the realizations of a (partially growing) portion of the pattern, and the right-hand side (RHS) relation contains the realizations of the added edge. The join conditions assert the (in)equalities of the corresponding graph nodes. To identify *partial updates*, that haven't been properly completed, we similarly traverse the graph. But instead of the abovementioned *join* operator, we employ a *full outer-join* [6], with analogous (in)equality conditions. Note that, unlike the join, the full outer-join also records in the output relation those LHS (resp. RHS) tuples not matching any RHS (LHS) tuple, *padding the missing attribute values with nulls*. In terms of our patterns, partial pattern realizations (resp. action realizations) that are missing a corresponding action (partial pattern) are also recorded in the relation, padded by null values. The incomplete edits can then be easily identified via a selection query retrieving tuples with null values. A result table keeping the attributes of original action relations is kept to record which missing updates cause null values.

Our algorithm for identifying partial updates is depicted in Algorithm 3. For a time window w and a pattern p, it focuses on the entity types in p. It invokes $reduced\_and\_abstract\_actions$ (described earlier), to examine their revision histories and construct the realization relations of their corresponding abstract actions (lines 1–2.) Next we traverse the pattern's graph g_p, and iteratively outer-join the corresponding relations (lines 8-9). We use $p_1 \ldots p_n$ to denote the incrementally growing sub-patterns (from the first singleton edge a_1, to the full pattern p). The array

Algorithm 3: Identifying partial updates

Input: window w, pattern p
Output: partial realizations of p in w

1 let S be the set of entity types in p;
2 call reduced_and_abstract_actions(S, w) to create abstract_actions[w] and realizations[w];
3 let $e_1, \ldots, e_n$ be the edges in the pattern's graph g_p, in some traversal order;
4 let $a_1 \ldots, a_n$ be the corresponding actions in p;
5 $p_1 = \{a_1\}$;
6 $all\_realizations[p_1] = realizations[w][a_1]$;
7 **for** $i = 2 \ldots n$ **do**
8 $\quad p_i = p_{i-1} \cup \{a_i\}$;
9 $\quad$ compute $all\_realizations[p_i]$ from $all\_realizations[p_{i-1}]$ and $realizations[w][a_i]$ using full outer-join;
10 $partial\_r = \{r \in all\_realizations[p] \mid r$ includes a null value $\}$;
11 **Return**$(partial\_r)$

$all\_realizations[p_i]$ is used record the intermediate (possibly incomplete) pattern instantiations. Finally we return all tuples that include null values (lines 10-11). An example of the algorithm execution appears in our technical report [3].

Edit assistance. Update patterns often appear periodically in multiple windows. For example, transfer windows occur each summer with a similar edit pattern. Our system automatically identifies such periodic patterns/windows and provides online edit assistance (via a plug-in) to users that update pattern entities within a given window, suggesting potential update completions, as explained above. The algorithm for identifying patterns that need completion follows similar lines, with the user alerted on partial edits that involve entities that she is updating.

6 EXPERIMENTS

We open this section by describing the experimental setup, the examined datasets, baselines, and evaluation methods. We then present the results, both in terms of running time and quality. Finally, we present a comparative analysis of heuristics, demonstrating the superior performance of the heuristic used by WC.

6.1 Experimental Setup

We have implemented WC as a web browser extension, with backend in Python, frontend in JavaScript, and SQL over pandas as the underlying query engine. All experiments were executed on an Intel i7 2.4Ghz with 96GB RAM and 16 cores server. We ran experiments over Wikipedia datasets and examined the system performance in terms of running times, the quality of the discovered patterns, and the number of detected errors, w.r.t. these patterns.

For the quality experiments (and measuring the running time) we use the default settings of WC. Recall that our algorithm is initialized with minimal window size (default is two weeks) and frequency thresholds (default 0.8), which are refined throughout the computation. As mentioned, the default refinement policy alternates between multiplying the window size by two and reducing the frequency thresholds by 20%, up to at most one year window and a minimal 0.2 frequency. For other experiments, that test the effect of each parameter, we vary the given parameter while setting all others to default values, as explained below.

Settings. To demonstrate the operation of WC in different entity domains, we examine here three Wikipedia domains: soccer (including players, teams, leagues, etc), cinematography (actors, movies, awards, etc) and US politicians (specifically US senators). To derive patterns (and correspondingly identify potential edit errors) we used the revision history for the year 2018. We then validated the signaled potential errors w.r.t. edits recorded in the revision history of 2019. To further assess (resp. validate) the identified patterns (signaled errors), we have also consulted three domain experts - one expert per each of the three domains.

For the soccer domain, we used major European leagues' soccer players for our seed set of entities. For the cinematography domain, we used actors from Hollywood-produced movies for the seed set. Lastly, in the politicians' domain, we used US senators for the seed entity set. In each domain, we considered different sizes of seed sets by randomly choosing between 100-1K entities from the respective seed type. We run each experiment 5 times and show the average running time (the variance was below 5%). For the entities selection, we used the "recently edited" criterion (edited in the last year of 2018) to focus on active pages with edits that may contribute to the mining process, and may also contain errors. Following Algorithm 2, we also considered related entity types and extracted their revision history in the corresponding period.

Due to the lack of an appropriate API, obtaining the Wikipedia data required crawling and parsing entities and it's revision logs. Nevertheless, we gathered data for 100K entities - about 10^{th} of the million frequently edited Wikipedia's entities [4].

Algorithms. The core of WC is Algorithm 2 (referred in the sequel as WC) which identifies time windows of interest and corresponding edit patterns. A main ingredient of WC is the pattern mining procedure depicted in Algorithm 1 (referred in the sequel as PM) that given a specific window w and frequency threshold τ, identifies the most specific frequent patterns in w (w.r.t. the seed type of interest). As explained in Section 4, PM refines conventional graph mining algorithms [15] by introducing two dedicated optimizations: (1) an efficient join-based SQL computation of patterns realization and frequencies, and (2) an incremental computation that avoids a full materialization of the edits graph. To demonstrate the importance of these two optimizations, we examine the running times of the following four algorithm variants.

- PM, our mining algorithm.
- PM^{-join}, a restricted variant of PM without our dedicated join-based queries. Instead, pattern realizations and frequencies are computed via conventional main memory nested loop.
- PM^{-inc}, a restricted variant of PM that does not utilize our incremental, on-demand graph construction. Instead, the full edits graph for the given window is materialized then given as input to the mining process (but patterns realization/frequency is still computed via our join-based queries).
- $\text{PM}^{-inc,-join}$, conventional graph mining without our two optimizations. The edits graph for the window is materialized as input to the mining process, with the pattern realizations/frequencies computed via the main memory nested loop.

Note that direct comparison to leading graph mining baselines is not possible due to their use of different frequency metric (not capturing connectivity property and relativity to a specific type) and lack of support for type hierarchy. We have thus adapted the most relevant variant to our context, denoted by $PM^{-inc-join}$, and benchmark w.r.t. it. See discussion in Section 2.

6.2 Running Time Analysis

Next, we examine how the running time is affected by (1) the size (number of entities) of the seed type of interest, (2) the frequency threshold, and (3) the window size. In each experiment, we vary one parameter while setting the others to a default value (500 seed entities, 0.7 frequency, and two weeks, resp.). As the results for the different domains show similar trends, we present here a representative set of experiments for the soccer domain.

Note that, as is common in graph mining algorithms, PM^{-inc} and PM$^{-inc,-join}$ require the full edits graph for the given window to be materialized. However, materializing this graph, even for relatively small time windows, can be infeasible. Indeed, our experiments show that even when considering a two-week time window, only 100 seed entities, and revision histories only of entities reachable from the seed set, the graph construction exceeded 24 hours (the time limit for the graph materialization). This is due to the dense connectivity of the Wikipedia graph[6] [4], the previously mentioned high volume of edits, and the lack of adequate API, as noted above. Thus, as the graph must be constructed for each considered window, we initially focus only on the two feasible algorithms: PM and PM^{-join}, with the infeasible algorithms evaluated over reduced inputs. We report below the sizes of the partial graphs built by PM and PM^{-join}. For intuition on the relative savings, we note that the graph for the 100 seed entities during these 2-weeks contains over 100K entities.

Seed set cardinality. We start by examining the running time as a function of the size of the seed set. Naturally, the more entities in the seed set, the more related updates need to be examined and the more revision histories are processed. Consequently the running time of both PM and PM^{-join} increases, as illustrated in Figure 4(a). The threshold is set here to the default value of 0.8 and the window is the month of August. Similar results are obtained for other thresholds/months. Next to the size of each seed set, we give (in parenthesis) the overall number of related entities (graph nodes) processed by the algorithm. In each column, the upper part shows how much time (in hours) it took to parse the revision history of the relevant entities and extract the reduced updates set. This is naturally identical for both algorithms (as they only differ in the computation of pattern realizations/frequencies). It should be noted that this time would be much shorter if Wikipedia had provided a more convincing API for its revision logs or, publicly-available structured revisions database. The lower part of each column shows the running time dedicated to the pattern mining itself. We can see that is significantly shorter for PM that employs our efficient join-based queries. For PM the pattern mining time only marginally grows when the seed set size increases and stays below 15 min, which is very reasonable for offline computation.

Frequency threshold. Next, we examine the running time as a function of the frequency threshold. The seed set size is set to a default size of 500 and the window is the month of August. (Similar results are obtained for other sizes/months). The lower the threshold, the more potential patterns (and revision histories of involved entity types) need to be examined, and, consequently, the processing time of both algorithm increases, as illustrated in Figure 4(b). The processing time for the revision logs is the same in both algorithms, but PM mines the patterns much faster. Again, for PM the pattern mining time increases only moderately when the threshold decreases and stays below 15 min.

Window size. In this experiment, we measure the preprocessing time for varying window sizes. Figure 4(c) illustrates the processing time for 2, 4 and 8 weeks window. Specifically, we see here the running times for the first two weeks of August, the whole month of August, and the two months July and August, but similar results are obtained for other similar-length windows. (The seed set size here is again set to default size of 500 and the frequency default 0.8. Similar results are obtained for other sizes/frequencies). Naturally, the larger the window, the more updates need to be processed and as a result, more patterns may occur. Consequently, the running time increases. Again, the processing time for the revision logs in both algorithms is the same, but PM mines the patterns much faster.

Parallelism. So far we examined the performance of the PM component of WC. To complete the discussion we now examine the full operation of WC, highlighting, in particular, its embarrassingly parallelized nature. Recall that WC splits the timeline into non-overlapping windows that may be processed in parallel. Similarly, independent entity types can be processed in parallel. This is easily exploitable in a multi-core setting as shown in 4(d). We focus here on the pattern mining process (the revision logs processing shows similar trends). The figure shows the time in minutes (in log scale) of the pattern mining computation for a single core vs 16 cores, for varying sizes of seed entity sets. As before, next to the size of each seed set we give in parenthesis the overall number of related entities (nodes) processed by the algorithm. Note that the numbers here are the *total number of nodes* processed through all iterations, for all examined windows and threshold values. Running on 1000 entities takes less than one minute on a single core. 5K entities need 6 minutes to process on one core and about 1 minute on 16 cores. For 100K entities - the largest entities set generated in the algorithm execution on the three domains mentioned above - it took 58 minutes on one core and about 15 minutes on 16 cores. Overall, the parallelization speedup is about 4x.

Based on known statistics of approximately 5.9 million Wikipedia entities (one million of them are of mid-to-high importance) [4], given a preprocessed Wikipedia revisions database/graph (which unfortunately is currently not publicly available), running on all Wikipedia entities will take about six hours (one hour on mid-to-high importance entities) on a 16 core server.

Experiments with small data. As mentioned above, the materialization of the entire edits graph of Wikipedia, which is a necessary input for PM^{-inc} and $PM^{-inc-join}$, takes impractical time. To, nevertheless, evaluate the efficiency of these two algorithms, we also conducted experiments over considerably smaller subsets of the Wikipedia graph. Over such small instances, the running time is less meaningful, however, we can focus instead on the number of considered pattern candidates as an indication of the efficiency of these algorithms. Note that, since this experiment is only possible over small data, typically negligible amounts of noise become significant, and since the number of seed entities is small, many of the identified candidates will exceed the threshold. Therefore, we do not examine any quality indicators.

Concretely, we examined a small subset of Wikipedia, consisting of 10 seed entities from the soccer domain, and all the revisions of these entities that occurred within an arbitrarily chosen two-week period. We constructed the corresponding edits graph, containing the seed entities and a close (2-reachable) neighborhood of these seeds in two phases, as follows. We first added to this graph all the entities that are connected within one link from the seeds and were also edited in the chosen time window, and then we also added, analogously, another layer of neighboring entities - all the entities that are connected within one link to the previously added entities, and were also edited in the chosen time window. We did not extend the graph further, as it could not be materialized within the time frame we defined. The above construction resulted in a graph with roughly 10K entities.

We compared the performance of PM^{-inc} and $PM^{-inc-join}$ over this graph, to that ofPM (our pattern mining algorithm) and PM^{-join} (which does not include our dedicated join-based

[6]Wikipedia contains about 6 million entities (of which 4 million are considered of marginal importance) and over 80 million internal links as to 2010.

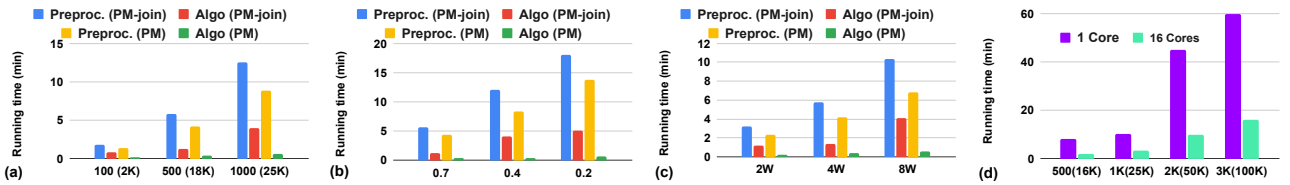

Figure 4: Running time when varying the (a) DB size (b) threshold (c) window size (d) WC execution time on 1 vs 16 cores

queries) over a Wikipedia subgraph of the same size. Recall thatPM and PM^{-join}, in contrast to PM^{-inc} and $PM^{-inc-join}$, do not receive the complete graph as input, rather create the relevant edits subgraph (of the Wikipedia graph) incrementally on-the-fly. Therefore, to ensure a meaningful comparison, we used as input a set of 200 seeds, as this results in subgraphs of roughly 10K entities (which is also the size of the input graphs for PM^{-inc} and $PM^{-inc-join}$). Moreover, as we focus solely on the number of considered candidates, this value will be the same for all variants ofPM, when employed over the same graph, as the frequency definition is identical for all baselines. Therefore, the result will be the same forPM and PM^{-join}, and also the same for PM^{-inc} and $PM^{-inc-join}$. Hence, we essentially compare only two approaches in this experiment (receiving the complete graph in advance versus computing a more relevant subgraph on-the-fly).

The results show that PM^{-inc} and $PM^{-inc-join}$ consider more candidates (524), compared toPM and PM^{-join} (125). This demonstrates the superiority of our incremental graph construction approach, which prunes many of the irrelevant candidates.

6.3 Quality Analysis

To assess the usefulness of WC for error detection we evaluated the quality of the discovered patterns and the validity of the potential errors signaled using these patterns. We employed WC, over the subsets of the Wikipedia 2018 revision log relating to the domains of *soccer*, *cinematography*, and *US politicians*, with the corresponding seed sets consisting of 1000 entities.

Ground truth patterns. To evaluate the correctness and coverage of the detected patterns, we asked each of the three experts to provide a comprehensive list of common periodic update patterns, in structured data. The soccer expert provided 11 such patterns (e.g., the page of a player that won the "Goal of the Month" award should link to the page of the award and vice versa). The cinematography expert provided 8 patterns (e.g., a TV series page should point to all the pages of its specific seasons). Lastly, the politics expert provided 5 patterns (e.g., the page of a newly-elected senator points to her predecessor's page and vice versa).

Discovered patterns and detected errors. Interestingly, the patterns derived by WC are a proper subset of the set of patterns provided by the experts, implying 100% precision. In terms of recall, our algorithm detected 9 (out of the 11) soccer-related patterns, 7 (out of the 8) cinematography related patterns and 4 (out of the 5) US politicians related patterns, yielding an average recall of 83.3% across all the domains. The discovered patterns were then used by WC to detect erroneous updates.

Running Algorithm 3 on the 2018 revision log we have identified 3743 potential errors for the soccer domain, 2554 potential errors for the cinema domain and 1125 potential errors for US politicians. To determine which of these are actual errors, we ran a two-step verification process. First, for each signaled potential error (partial pattern occurrence) we examined whether it still existed after the 2019 updates had been applied. Errors that were eliminated (corrected) are considered true errors. Note, however, that the remaining set may still include actual errors that went unnoticed. To determine how many such signaled, unnoticed errors Wikipedia still contains, we sampled 50 such errors per pattern and asked the relevant domain expert to determine their validity. Next, examples and results of discovered patterns are provided.

Soccer. Out of the 3743 signaled potential errors, 2680 were corrected in 2019 (71.6%). From the remaining examined cases, 82.1% were indeed verified as actual previously unnoticed errors.

To illustrate, the simplest pattern detected in the soccer domain indicates that, after joining a new club, the page of the player should link from the *career* table to the page of the club, which, in turn, should add a link to the player's page in the *current squad* table. This pattern has a frequency of 0.8 in the window consisting of the first week of August. Out of the 50 sampled errors for this pattern (partial pattern occurrences), 48 indeed turned out to be previously unnoticed errors (96%). A more complex pattern includes also the deletion in the player's page of the link to the old club, and vice versa. This pattern has a lower frequency (0.4) and a wider window size (the first two weeks of August). Here, out of 50 sampled potential errors, 44 were verified as actual errors (88%). An example of such an error, detected by the algorithm, relates to the page of Nikola Mitrovic, a player that switched leagues. His new club, ZTE, added him to its current squad table, while the previous club, Kesla, did not remove him. Similarly, Aleksandrs Cauna's page was updated when he joined his new club Jelgava, whereas the page of RFS, his old club, still pointed to his page past the transfer window. A relative frequent pattern, that the algorithm detected, includes an update of the *current league* link in the player's page. While this pattern is much less frequent (since a player may move to a club in the same league, in contrast to the previously mentioned patterns, where a violation almost certainly results in "incomplete" data), its relative frequency, nevertheless, exceeds the threshold. Out of the 50 detected potential errors, 14 were indeed actual previously unnoticed errors.

Cinematography. One example of a detected pattern relates to an actor/actress winning the Oscar award: the page of the winner should link to the page of the award and vice versa. In terms of quality evaluation, out of the 2554 signaled potential errors, 1731 were corrected in 2019 (67.8%). Of the remaining cases, 81.2% were determined to be true unnoticed errors.

US Politicians. An illustrative example of a discovered pattern in the US politics domain pertains to the election of a new senator. Given such an event, the pages of the new senator and the relevant state must point to each other, and also a link to the page of the previous senator is removed from the page of the state. The page of the previous senator should still point to the state, since the only modification relates to the adjacent text, detailing the period during which she held office. Out of the 1125 signaled potential errors, 728 were corrected in 2019 (67.8%). Of the remaining cases, 78.1% were determined to be previously unnoticed errors.

Insights. To conclude, we discuss insights derived from the above evaluation, that reaffirm the distinction between our intended use-cases and those addressed by previous works. As mentioned in the Introduction, our solution focuses on patterns that are associated with a well-defined time window, complementing existing solutions that target 'window-less' constraints. Indeed, for all the discovered patterns, a statistically significant time window was identified. In contrast, of the few overlooked patterns,

Table 1: Sample of heuristics test

(w, τ)	Running time (min)	Precision	Recall	F1 Score
2.0x, 20%	**2**	**1**	**0.84**	**0.91**
1.0x, 20%	1.2	0.88	0.68	0.77
2.0x, 0%	1.2	1	0.75	0.86
1.5x, 10%	3.2	1	0.68	0.81
3.0x, 40%	1.5	0.75	0.88	0.81

two are not clearly associated with any time window. This further reinforces the contrast between our solution and other works.

6.4 Parameter Tuning

When refining the two parameters across different iterations, *PM* alternates between multiplying the window size by two and reducing the frequency threshold by 20%. To arrive at these values, we performed a grid search, selecting the parameters that led to the fastest running time among the options that yield the best $F1$ score (w.r.t. patterns provided by experts). We checked combinations of reducing the threshold by X and multiplying the window size by Y, where X ranges from 1% to 100%, in steps of 5%, and Y ranges from 1.5 to 5, in steps of 0.5. In terms of the bounds for the above parameters, the window size is restricted to the range of two weeks to one year, while the threshold is restricted to $[0.2, 0.7]$. These intervals were also derived via an analogous grid search over various ranges. A sample of the results is depicted in Table 1, where the left column provides the combination of the changes in the values of the window size and the threshold. Note that the first row pertains to the combination used by `WC`.

These results demonstrate the advantages of our balanced approach, compared to more extreme approaches. Namely, opting for very small changes to the parameters increases the running time and lowers the recall. The recall drops because WC would terminate at an early stage, as new patterns are not likely to be discovered compared to the previous iteration. At the other extreme, drastically changing the parameter values, while improving the running time, lowers the precision score. The latter effect is due to quickly reaching iterations where the time window is large and the threshold is low, causing WC to discover erroneous patterns, whereas WC with our heuristic would terminate prior to this point.

7 CONCLUSION

This paper presents WC, a Wikipedia plug-in assisting editors in maintaining the correctness of inter-links. Given an entity type of interest, our efficient, highly parallelizable algorithm identifies relevant edit patterns across revision histories of entities of related types, along with time windows in which partial edits are acceptable. The discovered patterns/windows are then used by WC to alert editors on past edits that appear incomplete, and provide users with on-line assistance as they update Wikipedia. Our experiments with Wikipedia data demonstrate the efficiency and effectiveness of our approach in identifying and correcting errors.

There are several directions for future research. As our work considers inconsistencies in structured parts of Wikipedia, expanding our approach to consider free text, in particular parts related to the inter-links, is a challenge. Another intriguing future direction is enriching the expressiveness of the patterns to support value-specific instantiations (e.g., a pattern specific to PSG, but not to football clubs in general). Finally, applying our ideas to other domains where revision histories are available and link consistency is important (e.g., software repositories) is another challenge.

Acknowledgements This work has been partially funded by the Israel Science Foundation, the Binational US-Israel Science Foundation, and the Tel Aviv University Data Science center.

REFERENCES

[1] DBPedia. https://wiki.dbpedia.org/.
[2] Time Magazine, "Jimmy Wales". http://content.time.com/time/specials/packages/article/0,28804,1975813_1975844_1976488,00.html.
[3] WiClean Technical Report. http://slavanov.com/research/wc-tr.pdf.
[4] Wiki statistics. https://en.wikipedia.org/wiki/Wikipedia:Statistics.
[5] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *PVLDB*, 9(4):336–347, 2015.
[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.
[8] E. Alfonseca, G. Garrido, J.-Y. Delort, and A. Peñas. Whad: Wikipedia historical attributes data. *Language resources and evaluation*, 47(4), 2013.
[9] A. Assadi, T. Milo, and S. Novgorodov. Cleaning data with constraints and experts. In *WebDB*, pages 1:1–1:6, 2018.
[10] M. Atzori, S. Gao, G. M. Mazzeo, and C. Zaniolo. Answering end-user questions, queries and searches on wikipedia and its history. *IEEE Data Eng. Bull.*, 39(3):85–96, 2016.
[11] M. Bergman, T. Milo, S. Novgorodov, and W. C. Tan. Query-oriented data cleaning with oracles. In *SIGMOD*, pages 1199–1214, 2015.
[12] T. Bleifuß, L. Bornemann, T. Johnson, D. V. Kalashnikov, F. Naumann, and D. Srivastava. Exploring change: a new dimension of data analytics. *Proceedings of the VLDB Endowment*, 12(2):85–98, 2018.
[13] S. Bostandjiev, J. O'Donovan, C. Hall, B. Gretarsson, and T. Hollerer. Wigipedia: A tool for improving structured data in wikipedia. In *2011 IEEE Fifth International Conference on Semantic Computing*, pages 328–335, Sep. 2011.
[14] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509, Aug. 2013.
[15] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7), 2014.
[16] W. Fan, X. Wang, Y. Wu, and J. Xu. Association rules with graph patterns. *Proceedings of the VLDB Endowment*, 8(12):1502–1513, 2015.
[17] B. Fetahu, A. Anand, and A. Anand. How much is wikipedia lagging behind news? *CoRR*, abs/1703.10345, 2017.
[18] J. Ge and Y. Xia. Distributed sequential pattern mining in large scale uncertain databases. In *PAKDD*, pages 17–29. Springer, 2016.
[19] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.
[20] S. Goldberg, T. Milo, S. Novgorodov, and K. Razmadze. WiClean: a system for fixing Wikipedia interlinks using revision history patterns. *PVLDB*, 12(12):1846–1849, 2019.
[21] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.
[22] A. Heidari, J. McGrath, I. F. Ilyas, and T. Rekatsinas. Holodetect: Few-shot learning for error detection. *arXiv preprint arXiv:1904.02285*, 2019.
[23] S. Heindorf, M. Potthast, B. Stein, and G. Engels. Towards vandalism detection in knowledge bases: Corpus construction and analysis. In *SIGIR*, 2015.
[24] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1):75–105, 2013.
[25] P. Kin-Fong Fong and R. Biuk-Aghai. What did they do? deriving high-level edit histories in wikis. 01 2010.
[26] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271, 2005.
[27] J. Lajus, L. Galárraga, and F. Suchanek. Fast and exact rule mining with amie 3. In *European Semantic Web Conference*, pages 36–52. Springer, 2020.
[28] N.-T. Le, B. Vo, L. B. Nguyen, H. Fujita, and B. Le. Mining weighted subgraphs in a single large graph. *Information Sciences*, 514:149–165, 2020.
[29] M. Muzammal and R. Raman. Mining sequential patterns from probabilistic databases. *Knowledge and Information Systems*, 44(2):325–358, 2015.
[30] S. Ortona, V. V. Meduri, and P. Papotti. Rudik: Rule discovery in knowledge bases. *Proceedings of the VLDB Endowment*, 11(12):1946–1949, 2018.
[31] D. Raghu, S. Nair, and Mausam. Inferring temporal knowledge for near-periodic recurrent events. 2018.
[32] N. Rajkumar, M. Karthik, and S. Sivanandam. Fast algorithm for mining multilevel association rules. In *Proc. of TENCON*, pages 688–692, 2003.
[33] A. Sarabadani, A. Halfaker, and D. Taraborelli. Building automated vandalism detection tools for wikidata. WWW 2017 Companion.
[34] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of EDBT*, pages 1–17, 1996.
[35] O. Sunercan and A. Birturk. Wikipedia missing link discovery: A comparative study. In *2010 AAAI Spring Symposium Series*, 2010.
[36] T. P. Tanon, C. Bourgaux, and F. M. Suchanek. Learning How to Correct a Knowledge Base from the Edit History. In *WWW*, 2019.
[37] T. Tran and T. N. Nguyen. Hedera: Scalable indexing and exploring entities in wikipedia revision history. pages 297–300, 2014.
[38] R. West, A. Paranjape, and J. Leskovec. Mining missing hyperlinks from human navigation traces: A case study of wikipedia. In *WWW*, 2015.
[39] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Proc. of KDD*, pages 286–295, 2003.
[40] R. Ying, A. Wang, J. You, and J. Leskovec. Frequent subgraph mining by walking in order embedding space. 2020.
[41] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.