Ontology-based Entity Matching in Attributed Graphs

Hanchao Ma
Washington State University
hanchao.ma@wsu.edu

Morteza Alipourlangouri
McMaster University
alipoum@mcmaster.ca

Yinghui Wu
Washington State University
Pacific Northwest National Laboratory
yinghui.wu@wsu.edu

Fei Chiang
McMaster University
fchiang@mcmaster.ca

Jiaxing Pi
Siemens Corporate Technology
jiaxing.pi@siemens.com

ABSTRACT

Keys for graphs incorporate the topology and value constraints needed to uniquely identify entities in a graph. They have been studied to support object identification, knowledge fusion, and social network reconciliation. Existing key constraints identify entities as the matches of a graph pattern by subgraph isomorphism, which enforce label equality on node types. These constraints can be too restrictive to characterize structures and node labels that are syntactically different but semantically equivalent. We propose a new class of key constraints, *Ontological Graph Keys* (OGKs) that extend conventional graph keys by ontological subgraph matching between entity labels and an external ontology. We show that the implication and validation problems for OGKs are each NP-complete. To reduce the entity matching cost, we also provide an algorithm to compute a minimal cover for OGKs. We then study the entity matching problem with OGKs, and a practical variant with a budget on the matching cost. We develop efficient algorithms to perform entity matching based on a (budgeted) Chase procedure. Using real-world graphs, we experimentally verify the efficiency and accuracy of OGK-based entity matching.

PVLDB Reference Format:
Hanchao Ma, Morteza Alipourlangouri, Yinghui Wu, Fei Chiang, Jiaxing Pi. Ontology-based Entity Matching in Attributed Graphs. *PVLDB*, 12(10): 1195-1207, 2019.
DOI: https://doi.org/10.14778/3339490.3339501

1. INTRODUCTION

Keys are a fundamental integrity constraint defining the properties to uniquely identify an entity. Keys serve an important role in relational and XML databases during design, normalization, and query optimization where they are commonly used for object reconciliation, to minimize redundancy, and to improve query runtimes. All these benefits transfer to graphs, where key constraints have been studied for entity identification [3, 7, 10, 13, 16, 32]. The application of keys to graphs extends beyond deduplication to include emerging knowledge fusion [14] and fact checking [28].

Keys for graphs are inherently more complex than their relational counterparts due to the absence of schema, variances in topology and node types, and they may be *recursively defined*. Keys for graphs incorporate a topological constraint expressed by a graph pattern Q to uniquely identify entities. For example, keys for XML data identify duplicate entities via regular paths [10]; and for graphs, key constraints are posed on node matches induced by subgraph isomorphism [16]. Existing work has studied the theoretical foundations and applications of key constraints, and their generalized counterpart, graph functional dependencies [18,22,37]. These constraints have been applied to data cleaning [21], data validation [6], and entity matching [16].

Entities in real-world knowledge graphs often contain heterogeneous labels and multiple attributes. This poses two challenges for entity matching over graphs: (1) nodes that should refer to the same entity may not be captured by key constraints that only enforce label equality [31]; and (2) nodes with equal labels that match key patterns may not necessarily refer to the same entity, due to differing attributes. Furthermore, such graphs are often interpreted with respect to (w.r.t) an ontology that provide domain specific concepts and relationships, defining semantic equivalence among node labels. Consider the following example.

Example 1: Consider a knowledge graph G consisting of triples (subject, predicate, object) where subject and object are nodes, and predicate is an edge connecting subject to object. Figure 1 illustrates a fraction of DBpedia G, with three subgraphs describing three music entities $\{v_1, v_2, v_3\}$, where each node has an associated type denoted in parentheses. For example, entities *omg_mike* and *omg* in v_1 and v_3, respectively, are both of type *song*.

Consider graph keys φ_1 and φ_2 depicted as graph patterns P_1 and P_2 in Figure 1. φ_1 states that "*if two songs share the same name and album, then they refer to the same song*". Similarly, an album can be identified by its name, year of release and artist, characterized by φ_2. Note the dependence of φ_1 on φ_2, to identify a song, we need to first identify its artist, reflecting the recursive property of graph keys [16]. Applying φ_1 and φ_2 via subgraph isomorphism on G, we obtain only v_3 as a match, since (1) v_1 *end of days* is of type

Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339501

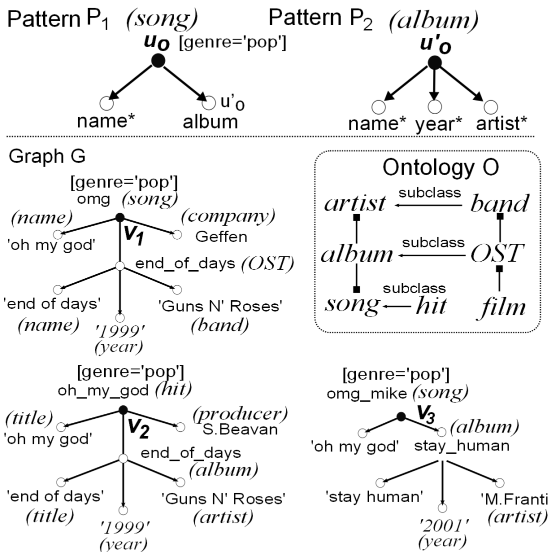


Figure 1: Entity matching with ontologies.

OST (rather than *album*); (2) v_2 is of type *hit* rather than *song*; and (3) the v_1 album predicate *band* fails to match the required label *artist*. Hence, φ_1 and φ_2 fail to identify v_1 and v_2 as the same song as they rely only on label matching.

Given an ontology O, as shown in Figure 1, we exploit ontological relationships and semantic equivalance to extend graph keys. For example, we recognize an *OST* is a type of *album* participating in a hyponym (subClassOf) relationship. Similarly, we note that labels *artist* and *band*, and types *hit* and *song* are semantically similar. By extending φ_1 and φ_2 with these ontological equivalences, we identify v_1 and v_2 are indeed the same song.

Not all labels in O are useful. Ontological similarity is often characterized within a scope such that concepts that are 'far apart' in O (*w.r.t.* a distance function) are not conceptually close. For example, if we replace the *OST* entity with a *film* entity to create an entity v_4, this will not match φ_2 using a distance threshold of two, since the distance between *album* and *film* in O exceeds this bound. □

Beyond entity deduplication, ontological extension of graph keys enrich the neighborhood of equivalent entities. For example, merging v_1 and v_2 yield two new edges for the *song* entity: one with *company Geffen*, and the second with *producer S. Beavan*, completing the *song* information. These semantic extensions have widespread applications to link prediction, learning and inference for knowledge base completion [19, 26, 28], and knowledge fusion [14].

The above example highlights the need for a new class of dependencies for graphs that go beyond existing subgraph isomorphism to consider entity matching with ontological similarity. We extend existing notions of keys for graphs, which uniquely identify an entity, to exploit the relationships among node labels given in an ontology. The recursive property of graph keys allows us to precisely define related entities for the keys. By incorporating ontologies, we further increase the scope of entities that can be matched to recursive keys to include matches that are ontologically similar.

While ontological extensions have been studied for traditional functional dependencies [8, 11], little work has been done to enrich graph keys with ontologies.

Contributions. We extend keys for graphs with ontological pattern matching, and illustrate their applications.

(1) We propose *Ontological Graph Keys* (OGKs), a new class of key constraints that exploit ontologies to enhance keys for graphs. An OGK includes an event pattern that defines an entity u, which is used to identify similar concepts w.r.t. an ontology O. We characterize entity equivalence by matching: (i) pairs of equivalent subgraphs w.r.t. the key constraints, which may be recursively defined; and (ii) value constraints defined on the node attributes.

(2) We study the foundations of OGKs, including their satisfiability, validation, and implication. We show that adding ontologies does not make these fundamental problems harder, compared to traditional keys that enforce label equality. For example, both validation and implication of OGKs is NP-complete. We also provide an algorithm to compute the minimal cover for OGKs based on our implication analysis.

(3) We formally introduce the entity matching problem using OGKs. Given a set of OGKs Σ, a *scope* (G, O, θ) that consists of graph G, ontology O, and a matching cost threshold θ to ensure semantic closeness, the problem is to compute an equivalence relation R, such that the quotient graph induced by R of G satisfies Σ. While this problem is NP-complete, we introduce efficient algorithms to enforce Σ.

(a) To characterize the matching process, we revise the Chase process of conventional data dependencies for OGKs, by incorporating ontology matching that trigger a sequence of non-destructive "merge" operations over equivalent entities. We show that the Chase with OGKs satisfies the *Church-Rosser* property, i.e., Chase sequences are finite and terminating, resulting in a unique graph satisfying the OGKs.

(b) We define *early terminating* criteria for the revised Chase, and the corresponding entity matching algorithms over recursively defined OGKs. Our dynamic programming algorithm consists of two efficient phases: (i) a top-down phase that decomposes OGKs to smaller, tree constraints to refine matches, and perform early validation; and (ii) a bottom-up synthesizing phase that assembles the matches, and induced entity equivalence classes for recursive entity matching. We develop optimization techniques to prune unpromising matches that reduce the verification cost.

(c) Given limited resources for entity matching, we propose a practical variant of the Chase that includes a cost model for matching and editing entities in G. We compute a Chase sequence that minimizes the cost under budget B, enforcing OGKs that tend to merge highly similar entities. We develop an anytime algorithm that can be interrupted to return the Chase sequence identified thus far, with tunable memory.

(4) We experimentally verify the efficiency and effectiveness of our OGK-based techniques using two real-world benchmarks. We compare against two existing baselines to verify our ability to identify semantically equivalent entities in attributed graphs that are ignored by existing solutions.

2. ONTOLOGICAL GRAPH KEYS

We provide definitions, and introduce ontological graph keys.

2.1 Preliminaries

Graphs. We consider directed, attributed graphs $G = (V, E, L, F_A)$, where V is a set of nodes, and $E \subseteq V \times V$ is a set of edges. For each node $v \in V$ (resp. edge $e \in E$),

$L(v)$ (resp. $L(e)$) is a *type* (resp. a relation) from a finite alphabet τ. For each node v, its value is denoted as v.val. The v.val is an example of an *attribute* of v, describing a node property. For each node v, its attributes $A_i \in \mathcal{A}$, $i \in [1, n]$ are captured in its *property tuple*, $F_A(v)$, defined as a sequence of attribute-value pairs $\{(v.A_1, a_1), \ldots (v.A_n, a_n)\}$. Each pair $(v.A_i, a_i)$ states that the attribute $v.A_i = a_i$.

Entity identifiers. To define the mapping between a node v and a real-world entity u, we introduce entity identifiers. Given a set of entities $\{u_1, \ldots, u_m\}$, we associate a unique entity identifier eid_i to entity u_i ($i \in [1, m]$). Each node v carries a (possibly empty) list of *entity identifiers* $\{v.\mathsf{eid}_1, \ldots, v.\mathsf{eid}_m\}$, For each $\mathsf{eid}_i \neq \mathsf{Null}$, this indicates that v encodes an instance of entity u_i. We enforce two types of node equality: (1) two nodes v and v' are *value equivalent* if v.val $= v'$.val; and (2) v and v' are *entity equivalent w.r.t.* entity u (with entity identifier eid_u), if $v.\mathsf{eid}_u = v'.\mathsf{eid}_u$.

Remarks. Nodes in G may encode a *node identifier*, and an *entity identifier*(eid) to distinguish different nodes and different entities, respectively. In real world graphs, a single node may model instances of two different entities, and similarly, two distinct nodes may model the same instance of an entity. These specifications often occur in multi-typed entities, which are common in property graphs [5], knowledge bases [24] and social networks [27]. Existing keys for graphs only enforce node identity, and do not differentiate between entity vs. node identifiers [10, 16].

Ontologies. An ontology is a directed graph $O = (V_o, E_o)$, where V_o is a set of *concept labels* and $E_o \subseteq V_o \times V_o$ is a set of semantic relations among the concept nodes. In practice, an edge $(v, v') \in E_o$ may encode three types of relations [25]: (a) *equivalence*, which state that v and v' are semantically equivalent, representing relations such as "refers to" or "known as"; (b) *hyponyms* that state v is a kind of v', modeling "is-a" or "'subClassOf" relations that define a preorder over V_o; and (c) *descriptive*, which state that v is described by v' in terms of 'association' or 'part-of' relations. In practice, an ontology may encode a taxnonomy, thesauri, or RDF schema. By incorporating ontologies, OGKs are more expressive than traditional graph keys [16], and capture semantic similarity relations during entity matching.

Example 2: We return to Figure 1, where entities $v_1 - v_3$ all have the property *genre* = *pop*, but v_1 and v_3 are both of type *song*, and v_2 is of type *hit*. By using the ontology O associated with G, we expand the notion of similarity to include semantic relationships among the node labels. For example, *OST* is a subclass of *album* via a hyponym edge, and *band* is semantically similar to *artist*. □

<u>*Relevant set*</u>. Given an ontology O and a concept label l, the *relevant set* to l refers to the set of concepts similar to l in O, denoted as $\mathsf{lsim}(l)$, according to a distance function $\mathsf{dist}(\cdot)$. Formally, $\mathsf{lsim}(l) = \{l' | \mathsf{dist}(l, l') \leq \alpha\}$, where $\mathsf{dist}(\cdot) : V_o \times V_o \to [0, 1]$ computes the distance between l and l', and the threshold α defines the scope of similarity. $\mathsf{dist}(l, l')$ can be defined as the normalized sum of the edge weights along the shortest undirected path between l to l' in O [24, 36]. To differentiate among the relations in O, one can assign weights w_1, w_2, w_3 to the edges representing equivalence, hyponym, and descriptive relations, respectively [25].

Example 3: For ontology O in Figure 1, we set the weights $w_1 = 0.1$, $w_2 = 0.3$ and $w_3 = 0.6$ to represent the relative cost among the ontological relations. The $\mathsf{dist}(album, OST) = 0.3$, since there is a path length 1 between these two concepts sharing a hyponym relation. Similarly, $\mathsf{dist}(film, OST) = 0.6$, for a descriptive relation. Given threshold $\alpha = 0.3$, the relevant set to *OST* is $\mathsf{lsim}\ (OST) = \{OST, album\}$. □

Entity patterns. An *entity pattern* $P(u_o)$ is a connected graph (V_P, E_P, L_P) containing a set of pattern nodes V_P, and pattern edges E_P. Each pattern node $u \in V_P$ (resp. pattern edge $e \in E_P$) has a label $L_P(u)$ (resp. $L_P(e)$). The pattern nodes V_P may be one of three types: (1) a designated *center* node $u_o \in V_P$, representing the primary entity to be identified; (2) a set of *variable nodes* $V_x \subseteq V_P$; and (3) a set of *constant nodes* $V_c = V_P \setminus (\{u_o\} \cup V_x)$.

Example 4: Consider the two entity patterns P_1 and P_2 in Figure 1, characterizing instances of *song*, and *album*, respectively. P_1 contains a constant node *name*, and a variable node *album*. Intuitively, the equivalent instances of *song* should be *recursively* determined by the equivalent instances of *album* as defined by pattern P_2. □

<u>*Matching cost*</u>. To identify entities in a graph G that match an entity pattern $P(u_o)$, we must define a mapping function from nodes and edges in $P(u_o)$ to those in G. Formally, a *matching* between $P(u_o)$ and G is an *injective* function f from V_P to V, such that, for each node $u \in V_P$, $L(f(u)) \in \mathsf{lsim}(L_P(u))$ (concepts in G are similar to the concept modeled by u), and if $(u, u') \in E_P$, then $(f(u), f(u')) \in E$.

We quantify the matching cost by applying the the principle of spreading activation [34], which propagates concept relevance by following links of semantic networks to quantify concept closeness. We treat a pattern node u_o with concept label l as a "compound" concept, characterized by its neighboring pattern nodes. Given a matching f and entity pattern $P(u_o)$, the *matching cost* of f is quantified by the distance between u_o and $f(u_o)$, which is defined as

$$c(u_o, f(u_o)) = \frac{1}{|V_P|} \sum_{u' \in V_P} c_r(u', f(u'))$$

where $c_r(u', f(u'))$ is the *relative cost* of matching u' with $f(u')$ *w.r.t.* u_o, and is computed as

$$c_r(u', f(u')) = \begin{cases} \beta^{d_{u'}} \cdot \mathsf{dist}(L_P(u'), L(f(u'))) & u' \notin V_x \\ \beta^{d_{u'}} \cdot c(u', f(u')) & u' \in V_x \end{cases}$$

Here $d_{u'}$ is the distance between u' and u_o in $P(u_o)$ (treated as an undirected graph), $\beta \in [0, 1]$ is a decay factor, and dist computes the distance of concept labels in O. When $u' = u_o$, $c_r(u', f(u))$ is simply $\mathsf{dist}(L_P(u_o), L(f(u_o)))$. Intuitively, $c(u_o, f(u_o))$ simulates a partial spreading activation focused on u_o, by aggregating the propagated cost ("dissimilarity") between each pattern node and its matches to u_o. When u' is a variable node, the cost is aggregated by recursively expanding the pattern(s) modelling u'.

Example 5: Consider $w_2 = 0.3$, and decay factor β=0.9. (1) Given a matching f between P_2(*album*) in G such that f(*album*) = *end_of_days(OST)*, f(*name*)=*oh my god(name)*, f(*year*) = *1999(name)*, and f(*artist*) = *Guns N' Roses(band)*, (a) For constant nodes *name* and *year* in P_2, the matching cost c_r(*name, oh my god(name)*) = c_r(*year, 1999(year)*) = $0.9^1 * 0 = 0$ (the concept labels *name* and *year* are omitted in ontology O); and c_r(*artist, Guns N' Roses(band)*) = $0.9^1 * 0.3 = 0.27$, due to matching *band* to *artist* via the subclassOf relation. (b) c(*album, end _ of _ days(OST)*) is thus

Table 1: Summary of notation.

Notation	Description
$G=(V,E,L,F_A)$	attributed graph G
$P(u_o)=(V_P,E_P,L_P)$	entity pattern $P(u_o)$; u_o: center node
$V_x \subseteq V_P$; $V_c \subseteq V_P$	variable nodes V_x; constant nodes V_c
$Q(u_o,G)$	query answer of Q in G
$c(u_o,f(u_o))$, $c_r(u,f(u))$	matching cost & relative cost
$\varphi(u_o)=(P(u_o),X)$	ontological graph key with literals X
(G,O,θ)	scope of OGKs with cost bound θ

computed as $\frac{1}{4}$ $(0.9^0*0.3+0.9*0+0.9*0+0.9*0.3)=0.14$, where $0.9^0*0.3$ is the relative matching cost from *album* to *end_of_days(OST)*. (2) Similarly, given a match between P_1(*song*) and G that matches *song* to v_1, *name* to *oh my god(name)* and *album* to *end_of_days(OST)*, c(*song*, *omg*) $=\frac{1}{3}(0.9^1*0+0.9^1*0+0.9*0.14)=0.042$. That is, the matching cost of a *song* depends on the propagated cost from its relevant variable and constant nodes. □

A matching of $P(u_o)$ under *scope* (G,O,θ) is an *injective* function f from V_P to V, such that: (i) for each node $u \in V_P$, there exists a *node match* $f(u) \in V$ where $L(f(u)) \in \mathsf{lsim}(L_P(u))$; (ii) for each edge $e_p=(u,u') \in E_P$, there exists an *edge match* $e=(f(u),f(u')) \in E$; and (iii) $c(u_o,f(u_o)) \leq \theta$. A *match* of P in G induced by f, denoted as $P(G,f)$, is the induced subgraph of G with nodes and edges from matching f. The notations are summarized in Table 1.

2.2 Ontological Graph Keys

We extend keys for graphs with ontologies, and present their semantics, matching criteria, and properties.

An *ontological graph key* (OGK) $\varphi(u_o)$ for an entity u_o is a pair $(P(u_o),X)$, where $P(u_o)$ is an entity pattern with center node u_o that is associated with a unique identifier eid_o, and X is a set of literals. Each literal $l \in X$ is either a constant literal of the form of $u.A=c$ (for a constant c), or a variable literal $u.A=u'.A'$, where u and u' are two nodes in $P(u_o)$, and A and A' are node attributes from $\mathcal{A}$.

Semantics. Given an OGK $\varphi(u_o)=(P(u_o),X)$, and scope (G,O,θ), a matching function f *satisfies* X, denoted as $f \models X$, if (i) $f(u_o) \neq \emptyset$ under threshold θ; and (ii) for each constant and variable literal in X, $f(u).A=c$ and $f(u).A = f(u').A'$, respectively.

Ontological Bisimilarity. Let $\varphi(u_o)=(P(u_o),X)$ be an OGK defined on the entity u_o with identifier eid_o. We define the criteria to identify equivalent entities. We say two matches $P(G,f_1)$ and $P(G,f_2)$ are *bisimilar* under scope (G,O,θ), denoted as $P(G,f_1) \sim P(G,f_2)$, if the following hold: (a) $f_1 \models X$, and $f_2 \models X$; and (b) for each pair of nodes (v_1,v_2) where $f_1(u)=v_1$ and $f_2(u)=v_2$, (i) if u is a constant node, then v_1,v_2 are *value equivalent*, i.e., $v_1.\mathsf{val}=v_2.\mathsf{val}$; or (ii) if u is a variable node, then v_1,v_2 are *entity equivalent*, i.e., $v_1.\mathsf{eid}_u = v_2.\mathsf{eid}_u$ (eid_u is the entity identifier of u).

A scope (G,O,θ) *satisfies* φ, denoted as $(G,O,\theta) \models \varphi$, if and only if for every pair of matches $P(G,f_1)$ and $P(G,f_2)$ are bisimilar $(P(G,f_1) \sim P(G,f_2))$, and the matches are entity equivalent *w.r.t.* eid_o $(f_1(u_o).\mathsf{eid}_o = f_2(u_o).\mathsf{eid}_o)$ Intuitively, OGK $\varphi(u_o)$ enforces the requirement that "*all nodes participating in bisimilar matches satisfying X under given scope (G,O,θ) should refer to the same entity as u_o*".

OGK **Properties**. We introduce two properties of OGKs.

Non-trivial. An OGK $\varphi=(P(u_o),X)$ is *non-trivial*, if $P(u_o)$ contains u_o, and at least one variable or constant node, and X is *satisfiable*. To define satisfiability, we first define the

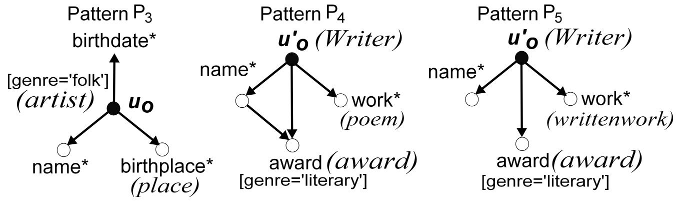


Figure 2: Ontological Graph Keys

closure of X (denoted as $\mathsf{cl}(X)$) as all literals that can be derived via transitivity of the equality relation. We can perform a fixed point inference that includes X in $\mathsf{cl}(X)$, and then adds $v.A=v'.A'$ to $\mathsf{cl}(X)$ if $v.A=c$ and $v'.A'=c$ in $\mathsf{cl}(X)$, or $v.A=v''.A''$ and $v''.A''=v'.A'$ are both in $\mathsf{cl}(X)$, until $\mathsf{cl}(X)$ no longer changes. We say X is satisfiable if there is no pair ($v.A=c$, $v.A=c'$) in X such that $c \neq c'$.

Well-defined. A set of OGKs Σ is *well-defined*, if for every OGK $\varphi(u_o)=(P(u_o),X)$ in Σ, and every variable node u' in $P(u_o)$, there exists an OGK $\varphi(u') \in \Sigma$. Henceforth, we consider well-defined Σ that contain nontrivial OGKs. We denote $\varphi(u_o)$ as φ, when u_o is clear.

Example 6: The two pattern constraints in Figure 1 extend to two OGKs: (1) $\varphi_1=(P_1(\textit{song}), \textit{song.genre}=\textit{pop})$, and (2) $\varphi_2=(P_2(\textit{album}),\emptyset)$. φ_1 states that if two nodes *ontologically* match *song*, with the same song name, and refer to the same *album* entity, then they are equivalent songs. Figure 2 shows three additional OGKs φ_3, φ_4 and φ_5, with entity patterns P_3(*artist*), and P_4, P_5 referring to entity *Writer*, respectively. When φ_2 and φ_3 both exist in Σ, the *artist* node in P_2 necessarily becomes a variable node. □

Relationship to other dependencies. We highlight the relationship between OGKs and other graph dependencies. For an OGK φ with $\theta=1$, ontology O is $\emptyset$ (f only enforces label equality), and all eids refer to node identifiers, φ can be considered as: (i) a general case of keys for graphs that only enforces X [16]; (ii) a special case of graph functional dependencies (GFDs), by "duplicating" its pattern P to $P(u_o)$ and $P'(u'_o)$ via graph isomorphism, and by enforcing attribute equality on eid (*i.e.,* $u_o.\mathsf{eid}=u'_o.\mathsf{eid}$) [18]; and (iii) a special case of graph entity dependencies (GEDs), which subsume GFDs, using more general graph homomorphism [17].

Example 7: OGKs φ_3 and φ_4 in Figure 2 define constraints for equivalent *artist* and *Writer*, respectively. Consider nodes v_4, v_5 and v_6, all referring to the name *Bob Dylan*, where (v_4,v_5) are entity equivalent *w.r.t. artist*, and (v_5,v_6) are entity equivalent *w.r.t. writer*. Node v_5 is both a *Writer* and *artist*, thereby enforcing eid_{artist} and eid_{writer}. Existing graph keys enforce only node identity and cannot differentiate multiple entities encoded by a single node. □

3. REASONING ABOUT OGKS

In this section, we study four fundamental problems for a set Σ of OGKs: (1) *satisfiability*: whether Σ can be satisfied; (2) *validation*: whether a given scope (G,O,θ) satisfies Σ; (3) *implication*: given $(G,O,\theta) \models \Sigma$, an OGK $\varphi \notin \Sigma$, determine whether $\Sigma \models \varphi$; and (4) *minimality:* find a *minimal cover*, i.e., a minimal set of OGKs Σ' equivalent to Σ.

Satisfiability. A set of OGKs Σ is *satisfiable* if there is a scope (G,O,θ), such that for each OGK $\varphi(P(u_o),X)$, there exists a non-empty match of $P(u_o)$ in G, and $(G,O,\theta) \models \Sigma$.

Unlike data dependencies on graphs [18], there always exists such a scope (G, O, θ), and this problem becomes trivial.

Lemma 1: *For any finite set of* OGKs Σ*, there always exists a scope* (G, O, θ)*, such that* $(G, O, \theta) \models \Sigma$. $\square$

Intuitively, for any OGKs with satisfiable X, we can always construct matches, and construct the corresponding scope (G, O, θ) using the matching function. Moreover, OGKs only enforce equivalence over entity identifiers, independent of variables in X. We present detailed proofs in [1].

Validation. Given scope (G, O, θ), and OGKs Σ, the validation problem is to decide whether $(G, O, \theta) \models \Sigma$.

Theorem 1: *The* OGK *validation is* NP*-complete.* $\square$

Proof sketch: The hardness of OGK validation can be verified by a reduction from subgraph isomorphism with type equality. To see the upper bound, we show there is an NP algorithm that guesses a *pair* of matches $(P(G, f), P(G, f'))$ for each OGK $\varphi(u_o)$ with pattern $P(u_o)$, where $v{=}f(u_o)$, $v'{=}f(u'_o)$. The algorithm verifies in polynomial time whether (v, v') are entity equivalent *w.r.t.* u_o. $\square$

Implication. Given a set Σ of OGKs, and an OGK $\varphi \notin \Sigma$, we say Σ *implies* φ, denoted as $\Sigma \models \varphi$, if for any triple (G, O, θ), if $(G, O, \theta) \models \Sigma$, then $(G, O, \theta) \models \varphi$. Given any finite set of OGKs $\Sigma \cup \{\varphi\}$, the *implication* problem is to determine whether $\Sigma \models \varphi$.

Theorem 2: *The* OGK *implication is* NP*-complete.* $\square$

To characterize implication for OGKs, we introduce the notion of *embeddings* between two OGKs.

Embedding of OGKs. Consider two entity patterns $P(u_o) = (V_P, E_P, L_P)$, and $P'(u'_o) = (V'_P, E'_P, L'_P)$, and ontology O. Intuitively, we say that $P'(u'_o)$ is *embedded* in $P(u_o)$, if there exists a bijection ρ that maps the node type, labels, (and edges) from each node $u' \in V'_P$ (each edge $e' \in E'_P$) to a subset of nodes in V_P (edges in E_P). Formally, the bijection ρ makes a correspondence between $P(u_o)$ and $P'(u'_o)$ such that: (i) for each $u' \in V'_P$, $\mathsf{lsim}\ (f(u')) \subseteq \mathsf{lsim}(u')$; (ii) $(u'_1, u'_2) \in E'_P$ if and only if $(f(u'_1), f(u'_2)) \in E_P$; and (iii) nodes u' and $f(u')$ are the same (constant/variable) type.

An OGK $\varphi' = (P'(u'_o), X')$ is *embedded* in another OGK $\varphi = (P(u_o), X)$, denoted as $\varphi' \preceq \varphi$, if $P'(u'_o)$ is embedded in $P(u_o)$ via a mapping ρ, and $X{=}\rho(X')$. That is, we can obtain each literal $l \in X$ by renaming a literal $l' \in X'$ by setting $u'.A$ in l' to $\rho(u').A$ in l.

Lemma 2: *Given an* OGK φ *and* OGKs Σ*,* $\Sigma \models \varphi$ *if and only if there exists an* OGK $\varphi' \in \Sigma$*, such that* $\varphi' \preceq \varphi$. $\square$

Example 8: Consider OGKs φ_4 and φ_5 in Figure 2 modeled by patterns P_4 and P_5, respectively. We can verify that $\varphi_5 \preceq \varphi_4$. For any scope $(G, O, \theta) \models \varphi_5$, $(G, O, \theta) \models \varphi_4$. Intuitively, a pair of entities representing *Writer* that is identified by the less stringent constraint in φ_5, will remain equivalent under more specific conditions posed by φ_4. P_5 imposes weaker topological constraints than P_4, and its literals subsume those in P_4, e.g., *writtenwork* subsumes *poem.* $\square$

The implication analysis requires verifying the condition in Lemma 2, which is NP-complete. For each OGK $\varphi' \in \Sigma$, we must check whether $\varphi' \preceq \varphi$, and compute the subgraph isomorphism between the entity patterns in φ' and φ (further details available in [1]).

Minimality of OGKs. To reduce the cost of entity matching, it is preferable to find a *minimal* set of OGKs that are equivalent to Σ, *i.e., a minimal cover* of Σ. Given two sets of OGKs, Σ and Σ', we say Σ' *covers* Σ, if for every OGK $\varphi \in \Sigma$, $\Sigma' \models \varphi$. A *minimal cover* of Σ, is a set Σ' of OGKs such that Σ' covers Σ, and no proper subset of Σ' covers Σ.

Computing a minimal cover. Given Theorem 2, computing a minimal cover is NP-hard. For practical entity matching using OGKs, we outline an algorithm that computes a minimal cover of Σ. Based on Lemma 2, the algorithm performs the following. First, for each OGK $\varphi \in \Sigma$, we verify whether $\Sigma \backslash \{\varphi\} \models \varphi$, by checking if there exists an OGK $\varphi' \in \Sigma \backslash \{\varphi\}$, such that $\varphi' \preceq \varphi$. If so, we remove φ. We repeat the above process, until no OGK can be removed from Σ. The above algorithm takes $O(|\Sigma|^2 |P_m|^{|P_m|})$ time, where P_m (usually small) refers to the largest entity pattern from the OGKs Σ.

Our analysis above show that the enriched semantics provided by OGKs do not make these fundamental problems harder, compared to their existing counterparts that enforce node identity [16]. Our results remain intact for the special case of OGKs with label equality, and are consistent with general GFDs [18]. We present the detailed proofs in [1].

4. ENTITY MATCHING WITH OGKS

We apply OGKs to entity matching, and introduce dynamic matching that extends the Chase process [2] over graphs.

Entity graphs. We define the notion of an entity (hyper) graph that identifies nodes from a graph $G = (V, E, L, F_A)$ referring to the same entity in each hyperedge. Formally, given a set of entities $\mathcal{E} = \{u_1, \ldots, u_m\}$, and scope (G, O, θ), the *entity graph* $V_{\mathcal{E}}$ is a *hypergraph* $(V, \bigcup_{u \in \mathcal{E}} V[u])$, where $V[u]$ the quotient set of V induced by the entity equivalent relation $R(u)$. Two nodes, $(v, v') \in R(u)$ if and only if $v.\mathsf{eid}_u = v'.\mathsf{eid}_u$; (v, v') are entity equivalent *w.r.t.* u. We can obtain a *base graph* G' of $V_{\mathcal{E}}$ that models entity equivalence for each pair of nodes in the hypergraph. We obtain G' by enforcing entity identifier equivalence for each pair of nodes in $V[u] \in V_{\mathcal{E}}$ in the original graph G.

The Chase for OGKs. We characterize entity matching by extending the Chase to an entity graph $V_{\mathcal{E}}$. Consider a set of OGKs, Σ defined on a set of center entities $\mathcal{E} = \{u_1, \ldots, u_m\}$, and scope (G, O, θ). Intuitively, given an initial hypergraph containing singleton nodes in a hyperedge (for each entity u), the $\mathsf{Chase}(\Sigma, G)$ continually merges edges containing entity equivalent nodes, according to $\varphi \in \Sigma$, until no further changes are induced by φ. Specifically, we start with an initial hypergraph $V^0_{\mathcal{E}}$, where each hyperedge $[v]_u$ in $V^0_{\mathcal{E}}$ is a singleton $\{v\}$ for every entity $u \in \mathcal{E}$. Given a triple $(\varphi, (v, v'))$, where $\varphi = (P(u), X)$, for $\varphi \in \Sigma$, f (resp. f') are two ontology matchings of $P(u)$, $v = f(u)$, $v' = f'(u)$, and $P(G, f) \sim P(G, f')$, a Chase *step* of G by $(\varphi, P(G, f), P(G, f'))$ at a hypergraph graph $V^i_{\mathcal{E}}$ is

$$V^i_{\mathcal{E}} \overset{(\varphi,(v,v'))}{\Longrightarrow} V^{i+1}_{\mathcal{E}}$$

Specifically, the following two Chase rules must be satisfied:

(1) if $v.\mathsf{eid}_o$ is not in $f_A(v)$, create a new equivalence class $[v]$ with $v.\mathsf{eid}_u = c_u$, where eid_u is the entity identifier for $u \in \mathcal{E}$ with a unique value c. (Inclusion of new eids.)

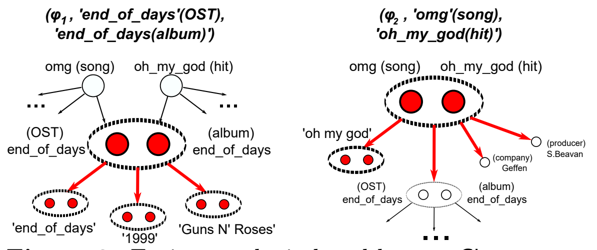


Figure 3: Entity graphs induced by two Chase steps.

(2) for all existing equivalence classes, $[v]_u$ and $[v']_u$ in $V^i_{\mathcal{E}}$ for entity u, merge $[v]_u$ and $[v']_u$ to a single equivalence class $[v]_u$, and update the set of edges $E_{\mathcal{E}}$ accordingly. (Merge hyperedges with entity equivalent nodes.)

By following these rules, the $\mathsf{Chase}(G, \Sigma)$ algorithm will generate a sequence of Chase steps that induce a sequence of hypergraphs $\{V^0_{\mathcal{E}}, \ldots, V^n_{\mathcal{E}}\}$. The $\mathsf{Chase}(G, \Sigma)$ *terminates* if there exists no $(\varphi, (v, v'))$ that elicits changes to $V^n_{\mathcal{E}}$ (no new eids, nor merges to enforce entity equality). Intuitively, $\mathsf{Chase}(G, \Sigma)$ verifies whether a set of nodes match the same pattern node via bisimilar matches. Since the matching process reduces to a Boolean function to determine whether two nodes a and b match pattern node u (via bisimilarity), a transitive closure holds over $\mathsf{Chase}(G, \Sigma)$. [1]

Example 9: Figure 3 illustrates a fraction of the entity graphs induced by two Chase steps over graph G in Figure 1 (with changed fraction marked in red). The first Chase step (φ_1, *end_of_days(OST)*, *end_of_days(album)*) creates three equivalent classes in $V^1_{\mathcal{E}}$ that merges equivalent constant nodes *e.g.,1999*, and a pair of equivalent *OST* and *album* entities. The second Chase step enforces OGK φ_2, and further merges equivalent *song* and *hit* entities given the equivalent classes in $V^1_{\mathcal{E}}$, and yields $V^2_{\mathcal{E}}$. □

We now introduce Lemma 3, which verifies that Chase with OGKs under scope (G, O, θ) preserves the *Church-Rosser property*. That is, all Chase sequences are terminating, and all terminating Chase produce the same $V^n_{\mathcal{E}}$.

Lemma 3: *Given scope (G, O, θ), (1) chasing with any set of* OGKs *Σ is finite and has the Church-Rosser property; and (2) any terminating* Chase *guarantees $(G', O, \theta) \models \Sigma$, where G' is the base graph of $V^n_{\mathcal{E}}$ when the* Chase *terminates.* □

Entity matching with Chase. Given scope (G, O, θ), and OGKs Σ, defined on a set of entities $\mathcal{E}=\{u_1, \ldots, u_m\}$, the entity matching problem is to compute the entity graph $V_{\mathcal{E}}$ induced by a terminating Chase sequence $\mathsf{Chase}(G, \Sigma)$.

This problem is, not surprisingly, NP-hard: the validation for OGKs alone is already NP-hard. Even for small OGKs, one needs to make $O(|\Sigma|N^2)$ comparisons of N matchings of a pattern in OGKs, where N is already large [9]. We show that this can be significantly reduced by effective pruning strategies given existed equivalent classes (Section 5).

5. ENTITY MATCHING ALGORITHM

Given the recursive nature of OGKs, an OGK cannot be enforced before all variable nodes are resolved. In this section, we introduce the OGK-*Entity Matching* (OGK-EM) algorithm that avoids exhaustive match enumeration of candidate pairs. OGK-EM focuses *early Chasing with non-recursive (sub)keys.* These Chase sequences (involving constant nodes) can be enforced directly, once bisimilar matches are identified, or help to prune matches for their recursive counterparts. Moreover, such sequences can be computed independently without being "blocked" by dependent entities, and thus can be computed as early as possible.

[1] "If a and b match pattern node u via bisimilar matches (i.e., a and b are equivalent), and b and c match u via bisimilar matches (b and c are equivalent), then a and c match u via bisimilar matches (a and c are equivalent)".

5.1 Dependency Graph

To model interactions among OGKs, OGK-EM maintains an auxiliary structure called *dependency graph.* Given a set of OGKs Σ, we define a dependency graph $G_\Sigma = (\Sigma, E_\Sigma)$, where (i) each node represents an OGK $\varphi(u_o)$ in Σ, and (ii) there exists an edge $(\varphi(u_o), \varphi'(u'_o)) \in E_\Sigma$ if $u'_o \in V_x$, where V_x is the set of variable nodes in $P(u_o)$. For each OGK $\varphi = (P(u_o), X)$, OGK-EM maintains: (a) match set $P(u, G)$ for each pattern node u in $P(u_o)$; (b) $V[u_o]$, a partition (hyperedge) of $P(u_o, G)$ induced by relation $R(u_o)$; and (c) a pair of boolean flags (isC, Val) that is set to true when, respectively, φ contains only constant nodes, and $(G', O, \theta) \models \varphi$ given all enforced Chase steps thus far.

For ease of presentation, we construct a directed acyclic graph (DAG) G_d from G_Σ by collapsing each strongly connected component (SCC) into a single SCC node. Abusing terms from trees, we say a *root* (resp. *leaf*) of G_d is a node without an incoming (resp. outgoing) edge. We set the constant node flag, isC = true for all leaves in G_d. Moreover, for each node v_d in G_d, a rank $r(v_d)$ is defined as: (i) $r(v_d) = 0$ if v_d is a root; (ii) $r(v_d) = \max(r(v'_d))+1$ otherwise, where v'_d ranges over the parents of v_d in G_d; and (iii) for an SCC node v_d, all nodes in v_d have the same rank $r(v_d)$.

5.2 Matching Algorithm

OGK-EM initializes entity graph, $V_{\mathcal{E}}$, with over estimated equivalence classes, where each class represents the set of nodes having concept labels l in the relevant set of entity node $u_i \in \mathcal{E}$, $l \in \mathsf{lsim}(l_{u_i})$. We dynamically refine these classes by retaining true ontological matches and splitting equivalence classes, with two major phases. First, a "top-down" *decomposition* phase decomposes each OGK to a set of *tree keys* (entity keys with tree patterns). We efficiently chase bisimilar tree matches to refine $V_{\mathcal{E}}$ by merging equivalence classes whenever possible. Second, in a "bottom-up" *synthesizing* phase, OGK-EM refines $V_{\mathcal{E}}$ by assembling bisimilar matches from the leaves of the dependency graph G_d, until all the nodes in G_d are processed.

Algorithm Phases. The main phases of OGK-EM is illustrated in Figure 4. It first constructs G_d and computes the node ranks. The decomposition and partial validation phase proceeds in ascending node rank order similar to a breadth-first traversal of G_d. The synthesizing phase starts at the leaf level proceeding in descending node rank order.

(1) *Top-down Decomposition.* For each root $\varphi = (P(u_o), X)$ in G_d, OGK-EM initializes the match sets $P(u, G)$ for each node u in $P(u_o)$ as $\{v | L(v) \in \mathsf{lsim}(L_P(u))\}$. It initializes $V[u_o]$ as $\{[v] | [v] = \{v\}\}$ for each $v \in P(u_o, G)$. The procedure Decomp decomposes φ to a set of *tree keys* $\mathcal{P}_\varphi = \{\varphi_1, \ldots, \varphi_n\}$. Each tree key $\varphi_i = (P_i(u), X_i)$ contains: (i) a tree-structured pattern $P_i(u)$ centered on entity u; and (ii) $X_i \subseteq X$ with literals involving pattern nodes in P_i only. Intuitively, these tree patterns constitute a tree cover of $P(u_o)$ with shared pattern node u_o, and $\bigcup_i^n X_i = X$.

Algorithm OGK − EM

Input: a set of OGKs Σ over entities $\mathcal{E}$, scope (G, O, θ);
Output: the entity graph $V_\mathcal{E}$ induced by $\mathsf{Chase}(\Sigma, G)$.

```
1.  initializes $V[u_i]$ ($u_i \in \mathcal{E}$); integer $i$=0;
2.  construct $G_d$; integer $r_m$:=max($r(v_d)$) ($v_d$ in $G_d$);
3.  while $i < r_m$ do /* "top-down" phase*/
4.    $\Sigma_i := \{\varphi | r(\varphi) = i\}$;
5.    for each $\varphi \in \Sigma_i$ do
6.      $\mathcal{P}_\varphi$ := Decomp($\varphi$); /*spawning tree keys*/
7.      $P(u_o, G)$ := PVal($\mathcal{P}_\varphi$); $i := i + 1$; /*partial validation;*/
8.  while $i >= 0$ do /* "bottom-up" phase*/
9.    $\Sigma_i := \{\varphi | r(\varphi) = i\}$; $i := i - 1$;
10.   for each $\varphi(u_o) \in \Sigma_i$ do
11.     $V[u_o]$:= SVal($\varphi, G_d$);
12. $V_\mathcal{E} := \bigcup_{u_i \in \mathcal{E}} V[u_i]$;
13. return $V_\mathcal{E}$;
```

Figure 4: Algorithm OGK-EM

(2) *Partial Validation.* Given a set of tree keys $\mathcal{P}_\varphi = \{\varphi_1, \ldots, \varphi_n\}$ of $\varphi = (P(u_o), X)$, OGK-EM first partitions constant tree keys $\mathcal{P}_c \subseteq \mathcal{P}_\varphi$, from variable tree keys $\mathcal{P}_v = \mathcal{P}_\varphi \setminus \mathcal{P}_c$, which contain at least a variable node besides u_o (via procedure PVal). We then partially validate each tree key and refine the match sets to update $V[u_o]$. Specifically, for each constant tree key φ_i, PVal sets its flag isC = true. Next, for each pattern node u, we refine match set $P(u, G)$ as $\bigcap_{i=1}^n P_i(u, G)$, from all tree keys in $\mathcal{P}_\varphi$ containing pattern node u. For each child $\varphi' = (P'(u'_o), X')$ of φ in G_Σ, we initialize the match set $P'(u'_o, G)$ with the refined match set of the corresponding variable node in P. Lastly, this decomposition and partial validation process is then repeated for all children of φ following a breadth-first traversal of G_d.

(3) *Bottom up Synthesizing.* After completing the top-down decomposition where all leaves in G_d are processed, OGK-EM traverses G_d "bottom up" from the leaves to the root. For each node $\varphi = (P(u_o), X)$ in G_d, visited in descending node rank, OGK-EM invokes procedure SVal to compute $P(u_o, G)$ and the bisimilar matches, to update $V[u_o]$ by enforcing equivalence classes. For each tree key φ, we refine its match set by using the match sets from its children. We update $V[u_o] \in V_\mathcal{E}$ by iteratively merging two equivalence classes ($[v], [v']$) induced by pairs (v, v') from bisimilar matches $P(u_o, G, f)$ and $P(u_o, G, f')$, where each pair simulates a Chase step. When we can no longer enforce such pairs, OGK-EM sets Val=true for φ. The bottom up traversal terminates once Val=true for all root nodes in G_d, *i.e.,* the enforced base graph G', $G \models \Sigma$.

We illustrate the running of top-down phase below and present the details of Decomp and PVal in [1].

Example 10: Consider the OGKs φ, φ' and φ'' for entities *album, producer*, and *band* in Figure 5. A partial dependency graph G_d is illustrated with two (dotted) edges (φ, φ') and (φ, φ''), for entities *producer* and *band*, respectively. In the top-down phase, Decomp decomposes φ to two tree keys with patterns P_1 and P_2. Their common matches, computed by PVal, refine the match set of P to $\{a_2, \ldots, a_5\}$. The process continues following G_d until it reaches leaves. □

Procedure SVal. Once the traversal reaches the leaf level (each interior variable node has been decomposed into its own tree keys, and every child φ' of φ in G_d has φ'.Val = true), SVal verifies the bisimilar matches for $P(u_o, G)$ with the following invocation cases (Figure 6).

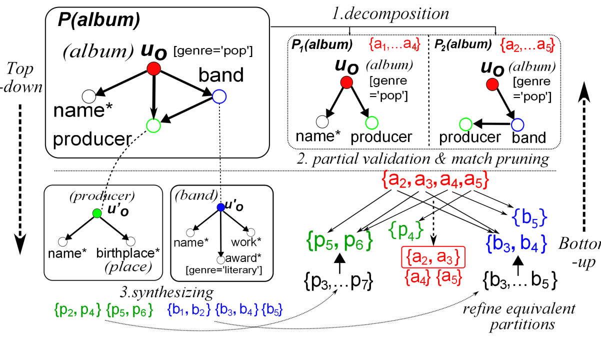

P(album)
(album) u_o [genre='pop']
band
name*
producer
Top -down
1.decomposition
P_1(album) $\{a_1, \ldots a_4\}$
u_o (album) [genre ='pop']
name* producer
P_2(album) $\{a_2, \ldots a_5\}$
u_o (album) [genre ='pop']
producer band
2. partial validation & match pruning
(producer) u'_o
name* birthplace* (place)
(band) u'_o
name* work* award* [genre='literary']
3.synthesizing
$\{p_2, p_4\}$ $\{p_5, p_6\}$ $\{b_1, b_2\}$ $\{b_3, b_4\}$ $\{b_5\}$
$\{a_2, a_3, a_4, a_5\}$
$\{p_5, p_6\}$ $\{p_4\}$ $\{b_5\}$ $\{b_3, b_4\}$
$\{a_2, a_3\}$ $\{a_4\}$ $\{a_5\}$
$\{p_3, \ldots p_7\}$ $\{b_3, \ldots b_5\}$
Bottom -up
refine equivalent partitions

Figure 5: OGK-EM: Entity matching and pruning.

$\underline{\varphi \text{ is a leaf.}}$ For $\varphi = (P(u_o), X)$, PVal computes matches by verifying for each pair (v, v') from $P(u_o, G)$, whether there exists a pair of bisimilar matches $P(G, f) \sim P(G, f')$. To avoid enumerating all pairwise comparisons, we extend the VF2 algorithm with backtracking [12] to consider partial matches (v, v') with an added feasibility condition: for all (v_1, v_2) that match a constant node u in $P(u_o)$, v_1.val = v_2.val. If (v, v') are induced by bisimilar matches, we merge $[v]$ and $[v']$ in $V[u_o]$. After all pairs have been evaluated, SVal sets φ.Val = true in G_d. We further optimize the matching by using an ontology index [36] (see details in [1]).

Leveraging children equivalence classes. If each child $\varphi'(u'_o)$ of $\varphi(u_o)$ in G_d has φ'.Val = true, then no further equivalence classes for $\varphi'(u'_o)$ can be merged from bisimilar matches. SVal uses these equivalence classes to refine $V_\mathcal{E}(u_o)$. First, for each variable node $u'_o \in V_x$ of $P(u_o)$, SVal induces a *local view* $V_\varphi[u'_o]$ of enforced equivalence classes $V[u'_o]$, where $V_\varphi[u'_o]$ is defined as $\bigcup_{v[u] \in V[u]} v[u] \cap P(u, G)$, *i.e.,* the partitions $V[u]$ that are induced by nodes in $P(u'_o, G)$. Second, SVal then constructs an equivalence relation $R(\varphi, u_o)$, where a pair of nodes $(v_o, v'_o) \in R(\varphi, u_o)$ if the following hold:

- $\{v_o, v'_o\} \subseteq P(u_o, G)$, and
- for each variable node $u_x \in V_x$, there exists v and v' such that $(v, v') \in V_\varphi[u'_o]$, and v_o, v (resp. v'_o, v') are from the same match $P_i(G, f)$ (resp. $P_j(G, f')$) of at least a tree key P_i (resp. P_j) of $P(u_o)$.

The relation $R(\varphi, u_o)$ partitions $V_\varphi[u_o]$ of $P(u_o, G)$, and SVal verifies each pair (v_o, v'_o) using only equivalence classes $v_\varphi[u] \in V_\varphi[u_o]$, *without* having to do pairwise comparisons from $P(u_o, G)$ (using the extended VF2 algorithm). Lastly, we update $V[u_o]$ to enforce equivalence classes of u_o, and set φ.Val = true. Lemma 4 shows that the reduced number of verification steps continues to preserve correctness.

Lemma 4: *For any* OGK $\varphi(u_o) \in \Sigma$, *and any entity equivalent pairs* $(v, v') \in R(u_o)$, $(v, v') \in R(\varphi, u_o)$. □

Early termination. Given Lemma 4, OGK-EM terminates the matching process early for φ whenever $R(\varphi, u_o)$ is an identity relation without verification. Indeed, entity graph $V_\mathcal{E}$ will remain unchanged for all such OGKs. The estimated $V[u_o]$ (with nodes from $P(u_o, G)$ that may contain non-matches) is used to prune the match sets of its parents.

For an SCC node v_d in G_d that contains multiple OGKs, OGK-EM resolves equivalence classes in a similar manner but conducts a fixpoint computation (see details in [1]).

Example 11: Continuing our example in Figure 5, in the bottom-up phase, a set of equivalence classes for *producer* $(\{\{p_2, p_4\}, \{p_5, p_6\}\})$ and *band* $(\{\{b_1, b_2\}, \{b_3, b_4\}, \{b_5\}\})$ are

Procedure $\mathsf{SVal}(\varphi(u_o), G_d)$

```
1.  if $V[u_o] = \emptyset$ then $V[u_o] := \{[v] | [v] = \{v\}, v \in P(u_o, G)\}$;
2.  if $r(\varphi) = r_m$ then /*a leaf node in $G_d$*/
3.    for each $(v_o, v'_o) \in P(u_o, G)$ do
4.      if (Verify($v_o, v'_o, u_o$)=true) then
5.        $[v_o]_{u_o} := [v'_o]_{u_o} \cup [v'_o]_{u_o}$;
6.        enforce equality of $\mathsf{eid}_o$ on $[v_o]_{u_o}$;
7.        update $V[u_o]$ with $[v_o]_{u_o}$; break;
8.  if $r(\varphi) < r_m$ then
9.    for each variable node $u'_o \in V_x$ do
10.     compute $V_\varphi[u'_o]$ and $V_\varphi[u_o]$ ($R(\varphi, u_o)$);
11.     for each $v_\varphi[u_o] \in V_\varphi[u_o]$ and $(v, v') \in v_\varphi[u_o]$do
12.       if (Verify($v, v', u_o$)=true) then
13.         merge $[v]_{u_o}$ and $[v']_{u_o}$; update $V[u_o]$;
14. return $V[u_o]$;
```

Figure 6: Procedure SVal

derived from the two children of OGK φ, respectively. For the node *producer* in φ with potential matches $\{p_3, \ldots, p_7\}$, SVal first induces a local equivalence partition given that all equivalent *producer* entities are known by enforcing φ'. This induces a non-singleton equivalence class $\{p_5, p_6\}$ and a singleton $\{p_4\}$. Similarly, it induces local equivalence classes $\{b_3, b_4\}$ and $\{b_5\}$ derived from φ'' for *band*.

The match set for *album* $\{a_2, \ldots, a_5\}$ (obtained in the top-down phase; Example 10) is then refined to $\{\{a_2, a_3\}, \{a_4\}, \{a_5\}\}$. Specifically, (1) a_2 and a_3 have a child p_5 and p_6 respectively, both from a same equivalent class for *producer*; and share a same child b_3 (*band*); (2) a_4 and a_5 each has children either from *producer* or *band* that distinguish them from equivalent entities. Thus, only a single pair of entities $\{a_2, a_3\}$ need to be verified for entity equivalence. □

Analysis. OGK-EM correctly enforces Σ. Indeed, $G' \models \Sigma$ when OGK-EM terminates, where G' is the base graph of the produced entity graph $V_\mathcal{E}$, ensured by the correctness of PVal and SVal. Let the set $C(u_o)$ be $\{v | L(v) \in \mathsf{lsim}(u_o), v \in V\}$ under scope (G, O, θ), d be the maximum diameter of a pattern in Σ, and C_m (resp. P_m) refers to the largest $C(u_o)$ (resp. $P(u_o)$ in Σ). It takes $O(|\Sigma|^2)$ time to construct G_d, and $O(|N_d(C(u_o))|^{2|P(u_o)|})$ time to identify equivalent pairs, where $N_d(C(u_o))$ are the d-hop neighbors of $C(u_o)$. Thus, OGK-EM takes $O(|\Sigma|^2 + |\Sigma||N_d(C_m)|^{|P_m|})$ time. In practice, $|\Sigma|$, $|P_m|$ and d are small (see Section 7).

6. BUDGETED ENTITY MATCHING

Resources are often limited in practice, and constraints are imposed to minimize the effort and cost to perform entity matching [29]. We introduce a budgeted version of OGK-EM that performs entity matching with bounded matching cost.

We start with a cost model for Chase with OGKs.

Cost Model. Given OGK $\varphi = (P(u_o), X)$, scope (G, O, θ), let $\mathsf{Chase}(i) = (\mathcal{G}^i \overset{(\varphi, f)}{\Longrightarrow} \mathcal{G}^{i+1})$, represent a single Chase step. We define the cost $c(\mathsf{Chase}(i))$ of a single Chase step as $c(u_o, f(u_o))$, which is the cost to match u_o and $f(u_o)$. For a Chase sequence, $\mathsf{Chase}(ij)$, we define $\rho = (\mathcal{G}^i \overset{(\varphi, (v_1, v_2))}{\Longrightarrow} \ldots \overset{(\varphi', (v'_1, v'_2))}{\Longrightarrow} \mathcal{G}^j)$, and the cost is computed as

$$c(\rho) = c(\mathcal{G}^i, \mathcal{G}^{i+1}) \cdot \sum_{i}^{j-1} c(\mathsf{Chase}(i))$$

where $c(\mathcal{G}^i, \mathcal{G}^j) = \frac{|U|}{|V|}$, and U refers to the total number of entity identifiers (v.eid) updated in $\mathcal{G}^i$ to enforce the Chase

Algorithm $\mathsf{BOGK-EM}$

Input: dependency graph G_d, scope (G, O, θ); budget B; beam size b;
Output: the entity graph $V_\mathcal{E}$ under budget B.

```
1.  initializes $V[u_i]$ ($u_i \in \mathcal{E}$);
2.  integer $r := l + 1$ (l: the maximum node rank of $G_d$);
3.  set $\pi* := \emptyset$; set $\pi := \emptyset$; stack $S$.push $(r, 0, B)$; cost $U$:=0;
4.  while $S.top() \neq \emptyset$ do
5.    $\pi$:= Bchase $(\pi, r, S, b, G_d)$;/*compute a Chase under budget*/
6.    if $\pi \neq \emptyset$ then
7.      $\pi^* = \pi$; $U$:= $\pi$.cost; /* current optimal chase*/
        /* set new cost range to explore Chase */
8.    while $S.top()$.cmin $\geq U$ do $S$.pop();
9.    $S$.top().cmin:= $S$.top.cmax; $S$.top().cmax:= $U$;
10.   if $S$=$\emptyset$ then
11.     construct $V_\mathcal{E}$ by enforcing sequence $\pi^*$;
12.     return $V_\mathcal{E}$.
```

Figure 7: Algorithm BOGK-EM

rules. Intuitively, we measure the cost to enforce Σ, which requires merging and transforming nodes in $\mathcal{G}^i$ to those in $\mathcal{G}^j$, and includes the matching cost of each Chase step.

Budgeted Entity Matching Problem. Given a set of OGKs Σ, scope (G, O, θ), and a budget B, we want to compute a Chase sequence ρ that generates a $\mathcal{G}$, such that ρ has a smallest cost $c(\rho)$ bounded by B.

Our goal is consistent with evaluating entity resolution with "merging" cost of entities [29], while (1) $c(\rho)$ aggregates the editing cost weighted by ontological matching cost; and (2) B encodes a threshold to distinguish "good" entity matching results and infeasible ones. Intuitively, we identify feasible entity matching result with a minimum cost. Budgeted entity matching generalizes the entity matching problem in Section 4: the latter carries unit Chase step cost with $B = \infty$, This leads to the following result.

Lemma 5: *Budgeted matching with* OGKs *is* NP-*hard.* □

To find a Chase with minimal cost, we model this as a *planning* problem that outputs a sequence of OGKs $\pi = \{\varphi_1, \ldots, \varphi_n\}$ to be enforced with minimum cost. We show that the search can be optimized by revising OGK-EM with *beam search and backtracking.*

Overview. We introduce a *budgeted* version of OGK-EM called BOGK-EM, optimized by *beam search with backtracking.* Beam search is a heuristic optimization of breadth-first search that traverses a search tree by expanding and exploring the most promising nodes, up to a fixed number b, called the *beam size.* BOGK-EM follows beam search to select the top-b "best" OGKs following node ranks in dependency graph, but dynamically reduces the allowed budget according to current best solution, and backtracks to explore alternative Chase sequences. Moreover, BOGK-EM dynamically estimates an upper bound of Chase cost to prioritize the selection of promising beam elements as OGKs.

Auxiliary structures. BOGK-EM uses the dependency graph G_d to coordinate the search. For each node v_d in G_d, it extends the auxiliary information of v_d to a vector $\{\mathsf{isC}, \mathsf{Val}, U\}$, where U is an estimate of the additional Chase cost to enforce all OGKs in v_d. At each layer i, BOGK-EM records: (1) an open set open(i) of candidate OGKs to be explored; (2) a set $\mathsf{L} \subseteq$ open(i) of OGKs to be validated and enforced with tunable memory size; and (3) a stack S containing values (i, c_{min}, c_{max}) that define the allocated cost range to OGKs (open(i)) evaluated at layer i.

Algorithm. The algorithm BOGK-EM is illustrated in Figure 7. Let the leaves in G_d be of rank l, and v_s be a pseudo node of rank $l+1$, connecting to the leaf nodes. BOGK-EM initializes the auxiliary structures, as well as the current best Chase and the newly constructed Chase with π^* and π, respectively (lines 1-3). It then executes a layered, beam search with backtracking starting at v_s, invoking procedure Bchase (lines 4-9) to update π^*. It then enforces π^* to perform entity matching (lines 10-12).

We next describe procedure Bchase (see details in [1]).

Beam selection. At each layer i, Bchase evaluates all nodes in G_d with rank i. If $i = (l+1)$, we initialize the candidate set of OGKs to explore, open(l), to all the leaves in G_d with rank l, set OGKs π and π^* to $\emptyset$, and push $(l, 0, B)$ to stack S. For each candidate node v_d in open($i-1$), representing an OGK $\varphi = (P(u_o), X \to l_o)$, we compute the matches for each node u in $P(u_o)$ using Decomp and PVal. We derive an upper bound of the matching cost $v_d.U$ as

$$v_d.U = \frac{|P(u_o, G)|}{|V|} \max_{v_o \in P(u_o, G)} \hat{c}(u_o, v_o)$$

where $\hat{c}(u_o, v_o)$ is an overestimated cost, computed as

$$\hat{c}(u_o, v_o) = \frac{1}{|V_P|} \sum_{u' \in V_P} \max_{v' \in P(u', G)} c_r(u', v')$$

We overestimate the matching cost by assuming that every pruned node in Decomp and PVal are indeed matches. The cost of an SCC node v_d is computed similarly, as the sum of the estimated cost for each OGK $\varphi \in v_d$.

Bchase initializes L with the b OGKs in open(l) of smallest cost within the range (S.*top*().cmin, S.*top*().cmax]. For beam selection when backtracking to layer i, we define the cost range for the next batch of OGKs by pushing a triple (i, $\max_{v_d \in \mathsf{L}}(v_d.U)$, B) onto the stack S.

Backtracking. We refine each OGK in L($i-1$) by refining its matches using procedure SVal. We also update $v_d.U$ to its true cost B', and add the selected OGK to the current Chase π. We must update the list of candidates for the next level, open($i-2$), as the set of nodes in G_d with validated children (using flag Val), and deduct B' from the current budget B. If $B > 0$, and candidates remain in open($i-2$) or open($i-1$), Bchase will continue processing the next layer $i-2$. Otherwise, we set the optimal Chase $\pi^* = \pi$, if $\pi^*=\emptyset$, or $c(\pi) < c(\pi^*)$. We backtrack to layer $i-1$, to populate the cache L($i-1$) with the b OGKs of lowest cost $v_d.U \in$ (S.*top*().cmin, S.*top*().cmax), and update the stack S. After processing all candidate OGKs in layer $i-1$, we set open($i-1$) $= \emptyset$. BOGK-EM terminates when open(l) is $\emptyset$, and enforces the equivalence classes for each OGK in π^*.

Example 12: Consider the dependency graph G_d in Figure 8. Let the beam size $b = 2$ and a budget $B = 9$, BOGK-EM computes a budgeted Chase as follows. (1) Starting from the pseudo node v_s (φ_0), it initializes the stack S with $(3, 0, 9)$, stating "Chase with cost in [0,9] will be explored". As the costs of φ_1, φ_2, and φ_3 are validated to be 3, 4 and 5 respectively and b=2, It sets open (2) as $\{\varphi_1, \varphi_2, \varphi_3\}$, $L(2)$ = $\{\varphi_1, \varphi_2\}$, and updates $S.top()$ to be $(3, 0, 5)$, to prevent traversing Chase with cost less than 5 when backtracking to level 2. (2) At level 2, BOGK-EM pushes $(2, 0, 9)$ to S, identifies $L(2)$ to be $\{\varphi_4, \varphi_5\}$, and updates the top of S to $(2, 0, 6)$ similarly. (3) At level 3, it constructs the current best Chase

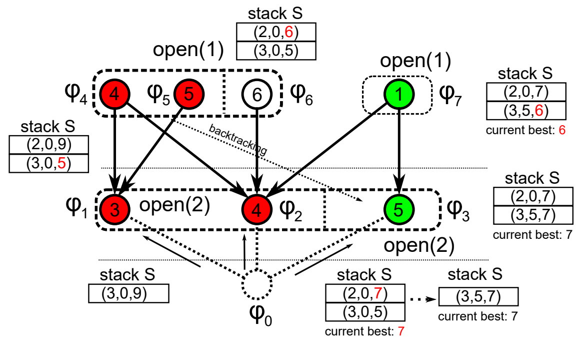


Figure 8: BOGK-EM: budgeted Chase with backtracking.

$\{\varphi_1, \varphi_4\}$. As the only successor of φ_4 exceeds the budget, it backtracks to level 1. (4) Given the current minimum cost 7, it updates $S.top()$.cmin to 5, and $S.top()$.cmax to 7, "switching" to explore Chase with cost in [5, 7]. This finally yields a better Chase $\{\varphi_5, \varphi_7\}$ with cost 6. □

Analysis. BOGK-EM is an anytime algorithm that can return the current Chase π upon request, and can continue to refine π if needed, with decreasing incremental cost to generate new solutions. It is also memory efficient, with the memory cost in $O(2b \cdot (N_d(C_m) + |P_m||C_m|) + |\Sigma|)$, where d is the diameter of P_m (see [1] for detailed analysis).

7. EXPERIMENTAL STUDY

We use real graphs and ontologies to evaluate: (1) the efficiency of entity matching using OGKs; (2) the effectiveness of entity matching (with ground truth), and the trade-off between effectiveness and efficiency (using injected redundancy); and (3) case studies for real entity matching.

Experimental Setup. We implement all our algorithms in Java v8, and ran our experiments on a Linux machine with AMD 2.7 GHz CPU with 256 GB of memory.

Datasets. We use two real benchmark datasets containing the ground truth to evaluate the efficiency and effectiveness of our techniques. We also develop a data generator to inject duplicates into one of the benchmark datasets to evaluate scalability. Table 2 summarizes the data characteristics.

DBpedia-Yago[2]: this benchmark graph (used in [35]) contains 50,248 verified ground truth entity pairs with aligned properties between DBpedia and Yago. These equivalent pairs cover 10 types of entities. To create a "hybrid" ontology, we added "is A" relations between two types for each ground truth entity pair, one from the DBpedia ontology, and the other from the Yago ontology.

DBpedia-IMDb[3]: this second benchmark graph (used in [15]) contains 33,437 entities covering 10 types, totaling 9,515 redundant pairs between DBpedia and IMDb. We create the corresponding ontology that links entities between DBpedia and IMDb following a similar process for DBpedia-Yago.

DBpYago-Dup*:* To evaluate the trade-off between efficiency and effectiveness, we create a data generator that injects a controlled number of duplicate entity pairs, facts, and labels into DBpedia-Yago, and its corresponding ontology. The duplicate entries are duplicated from the ground truth entities

[2]https://github.com/lgalarra/vickey

[3]https://www.csd.uoc.gr/~vefthym/minoanER/datasets.html

Table 2: Data characteristics.

Dataset	#entities	#triples	#labels	#duplicates
DBpedia-Yago	592K	4.5M	10	50248
DBpedia-IMDb	33K	200K	10	9515
DBpYago-Dup	4.6M	29M	935	65248

with varied labels from the original ontology. The generator injects pairs of "seed" equivalent entity pairs (v, v'), and duplicates v and v' in DBpedia and Yago respectively, along with their neighborhood up to 3 hops. The generator then disturbs the entity types of these duplicates to a concept label in the ontology, and randomly updates 50% of the literals of duplicated entities. We then identify a set of true examples Γ^+ (containing equivalent entity pairs), and false examples Γ^- (that do not refer to the same entity as a result of injecting noise into the labels and literals).

We use the two benchmark graphs to evaluate the comparative accuracy of OGK based techniques against the baselines, and the controllable DBpYago-Dup to verify efficiency.

OGKs Generation. We extend the key mining algorithm, KeyMiner [4] to generate OGKs. Given ground truth, it discovers OGKs via level-wise graph pattern mining to identify maximal entity patterns and literals that preserve bisimilar matches for entity equivalent pairs. We compute OGKs using a minimum support of 80%, with confidence level of at least 90% over the matched entities. For each dataset, we sample 40% of the examples as a training dataset. We validate the keys over 10% of the examples (as a validation dataset), and retain those with desirable accuracy. Using these OGKs, we apply entity matching over the remaining 50% of the dataset as a test dataset. We used hyponym edges to identify entity types, and equivalence and descriptive edges to identify concept labels, to be assigned to the OGK patterns. We extracted 10, 8, and 250 OGKs covering at least 40K, 7.6K and 52K of the ground truth in DBpedia-Yago, DBpedia-IMDb and DBpYago-Dup, respectively. We retain the OGKs with high support and confidence conditioned by proper literals. The OGKs for DBpYago-Dup are generated over the examples Γ^+ and Γ^-.

Algorithms. We implemented the following methods.

OGK-EM: our exact entity matching algorithm (Section 5).

EnumEM: a variant of OGK-EM without the top-down phase that follows a bottom-up strategy to perform pairwise verification of bisimilar matches without hypergraph refinement.

GK-EM: a variant of EnumEM that simulates graph key-based entity matching [16], by enforcing type equality without literals, and merges nodes instead of entity identifiers.

BOGK-EM: our weighted entity matching algorithm. We compare BOGK-EM against BEnumEM and BGK-EM, which are the budgeted versions that uses pairwise verification without hypergraph refinement, and enforces type equality, respectively. For pattern matching, we implement an algorithm that extends VF2 with ontology similarity [12].

Vickey [35] detects conditional keys via breadth-first search using logical rules with strict label equality. It considers neither topological nor ontological similarity.

Holistic [33] performs graph alignment by personalized page-rank from seed entities, and expands subgraphs (modeled as "pair graphs") to capture the impact of correlated entities.

We set a support threshold 2% for Vickey, and a similarity threshold 0.85 for Holistic to assert equivalence. We choose these settings to favor both methods under which they achieve the highest average precision and recall. Our source code and test cases are all available online[4].

Experimental Results. We next report our findings. We first evaluate the efficiency of OGK-EM and BOGK-EM against EnumEM, GK-EM, and BEnumEM, BGK-EM. Our objective is to evaluate the effectiveness of our optimizations on runtime, and the overhead of enriched key semantics.

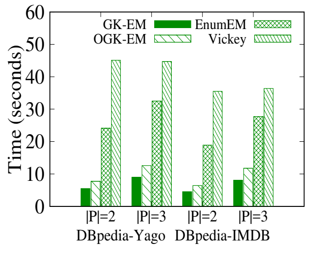
Exp-1: Efficiency of OGK-EM. Using DBpedia-Yago, we defined 10 OGKs. We set $\mathsf{lsim}(l)$, and θ to include similar labels that are within 2 hops of l in the corresponding ontologies. The center nodes have on average 12000 matches. Figure 9a reports the time of entity matching and the impact of pattern size (number of edges). (1) It is quite feasible to identify equivalent entities over real-world graphs using OGK-EM. For example, it takes on average 10 seconds to enforce OGKs Σ over DBpedia-Yago, and 9.1 seconds over DBpedia-IMDb. (2) OGK-EM outperforms EnumEM by 2.4 times, and has comparable performance with GK-EM. This is notable since GK-EM enforces only label equivalence, thereby inspecting fewer entities than OGK-EM. (3) OGK-EM outperforms Vickey by 4.2 times on average. While the major bottleneck for both methods are the pairwise comparisons of entities, OGK-EM performs less comparisons due to an enriched OGK semantics and pruning strategy. Vickey identifies relatively more entity pairs that must be compared. We also found that Holistic cannot run to completion after 350 seconds. The major bottleneck is the construction of auxiliary structures, such as pair graphs.

Exp-2: Scalability. Using DBpYago-Dup, we evaluate the scalability of our algorithms by varying: the number of variable nodes ($|V_x|$), thresholds (α, θ), recursion depth (r_m), the number of OGKs ($|\Sigma|$), graph size ($|G|=(|V|, |E|)$), budget (B), and beam size (b). We set $|\Sigma| = 10$, $|V_x| = 2$, $|G| = (16M, 20M)$, $r_m = 2$, and $(\alpha, \theta) = (2, 0.5)$ by default.

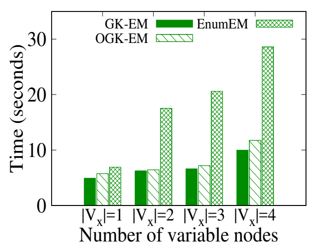
Varying variable size $|V_x|$. Figure 9b reports the impact of the number of variable nodes $|V_x|$ (varied from 1 to 4 by selecting corresponding OGKs groups) to the performance of OGK-EM. While all algorithms take more time for OGKs with larger variable nodes, OGK-EM is less sensitive compared to EnumEM and GK-EM, due to its aggressive pruning that reduce the pairwise comparison costs. More equivalence classes can also be refined during the "bottom-up" phase over larger $|V_x|$, further reducing verification. On average, OGK-EM outperforms EnumEM by 2 times.

Varying (α, θ). We varied α from 1 to 5 and θ from 0.1 to 0.5. Given a pair of equivalent entities from the positive examples (benchmark data), we calculate the distance between their labels in the corresponding ontology. We observe that 5 is an "upper bound" of α in order to identify ontologically similar entity types for most cases. Figure 9c shows that OGK-EM scales well as OGKs relax strict equality matching to allow approximate entity matching using similar concept labels by tuning θ and α. OGK-EM leads to longer running times due to verifying more matched entities. It is more sensitive to α comparing with θ, as larger α indicates more entities need to be verified. GK-EM lacks such flexibility due to strict label equality.

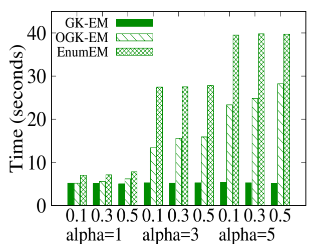
[4]https://github.com/HanchaoMaWSU/OGKEM.git



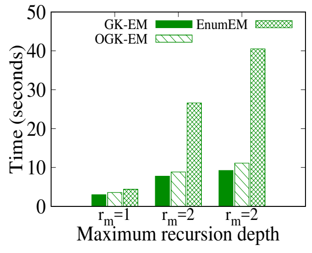
(a) Efficiency (Benchmark)



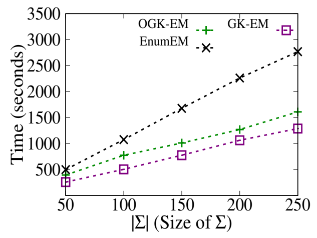
(b) Varying $|V_x|$



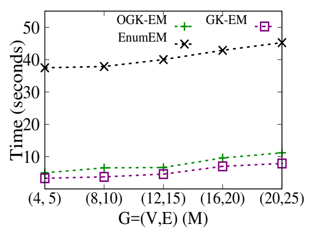
(c) Varying (α, θ)



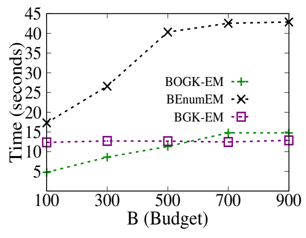
(d) Varying r_m



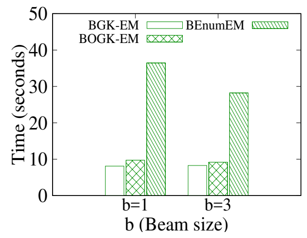
(e) Varying $|\Sigma|$



(f) Varying $|G|$



(g) Varying B



(h) Varying b

Figure 9: Efficiency of Entity Matching using OGKs.

Varying recursion r_m. We varied the OGK recursion depth in Σ (*i.e.,* the maximum node rank r_m in dependency graph G_d) from 1 to 3. This occurs when new references are identified among the entities in Σ. Figure 9d shows increased runtimes for all algorithms as deeper recursion r_m is enforced. However, OGK-EM terminates earlier for larger r_m, due to greater refinement of equivalence classes.

Varying $|\Sigma|$ and $|G|$. Figs 9e and 9f verify that all methods take longer time for larger $|\Sigma|$ and $|G|$. Specifically, OGK-EM outperforms EnumEM by 1.7 times and 4.8 times when $|\Sigma|$ = 250 and $|G|$ is varied to $(20M, 25M)$, respectively. OGK-EM has a reduced number of comparisons, and comparable runtime with GK-EM, which only enforces label equality.

We now evaluate the anytime BOGK-EM for budgeted entity matching, and compare it with BEnumEM and BGK-EM. For a fair comparison, we report the runtime for each algorithm to converge to the best Chase for a given budget B.

Varying budget B. Figure 9g shows that all three budgeted algorithms take more time as we vary B from 100 - 900, due to additional verification from more merge operations. BOGK-EM outperforms BEnumEM by 5 times on average.

Varying beam size b. Fixing $|G| = (16M, 20M)$, Figure 9h shows that larger beam sizes allow BOGK-EM to prioritize Chase and select more promising OGKs first. Aggressive pruning can also occur during the bottom-up phase due to more OGKs at the leaf level. BGK-EM is not affected by b since it does not execute a beam search.

Exp-3: Impact of Parameters to Accuracy. We investigate the impact of the thresholds (α, θ) and the budget B using DBpYago-Dup. We injected 15K true and false examples each to DBpYago-Dup ($|\Gamma^+| = |\Gamma^-| = 15$K). By default, we set $\alpha = 3$, $\theta = 0.8$, and $B = 500$.

Varying threshold pair (α, θ). Recall that larger values of α and θ relax the strict equality conditions for label and ontological matching, respectively, e.g., label equality is enforced at $\alpha = 0$. By allowing this flexible similarity matching, we achieve precision gains over existing techniques. For example, OGK-EM achieves an 18% gain in precision on average over GK-EM by capturing more correct entities via ontological matching (not shown). Figure 10a shows that as (α, θ) increase, precision decreases, as more weakly related entities (false positive examples) are captured due to approximate matching. We also verify this trend as fpr increases for larger (α, θ) values. In contrast, Figure 10b shows that increasing α and θ improves recall. We observe a 53% gain in recall, and a 25% gain in F_1 score when (α, θ) is varied from (1, 0.4) to (3, 1), since more true positive entities are captured by OGKs via ontological matching.

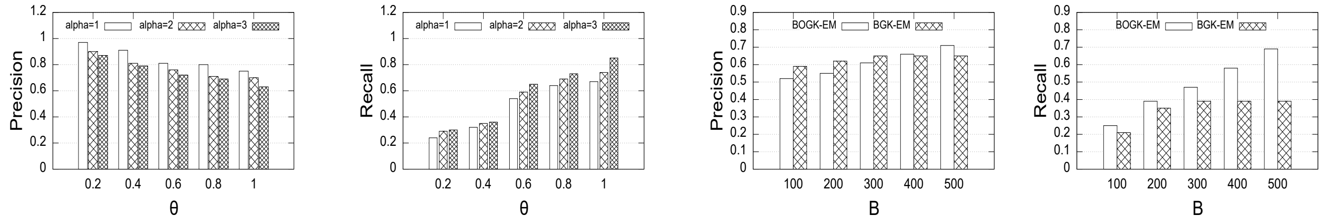
Table 3: Comparative accuracy against baselines

Entity Type	OGK-EM	Vickey	Holistic
(#entities, #positives)	P/R/F	P/R/F	P/R/F
Book(126K, 13.2K)	**0.97/0.85/0.9**	0.96/0.36/0.5	0.8/0.58/0.67
Actor(204K, 5K)	**1/0.66/0.79**	0.95/0.12/0.2	0.6/0.36/0.46
Museum(34K, 3.2K)	0.99/**0.42/0.58**	**1**/0.1/0.18	0.82/0.14/0.2
Scientist(154K, 16.1K)	**0.99/0.67/0.8**	**0.99**/0.2/0.33	0.72/0.56/0.6
University(215K, 12.6K)	**0.96/0.5/0.66**	0.93/0.11/0.2	0.79/0.3/0.43
Movie(33K, 9.5K)	**0.95/0.74/0.8**	0.95/0.1/0.18	0.65/0.1/0.17

Varying budget B. Figure 10c and Figure 10d show that algorithms BOGK-EM and BGK-EM achieve improved precision and recall for larger B (we set (α, θ) to (2, 0.8)). As expected, an increased budget allows for more OGKs to be evaluated, and more merges of entity equivalent nodes to occur, thereby capturing more true positives. We observe a similar trend for increased fpr values (not shown) for increasing B, α, θ values. We note that BOGK-EM is more accurate than BGK-EM in capturing true positives as it achieves on average 20% gain in recall. We also observe a 47% gain in F_1 score by BOGK-EM when we vary B from 100 to 500, and a lesser gain of 15% by BGK-EM (not shown).

Comparing the above analysis with their efficiency counterparts in Exp-2 (Figures 9c and 9g), we verify that OGK-based techniques support a flexible trade-off between matching efficiency and accuracy by tuning (α, β) and budget B.

Exp-4. Case Analysis. We study the accuracy of OGK-EM against the baselines, and report a case analysis over 6 common entity types in Table 3, in terms of precision (P), re-



(a) prec vs.(α, θ)(DBpYago-Dup) **(b)** rec vs.(α, θ)(DBpYago-Dup) **(c)** prec vs. B (DBpYago-Dup) **(d)** rec vs. B (DBpYago-Dup)

Figure 10: Effectiveness of Entity Matching using OGKs.

call (R) and F-measure (F). All entities are from DBpedia-Yago except for *Movie* entities, which are from DBpedia-IMDb. We set α=2, θ=0.8, and B=500. These settings over real benchmark graphs are guided by cases that have high accuracy using DBpYago-Dup. When no ground truth is available, a configuration can be initialized by duplicating entities with equal labels and examining cases with reasonable accuracy that cover these ground truth subsets.

We observe the following. (1) OGK-EM achieves the best precision and recall in most cases using OGKs: the joint topological and value constraints improve precision, while the ontological matching mitigates loss of recall. (2) Vickey achieves comparable precision at a cost of low recall due to enforcing label equality. In addition, conditional keys mined by Vickey may be overfitted to the datasets leading to lower recall (*e.g.,* "Actor" in Table 3). For example, Vickey fails to identify *Michael Burrows* (scientist) and *Michael Burrows* (person). (3) Holistic achieves higher recall but lower precision compared with Vickey. Indeed, not all "similar" entities are equivalent. For example, two scientists *William Arthur* (botanist) and *William Arthur* (mathematician) are linked by Holistic due to the same name, work place and nationality, but are correctly distinguished by OGKs due to the high ontological matching cost. (4) GK-EM incurs an average 21% decrease in recall compared to OGK-EM by enforcing only type equality without considering entity ids. EnumEM provides similar accuracy results with OGK-EM. Due to limited space, we report full details in [1].

8. RELATED WORK

Ontological Dependencies. Existing work has coupled ontologies with functional dependencies (FDs), and equality-generating dependencies (EGDs) over RDF triples [3,23,30]. These constraints, however, are limited to value bindings with no topological restrictions. Ontological FDs (OFDs) [8] is a tighter integration of FDs and ontologies in relational data, by relaxing the strict equality conditions in FDs to include synonym and inheritance (is-a) notions of semantic equivalence. OFDs have shown to significantly reduce false positive errors, improve recall in entity resolution.

This work extends keys for graph to include both graph patterns and ontologies to capture topological constraints and semantic equivalence. OFDs restrict ontological extensions to only the consequent attribute of the dependency. Furthermore, OFDs cannot be directly applied to characterize OGKs with topology and value constraints. Our work also benefits from pattern matching with ontologies [36].

Graph Dependencies. Recent work has proposed variants of graph functional dependencies (GFDs) that define value relationships over entities satisfying topology constraints [17, 18]. Keys for graphs specialize GFDs by enforcing *node identity* to identify an entity [16]. In addition, keys for graphs may be *recursively defined*, allowing entity identification to be dependent on sub-entities. This recursive dependency makes keys for graphs inherently more complex than GFDs. In OGKs, this recursive complexity manifests in entity matching to efficiently resolve all dependent sub-entities, and to accurately propagate semantic similarity across the matched nodes. Graph entity dependencies (GEDs) unifies GFDs and keys [17]. These bindings are fixed and inflexible to exploit external ontologies to reconcile semantically equivalent entities. Closer to our work is conditional keys for RDFs [35]. These constraints are defined by a conjunctive condition over attribute properties, and enforce attribute identity. Nevertheless, conditional keys are not characterized by patterns and topological constraints.

Graph Matching and Entity Categorization. Probabilistic graph matching seeks a mapping that induces similar subgraphs from a pair of graph instances, and their node features [38]. Entity matching differs from graph matching as two similar nodes in graph matching may not necessarily refer to the same entity. By using graph patterns and ontologies, we can precisely define the context of entity equivalence and also enable feasible entity matching for big graphs. Unlike entity categorization [20], OGKs incorporate ontological similarity during the matching process, and dynamically propagate this similarity to neighboring nodes to identify entity pairs that are semantically equivalent.

9. CONCLUSION

We proposed a class of ontological graph keys (OGKs), which are a variant of graph keys by relaxing node identity to entity identifier equivalence, and type equality to ontological matching. We show that adding ontological matching does not make reasoning of OGKs harder: validation and implication problems are both NP-complete. We extended Chase to characterize entity matching with OGKs, and show Chase preserves the Church Rosser property with fixed scope. We developed efficient algorithms with early termination, for both exact matching and budgeted matching. OGKs are a first class of approximate graph key constraints with tunable tolerance on type mismatching. As next steps, we intend to study the application of OGKs to knowledge fusion (e.g., compared with graph embeddings), and the parallel discovery of OGKs in distributed graphs.

Acknowledgements. This work is supported in part by USDA/NIFA 2018-67007-28797, Siemens, and the Department of Energy under DE-IA0000025 for UI-ASSIST project.

10. REFERENCES

[1] Full version. *http://www.cas.mcmaster.ca/∼alipoum/full.pdf.*

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] W. Akhtar, A. Cortés-Calabuig, and J. Paredaens. Constraints in RDF. In *SDKB*, pages 23–39, 2011.

[4] M. Alipourlangouri and F. Chiang. Keyminer: Discovering keys for graphs. In *VLDB workshop TD-LSG*, 2018.

[5] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[6] M. Arenas and L. Libkin. A normal form for xml documents. *ACM TODS*, 29(1):195–232, 2004.

[7] M. Atencia, M. Chein, M. Croitoru, J. David, M. Leclère, N. Pernelle, F. Saïs, F. Scharffe, and D. Symeonidou. Defining key semantics for the rdf datasets: experiments and evaluations. In *ICCS*, pages 65–78, 2014.

[8] S. Baskaran, A. Keller, F. Chiang, L. Golab, and J. Szlichta. Efficient discovery of ontology functional dependencies. In *CIKM*, pages 1847–1856, 2017.

[9] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. The traveling salesman problem in bounded degree graphs. *ACM Trans. Algorithms*, 8(2):1–18, 2012.

[10] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for xml. *Computer networks*, 39(5):473–487, 2002.

[11] A. Calì, G. Gottlob, and T. Lukasiewicz. Datalog±: a unified approach to ontologies and integrity constraints. In *ICDT*, pages 14–30, 2009.

[12] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop, GbRRR*, pages 149–159, 2001.

[13] A. Cortés-Calabuig and J. Paredaens. Semantics of constraints in RDFS. In *AMW*, pages 75–90, 2012.

[14] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *VLDB*, 7(10):881–892, 2014.

[15] V. Efthymiou, K. Stefanidis, and V. Christophides. Benchmarking blocking algorithms for web entities. *IEEE Transactions on Big Data*, pages 1–14, 2016.

[16] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs. *PVLDB*, 8(12):1590–1601, 2015.

[17] W. Fan and P. Lu. Dependencies for graphs. In *PODS*, pages 403–416, 2017.

[18] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD*, pages 1843–1857, 2016.

[19] M. Gardner and T. M. Mitchell. Efficient and expressive knowledge base completion using subgraph feature extraction. In *EMNLP 2015*, pages 1488–1498, 2015.

[20] S. Hao, N. Tang, G. Li, and J. Feng. Discovering mis-categorized entities. In *ICDE*, pages 413–424, 2018.

[21] S. Hao, N. Tang, G. Li, and J. Li. Cleaning relations using knowledge bases. In *ICDE*, pages 933–944. IEEE, 2017.

[22] J. Hellings, M. Gyssens, J. Paredaens, and Y. Wu. Implication and axiomatization of functional constraints on patterns with an application to the RDF data model. In *FoIKS*, pages 250–269, 2014.

[23] J. Hellings, M. Gyssens, J. Paredaens, and Y. Wu. Implication and axiomatization of functional and constant constraints. *Annals of Mathematics and Artificial Intelligence*, 76(3-4):251–279, 2016.

[24] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[25] R. Knappe, H. Bulskov, and T. Andreasen. Perspectives on ontology-based querying. *International Journal of Intelligent Systems*, 22(7):739–761, 2007.

[26] N. Lao, T. M. Mitchell, and W. W. Cohen. Random walk inference and learning in A large scale knowledge base. In *EMNLP*, pages 529–539, 2011.

[27] G. Levchuk, J. Roberts, and J. Freeman. Learning and detecting patterns in multi-attributed network data. In *AAAI Fall Symposium Series*, 2012.

[28] P. Lin, Q. Song, J. Shen, and Y. Wu. Discovering graph patterns for fact checking in knowledge graphs. In *DASFAA*, pages 783–801, 2018.

[29] D. Menestrina, S. E. Whang, and H. Garcia-Molina. Evaluating entity resolution results. *PVLDB*, 3(1-2):208–219, 2010.

[30] B. Motik, I. Horrocks, and U. Sattler. Adding Integrity Constraints to OWL. In *OWLED*, 2007.

[31] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, pages 11–33, 2016.

[32] N. Pernelle, F. Saïs, and D. Symeonidou. An automatic key discovery approach for data linking. *Web Semantics: Science, Services and Agents on the World Wide Web*, 23:16–30, 2013.

[33] M. Pershina, M. Yakout, and K. Chakrabarti. Holistic entity matching across knowledge graphs. In *IEEE Big Data*, pages 1585–1590, 2015.

[34] M. R. Quillan. Semantic memory. Technical report, Bolt Beranek and Newman Inc. Cambridge MA, 1966.

[35] D. Symeonidou, L. Galárraga, N. Pernelle, F. Saïs, and F. Suchanek. Vickey: Mining conditional keys on knowledge bases. In *ISWC*, pages 661–677, 2017.

[36] Y. Wu, S. Yang, and X. Yan. Ontology-based subgraph querying. In *ICDE*, pages 697–708, 2013.

[37] Y. Yu and J. Heflin. Extending functional dependency to detect abnormal data in RDF graphs. In *ISWC*, pages 794–809, 2011.

[38] R. Zass and A. Shashua. Probabilistic graph and hypergraph matching. In *CVPR*, pages 1–8, 2008.