Keys for Graphs

Wenfei Fan[1,2] Zhe Fan[3] Chao Tian[1,2] Xin Luna Dong[4]
[1]University of Edinburgh [2]Beihang University [3]Hong Kong Baptist University [4]Google Inc.
{wenfei@inf., chao.tian@}ed.ac.uk, zfan@comp.hkbu.edu.hk, lunadong@google.com

ABSTRACT

Keys for graphs aim to uniquely identify entities represented by vertices in a graph. We propose a class of keys that are recursively defined in terms of graph patterns, and are interpreted with subgraph isomorphism. Extending conventional keys for relations and XML, these keys find applications in object identification, knowledge fusion and social network reconciliation. As an application, we study the entity matching problem that, given a graph G and a set Σ of keys, is to find all pairs of entities (vertices) in G that are identified by keys in Σ. We show that the problem is intractable, and cannot be parallelized in logarithmic rounds. Nonetheless, we provide two parallel scalable algorithms for entity matching, in MapReduce and a vertex-centric asynchronous model. Using real-life and synthetic data, we experimentally verify the effectiveness and scalability of the algorithms.

1. INTRODUCTION

Keys provide an invariant connection between a real-world entity and its representation in a database. They are fundamental to relational databases: data models, conceptual design, and prevention of update anomalies. They are found in almost every database textbook. Keys have also been extensively studied for XML and are part of XML Schema.

For all the reasons that keys are essential to relations and XML, keys are also needed for graphs. The need is evident when relations are represented as graphs [6, 8, 32, 36], and for citations of "digital objects" of graph structures [11]. They are also important to emerging applications such as knowledge fusion and knowledge base expansion [15, 16, 34], to deduplicate entities and to fuse information from different sources that refers to the same entity. Another application is social network reconciliation, to reconcile user accounts across multiple social networks [28]. However, keys for graphs are more challenging than conventional keys.

Example 1: We illustrate keys for graphs by using examples taken from various domains in knowledge bases.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

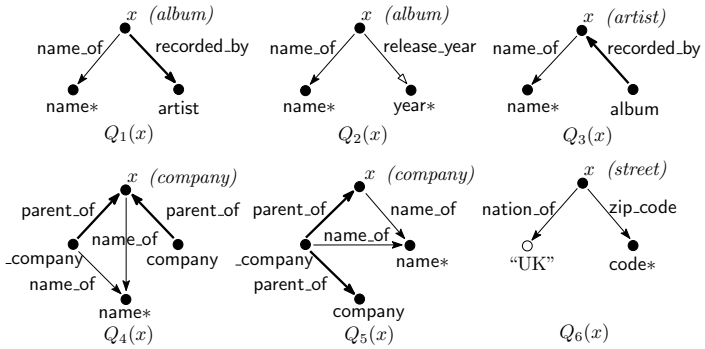


Figure 1: Keys for graphs as graph patterns

Music. Consider a knowledge base G_1 consisting of triples (s, p, o), indicating subject, predicate and object, respectively; *e.g.,* (album, recorded_by, artist) says that an album is recorded by an artist. It is modeled as a graph in which s and o are nodes, connected by an edge from s to o labeled p.

One might think that name is a key for album. However, this is not the case. For instance, there are different albums recorded by the Beatles and John Farnham with the same name "Anthology 2" in Freebase. Indeed, the name of an album x uniquely identifies x only among all albums recorded by *the same* artist[1]. Alternatively, an album can be identified by its name and its year of initial release. These yield two keys for albums: *An album can be uniquely identified by*

Q_1: *its name and its primary recording artist,* or
Q_2: *its name and its year of initial release.*

For the same reason, an artist may not be identified by its name. Indeed, there are 6 artists or bands named "Everest". Nonetheless, a key for artist can be given by incorporating one of the albums that the artist recorded.

Q_3: *An artist can be identified by the name, and the album he or she recorded.*

These keys are depicted as *graph patterns* $Q_1(x)$, $Q_2(x)$ and $Q_3(x)$ in Fig. 1, respectively, where x denotes an entity of a particular type to be identified. Intuitively, $Q_1(x)$ says that if two album entities x_1 and x_2 have the same name and are recorded_by the same artist, then x_1 and x_2 must be the same album; similarly for $Q_2(x)$ and $Q_3(x)$.

In contrast to keys for relations and XML, keys for graphs specify *topological constraints* with a graph pattern. Such keys (a) may consider not only *value equality* based on value bindings of properties, *e.g.,* name in Q_1, but also *node*

[1]We require exact match in the examples for simplicity; nonetheless, we can easily relax the constraints to similarity match.

identity, *e.g.,* the identity of artist node in Q_1; and (b) can be *recursively defined*, *e.g.,* to identify an album entity x, we may need to identify its primary artist y, while to identify an artist entity y, we need to identify one of its albums x.

Business. As another example, consider the domain for businesses. Typically we can identify a company in the US by its name and head-quarter location. However, there is often business merging and splitting, and very commonly the child company may carry the same name of the parent company without moving the head-quarter (*e.g.,*$AT\&T$ and SBC merged in 2005 and the new company carried the name of $AT\&T$). To distinguish the parent company and the child company in this case, we need to encode the parent-child relationship in the key. That leads to the following two keys to identify companies in a knowledge base G_2, the former for the case of merging and the latter for splitting.

Q_4: *A company merged from a parent company of the same name can be identified by the company name and the other parent company.*

Q_5: *A company split from a parent company of the same name can be identified by the company name and another child company after splitting.*

These keys demonstrate another departure from traditional keys: (a) Q_4 and Q_5 are directed acyclic graphs (DAG), as shown in Fig. 1; and (b) they include properties of different entities, *e.g.,* $Q_4(x)$ for company incorporates both the name of the company and the name of its parent company.

Address. To identify a street in the UK, one can use:

Q_6: *A street in the UK can be identified by its zip code.*

This key does not hold for streets in, *e.g.,* the US. As shown in Fig. 1, Q_6 is specified with *a constant* as a condition. This is another departure from conventional keys. □

Keys for graphs are a departure from conventional keys. To make practical use of such keys, several questions have to be answered. How should we define keys for graphs, from syntax to semantics? What is the complexity of identifying entities with keys? Is there any scalable algorithm to identify entities with keys in big graphs?

Contributions. This paper studies keys for graphs, from specifications and semantics to applications.

(1) We propose a class of keys for graphs (Section 2). We define keys in terms of *graph patterns*, to specify topological constraints and value bindings needed for identifying entities. Moreover, keys may be *recursively defined*: to identify a pair of entities, we may need to decide whether some other entities can be identified, as shown by Q_1, Q_3–Q_5 of Example 1. We interpret keys by means of graph pattern matching via subgraph isomorphism. These make such keys more expressive than our familiar keys for relations and XML.

(2) We study *entity matching*, an application of keys for graphs (Section 3). Given a graph G and a set Σ of keys for graphs, *entity matching* is to find all pairs of entities (vertices) in G that can be identified by keys in Σ. We formalize the problem by revising the chase [3] studied in the classical dependency theory. While entity matching is in PTIME (polynomial time) for relations and XML with traditional keys, we show that its decision problem is NP-complete for graphs. Worse still, recursively defined keys pose new challenges. We show that entity matching does not have the polynomial-fringe property (PFP) [4], and cannot be solved in logarithmic parallel computation rounds.

Nonetheless, we show that entity matching is within reach in practice, by providing parallel scalable algorithms.

(3) We develop a MapReduce algorithm for entity matching (Section 4). As opposed to subgraph isomorphism, entity matching with recursively defined keys requires a fixpoint computation, and in each round, multiple isomorphism checking for each entity pair. We show that the algorithm is *parallel scalable*, *i.e.,* its worst-case time complexity is $O(t(|G|, |\Sigma|)/p)$, where $t(,)$ is a function in $|G|$ and $|\Sigma|$, and p is the number of processors used. It guarantees to take proportionally less time with the increase of p, which is *not* warranted by many parallel algorithms. We also develop optimization methods to process recursively defined keys.

(4) We give another algorithm in the vertex-centric asynchronous model of [31] (Section 5). This algorithm not only checks different entity pairs in parallel, but also inspects different mappings in parallel when checking each entity pair, via asynchronous message passing. It reduces unnecessary costs inherent to the I/O bound and the synchronization policy ("blocking" of stragglers) of MapReduce. We show that the algorithm is also parallel scalable. Moreover, we propose optimization techniques to reduce message passing.

(5) Using real-life and synthetic data, we experimentally evaluate our algorithms (Section 6). Despite the intractability and the hardness of parallelization, we find that our MapReduce and vertex-centric algorithms are indeed parallel scalable: they are *4.8 and 4.9 times faster* on average, respectively, when the number of processors increases from 4 to 20. They are reasonably efficient: they take *27 and 1.5 seconds* on average with 20 processors, respectively, on graphs with 600 million nodes and edges, for 500 recursively defined keys. Moreover, our optimization techniques for the two are effective, and improve the performance by 200% and 50%, respectively. We also find that the vertex-centric one reduces inherent costs of MapReduce, and performs better.

We contend that these keys provide an analogy of traditional keys for graph-structured data. Like relational and XML keys, they specify *the semantics of the data* and remain invariant regardless of changes to the data. They are important to not only traditional use of keys but also several emerging applications. Moreover, entity matching permits parallel scalable algorithms and is feasible in big graphs.

We focus on definition and application of keys in this paper, and defer the study of key discovery by, *e.g.,* path-identification [29] or communication theory [23], to another publication. All proofs of the results of the paper are in [2].

Related work. We characterize related work as follows.

Keys. Relational keys are defined over a relation schema in terms of a set of attributes [3]. XML keys are specified in terms of path expressions in the absence of schema [10].

In contrast to traditional keys, keys for graphs (a) are defined in terms of *graph patterns*, specifying constraints on both topological structure and value bindings, in the absence of schema; (b) they are interpreted based on *graph pattern matching*, with both value equality and node identity; and (c) they can be *recursively defined*. These keys are useful in emerging applications besides their traditional use.

To the best of our knowledge, the only prior work on keys

for graphs is [33], which specifies keys for RDF data in terms of a combination of object properties and data properties defined over OWL ontology. Such keys differ from keys of this work in that they (a) cannot be recursively defined, (b) do not enforce topological constraints imposed by graph patterns, and (c) adopt the unique name assumption via URIs, which is often too strong for entity matching.

Entity resolution. Entity resolution (*a.k.a.* entity matching, record linkage, etc.) is to identify records that refer to the same real-world entity. There has been a host of work on the topic, following iterative clustering [7,32], learning-based [27, 36], rule-based methods [6, 17] (see [12, 20] for surveys).

Keys for graphs yield a *declarative and deterministic* method to provide an invariant connection between vertices and the real-world entities they represent, and fall in the rule-based approach. Prior rule-based methods mostly focus on *relational data*; this work is to define a primary form of constraints for *graphs*, namely, keys. The quality of matches identified by keys highly depends on keys discovered and used, although keys help us reduce false positives. We defer the topic of key discovery to another paper, and focus primarily on the efficiency of applying such constraints.

One branch of entity resolution, called collective entity resolution [8, 14, 36], is to jointly determine entities for co-occurring references and propagate similarities of entities. Analogous to datalog rules [6], keys for graphs extend this approach by providing recursively defined rules, based on *graph pattern matching*. This work addresses some of the emerging challenges highlighted in [20], by targeting graphs when data is "more linked", and by providing parallel scalable algorithms for "larger datasets".

Finally, we remark that entity resolution is just *one of* the applications for keys for graphs, besides, *e.g.,* digital citations [11] and knowledge base expansion [15].

Parallel algorithms. Parallel algorithms have been developed for subgraph isomorphism [22, 26, 35, 38], and for entity resolution [7,25,27,32,36]. As remarked earlier, [7,27,32,36] target record matching in relations; [25] deals with graphs but adopts relational record matching methods.

Our algorithms differ from previous ones in the following. (a) Entity matching is far more intriguing than conventional subgraph isomorphism, and the prior algorithms [22, 26, 35, 38] cannot be applied to entity matching. (b) For the same reasons, entity matching is more involved than record matching of [7,27,32,36] to identify tuples in relations, and than the task of [25] that does not enforce topological constraints in the matching process. (c) We propose optimization strategies that have not been studied before.

Related to this work are also parallel algorithms for evaluating datalog [4, 37]. However, entity matching with keys requires to identify *bijective functions* for subgraph isomorphism, which are more challenging to compute. Worse still, we show that entity linking does not have PFP [4], and is harder to be parallelized than, *e.g.,* transitive closures.

2. SPECIFYING KEYS WITH GRAPH PATTERNS

In this section, we formally define keys for graphs.

2.1 Graphs and Graph Pattern Matching

We start with graphs, patterns and pattern matching.

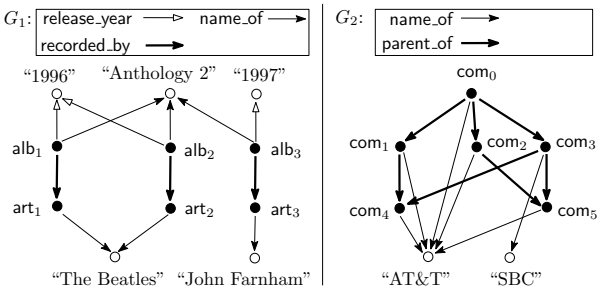


Figure 2: Fragments of two knowledge graphs

Graphs. Assume a set $\mathcal{E}$ of entities, a set $\mathcal{D}$ of values, a set $\mathcal{P}$ of predicates (labels), and a set Θ of types. Each entity e in $\mathcal{E}$ has a *unique ID* and a *type* in Θ.

A *graph* G is a set of triples $t = (s, p, o)$, where *subject* s is an entity in $\mathcal{E}$, p is a *predicate* in $\mathcal{P}$, and *object* o is either an entity in $\mathcal{E}$ or a value d in $\mathcal{D}$. It can be represented as a directed edge-labeled graph (V, E), also denoted by G, such that (a) V is the set of nodes consisting of s and o for each triple $t = (s, p, o)$; and (b) there is an edge in E from s to o labeled p for each triple $t = (s, p, o)$.

We consider two types of equality:

(a) *node identity* on $\mathcal{E}$: $e_1 \Leftrightarrow e_2$, if entities e_1 and e_2 have the same ID, *i.e.,* they refer to the same entity; and

(b) *value equality* on $\mathcal{D}$: $d_1 = d_2$ if they are the same value.

In G, e_1 and e_2 are represented as the same node if $e_1 \Leftrightarrow e_2$; similarly for values d_1 and d_2 if $d_1 = d_2$.

Example 2: Two graphs G_1 and G_2 are shown in Fig. 2. (1) Graph G_1 represents a fragment of a knowledge base consisting of artists and their albums. For instance, in triple (art_1, name_of, "The Beatles"), subject art_1 is an entity of type *artist*, and object "The Beatles" is a value; in G_1, both are represented as nodes, and the triple is presented as an edge labeled name_of from art_1 to "The Beatles".

(2) Graph G_2 depicts a set of triples for companies. It tells us that, *e.g.,* "AT&T" (com_4 of type *company*) has parent companies "AT&T" (com_1) and "SBC" (com_3). □

Graph patterns. A *graph pattern* $Q(x)$ is a set of triples (s_Q, p_Q, o_Q), where s_Q is a *variable* z, o_Q is either a value d or a variable z, and p_Q is a predicate in $\mathcal{P}$. Here z has one of three forms: (a) *entity variable* y, to map to an entity, (b) *value variable* $y*$, to map to a value, and (c) *wildcard* _y, to map to an entity. Here s_Q can be either y or _y, while o_Q can be y, $y*$ or _y. Entity variables and wildcard carry a *type*, denoting the type of entities they represent. In particular, x is a designated variable in $Q(x)$, denoting an entity.

As will be seen shortly when we define keys, we enforce node identity ($\Leftrightarrow$) on variables y, and value equality ($=$) on $y*$; for a wildcard _y, we just require the existence of an entity with the type of _y without checking its node ID or value. Value d in $Q(x)$ indicates a *value binding* condition.

A graph pattern can also be represented as a graph such that two variables are represented as the same node if they have the same name of y, $y*$ or _y; similarly for values d. We assume *w.l.o.g.* that $Q(x)$ is *connected*, *i.e.,* there exists an *undirected* path between x and each node in $Q(x)$.

Example 3: Six graph patterns are depicted in Fig. 1. For instance, $Q_4(x)$ represents triples (x, name_of, name*), (_company, name_of, name*), (_company, parent_of, x) and (company, parent_of, x). Here x is the designated variable

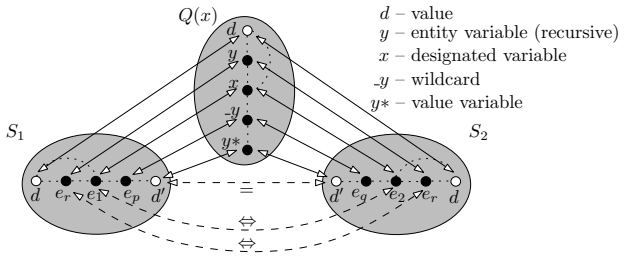


Figure 3: The semantics of keys for graphs

(type *company*), name∗ is a value variable, company is an entity variable and _company is a wildcard for *company*. In Q_6, "UK" is a constant value (*i.e.,* d) as a condition. □

A *valuation* of $Q(x)$ *in a set* S *of triples* is a mapping ν from $Q(x)$ to S that preserves values in $\mathcal{D}$ and predicates in $\mathcal{P}$, and maps variables y and $\_y$ to entities of *the same type*. More specifically, for each triple (s_Q, p_Q, o_Q) in $Q(x)$, there exists (s, p, o) in S, written as $(s_Q, p_Q, o_Q) \mapsto_\nu (s, p, o)$ or simply $(s_Q, p_Q, o_Q) \mapsto (s, p, o)$, where

(a) $\nu(s_Q) = s$, $p = p_Q$, $\nu(o_Q) = o$;
(b) o is an entity if o_Q is a variable y or $\_y$; it is a value if o_Q is $y*$, and $o = d$ if o_Q is a value d; and
(c) entities s and s_Q have the same type; similarly for entities o and o_Q if o_Q is y or $\_y$.

We say that ν is a *bijection* if ν is one-to-one and onto.

Graph pattern matching. Consider a graph G and an entity e in G. We say that G *matches* $Q(x)$ *at* e if there exist a set S of triples in G and a valuation ν of $Q(x)$ in S such that $\nu(x) = e$, and ν is a bijection between $Q(x)$ and S. We refer to S as a *match* of $Q(x)$ in G at e under ν.

Intuitively, ν is an isomorphism from $Q(x)$ to S when $Q(x)$ and S are depicted as graphs. That is, we adopt *subgraph isomorphism* for the semantics of graph pattern matching.

Example 4: Consider $Q_4(x)$ of Fig. 1, G_2 of Fig. 2, and a set S_1 of triples in G_2: {(com1, name_of, "AT&T"), (com4, name_of, "AT&T"), (com1, parent_of, com4), (com3, parent_of, com4)}. Then S_1 is a match of $Q_4(x)$ in G_2 at com4, which maps variable x to com4, name∗ to "AT&T", wildcard _company to com1, and company to com3. □

2.2 Keys for Graphs

Based on graph patterns, we next define keys for graphs.

Keys. A *key for entities of type* τ is a graph pattern $Q(x)$, where x is a designated entity variable of type τ.

Intuitively, it says that in a graph G, for entities e of type τ, the conditions specified in $Q(x)$ uniquely identify e. For example, Q_1 and Q_2 of Example 1 are keys for *album*, Q_3 is a key for *artist*, and Q_4 and Q_5 are keys for *company*.

To give the semantics of keys, we use the following notion. Consider matches S_1 and S_2 of $Q(x)$ at e_1 and e_2 in G under ν_1 and ν_2, respectively. We say that S_1 *coincides with* S_2, denoted by $S_1(e_1) \cong_Q S_2(e_2)$, if a bijection μ between S_1 and S_2 can be derived from ν_1 and ν_2, preserving node identity and value equality. That is, for each (s_Q, p_Q, o_Q) in $Q(x)$ such that $(s_Q, p_Q, o_Q) \mapsto_{\nu_1} (s_1, p_1, o_1)$ and $(s_Q, p_Q, o_Q) \mapsto_{\nu_2} (s_2, p_2, o_2)$, it satisfies conditions:

(a) if s_Q is a variable y that is distinct from x, then $s_1 \Leftrightarrow s_2$; similarly for o_Q; and
(b) if o_Q is a variable $y*$, then $o_1 = o_2$.

Symbols	Notations
$\mathcal{E}$, $\mathcal{P}$, $\mathcal{D}$	entities, predicates and data values, respectively
G, $Q(x)$	graph and graph pattern, respectively
$e_1 \Leftrightarrow e_2$	node identity
$d_1 = d_2$	value equality
y, $y*$, $\_y$	variables for entities, values and wildcards, resp.
$\mapsto_\nu$, $\mapsto$	mapping from (s_Q, p_Q, o_Q) to (s, p, o)
$S_1(e_1) \cong_Q S_2(e_2)$	match S_1 at e_1 coincides with S_2 at e_2
$G \models Q(x)$	G satisfies key $Q(x)$
$(G, \Sigma) \models (e_1, e_2)$	entities e_1 and e_2 are *identified* by keys in Σ
$\|G\|$, $\|Q(x)\|$	the size of graph G and $Q(x)$
$d(Q, x)$	the radius of $Q(x)$

Table 1: Notations

When s_Q is a wildcard $\_y$, we *do not* require that $s_1 \Leftrightarrow s_2$, *i.e.,* s_1 and s_2 may be distinct entities; similarly for o_Q.

We say that G *satisfies* key $Q(x)$, denoted by $G \models Q(x)$, if *for all entities* e_1 and e_2 in G, if *there exist* matches S_1 and S_2 of $Q(x)$ such that $S_1(e_1) \cong_Q S_2(e_2)$, then $e_1 \Leftrightarrow e_2$.

As shown in Fig. 3, the key says that if there exist S_1 and S_2 verifying that e_1 and e_2 satisfy the conditions of $Q(x)$, respectively, and if $S_1(e_1) \cong_Q S_2(e_2)$, then e_1 and e_2 must have the same ID, *i.e.,* they are the same entity.

Example 5: Continuing with Example 4, one can see that $G_2 \not\models Q_4(x)$. Consider S_1 of Example 4, and a match S_2 of $Q_4(x)$ at com5: {(com2, name_of, "AT&T"), (com5, name_of, "AT&T"), (com2, parent_of, com5), (com3, parent_of, com5)}. Then $S_1(\mathsf{com}_4) \cong_{Q_4} S_2(\mathsf{com}_5)$ but com4 and com5 are distinct entities in G_2. Thus either com4 or com5 is a duplicate.

Similarly in G_1, either alb1 or alb2 is a duplicate (violation of Q_2), and either art1 or art2 is a duplicate (by Q_3). However, these are not very obvious since keys for *album* and *artist* are defined by mutual recursion. □

We say that a key $Q(x)$ is *recursively-defined* if it contains some variables y other than x, and is *value-based* otherwise. Intuitively, when $Q(x)$ is recursive, $e_1 \Leftrightarrow e_2$ depends on whether $e \Leftrightarrow e'$ for some other entities e and e' can be identified by variable y, which involves node identity that is determined by using (possibly other) keys. In contrast, when $Q(x)$ is value-based, it decides whether $e_1 \Leftrightarrow e_2$ simply based on value equality on relevant triples in S_1 and S_2.

Example 6: Keys Q_1, Q_3, Q_4 and Q_5 depicted in Fig. 1 are all recursive, while Q_2 and Q_6 are value-based. □

Remark. (1) For simplicity, we focus on keys defined in terms of value equality and node identity. Nonetheless, the results of this paper remain intact when *similarity predicates* are used along the same lines as value equality. (2) Relational keys [3] and XML keys [10] can be readily expressed as value-based keys with patterns of a form of trees.

We will also use the following notations: (1) $|G|$ (resp. $|Q|$) denotes the number of triples in G (resp. $Q(x)$); (2) for a set Σ of keys, $|\Sigma| = \sum_{Q(x)\in\Sigma} |Q|$ and $\|\Sigma\|$ is its cardinality; and (3) the *radius* of $Q(x)$, denoted by $d(Q, x)$, is the longest distance between x and any node in $Q(x)$ when $Q(x)$ is treated as an *undirected* graph, ignoring the edge direction.

The notations of this paper are summarized in Table 1.

3. THE ENTITY MATCHING PROBLEM

In the rest of the paper we focus on *entity matching*, an important application of keys. We formalize the problem (Section 3.1) and establish its complexity (Section 3.2). Moreover, we show that in the presence of recursively defined keys, entity matching is hard to be parallelized (Section 3.3).

3.1 Entity Matching with Keys

Example 5 shows that $G_2 \not\models Q_4(x)$, since $S_1(\mathsf{com}_4) \cong_{Q_4} S_2(\mathsf{com}_5)$ but com_4 and com_5 are distinct. However, key $Q_4(x)$ tells us that com_4 and com_5 refer to the same entity and should be *identified*. To formalize this, we revise the classical chase [3] by using keys as rules for entities in graphs.

Chase revisited. Consider a set Σ of keys and a graph G. We use Eq to denote *the equivalence relation* of a set of pairs (e, e') of entities in G of the same type that have been identified by keys in Σ, *i.e.,* Eq is reflexive, symmetric and transitive. We denote by Eq_0 the node identity relation $\Leftrightarrow$, *i.e.,* the set of pairs (e, e) for all entities e in G.

Consider a key $Q(x) \in \Sigma$ and matches S_1 and S_2 of $Q(x)$ at e_1 and e_2 in G under valuations ν_1 and ν_2, respectively. We define $S_1(e_1) \cong_Q^{\mathsf{Eq}} S_2(e_2)$ by using Eq instead of relation $\Leftrightarrow$ in the definition of $S_1(e_1) \cong_Q S_2(e_2)$. More specifically, for each triple (s_Q, p_Q, o_Q) in Q, if $(s_Q, p_Q, o_Q) \mapsto_{\nu_1} (s_1, p_1, o_1)$ and $(s_Q, p_Q, o_Q) \mapsto_{\nu_2} (s_2, p_2, o_2)$, then

(a) if s_Q is a variable y distinct from x, then $(s_1, s_2) \in \mathsf{Eq}$ (instead of $s_1 \Leftrightarrow s_2$); similarly for o_Q; and

(b) if o_Q is a variable $y*$, then $o_1 = o_2$.

We define *a chase step* of G by Σ at Eq as

$$\mathsf{Eq} \Rightarrow_{(e_1,e_2)} \mathsf{Eq}',$$

where (e_1, e_2) is a pair of entities in G such that (a) $(e_1, e_2) \notin \mathsf{Eq}$, (b) there exist a key $Q(x)$ in Σ and matches S_1 and S_2 of $Q(x)$ at e_1 and e_2, respectively, such that $S_1(e_1) \cong_Q^{\mathsf{Eq}} S_2(e_2)$; and (c) Eq' is the equivalence relation of $\mathsf{Eq} \cup \{(e_1, e_2)\}$.

Intuitively, when e_1 and e_2 are identified by using a key in Σ, Eq is expanded to Eq' by including (e_1, e_2). For instance, in G_1, $\mathsf{Eq}_0 \Rightarrow_{(\mathsf{alb}_1, \mathsf{alb}_2)} \mathsf{Eq}_1$, where Eq_1 is the extension of node identity relation $\Leftrightarrow$ in G_1 by including $(\mathsf{alb}_1, \mathsf{alb}_2)$.

A *chasing sequence* of G by Σ is a sequence

$$\mathsf{Eq}_0, \mathsf{Eq}_1, \ldots, \mathsf{Eq}_k,$$

such that for all $i \in [0, k-1]$, there exists a pair (e_1, e_2) of entities in G, where $\mathsf{Eq}_i \Rightarrow_{(e_1,e_2)} \mathsf{Eq}_{i+1}$. The sequence is *terminal* if no chase step by Σ is defined at Eq_k. We refer to Eq_k as the *result* of the chasing sequence.

Chasing of keys has the *Church-Rosser property*:

Proposition 1: *For any set Σ of keys and graph G, all terminal chasing sequences of G by Σ are finite and have the same result, regardless of how the keys are applied.* □

We denote by $\mathsf{chase}(G, \Sigma)$ the result of a terminal chasing sequence of G by Σ. By Proposition 1, this notion is well-defined. We say that entities e_1 and e_2 in G are *identified* by Σ, denoted by $(G, \Sigma) \models (e_1, e_2)$, if $(e_1, e_2) \in \mathsf{chase}(G, \Sigma)$.

Example 7: Let $\Sigma_1 = \{Q_1(x), Q_2(x), Q_3(x)\}$ from Fig. 1, and $\Sigma_2 = \{Q_4(x), Q_5(x)\}$. Then in G_1 of Fig. 2, $(G_1, \Sigma_1) \models (\mathsf{alb}_1, \mathsf{alb}_2)$ by applying $Q_2(x)$, since alb_1 and alb_2 have the same name "Anthology 2" and were initially released in "1996". This is followed by $(G_1, \Sigma_1) \models (\mathsf{art}_1, \mathsf{art}_2)$ by applying $Q_3(x)$ to entities $\{\mathsf{art}_1, \mathsf{alb}_1\}$ and $\{\mathsf{art}_2, \mathsf{alb}_2\}$. Note that art_1 and art_2 are identified *after* we identify alb_1 and alb_2. This is because in contrast to $Q_2(x)$, $Q_3(x)$ is recursively defined: it has an entity variable album. That is, recursively defined keys impose dependency on entities.

In G_2 of Fig. 2, from the discussion above it follows that $(G_2, \Sigma_2) \models (\mathsf{com}_4, \mathsf{com}_5)$ by $Q_4(x)$. Similarly, $(G_2, \Sigma_2) \models (\mathsf{com}_1, \mathsf{com}_2)$ by applying $Q_5(x)$ to $\{\mathsf{com}_1, \mathsf{com}_0, \mathsf{com}_3\}$ and $\{\mathsf{com}_2, \mathsf{com}_0, \mathsf{com}_3\}$. Note that com_4 and com_5 can be identified before we identify com_1 and com_2, since the *wildcard* _company in $Q_4(x)$ does not require $\mathsf{com}_1 \Leftrightarrow \mathsf{com}_2$. This is why we separate entity variable y from wildcard _y. □

Problem. The *entity matching problem* is stated as follows.

- *Input*: A set Σ of keys, and a graph G.
- *Output*: $\mathsf{chase}(G, \Sigma)$.

3.2 The Complexity of Entity Matching

Given a set of keys on a relation R, it is in PTIME to find all pairs of tuples in R that are identified by the keys. In contrast, the entity matching problem is nontrivial. To see this, consider its decision problem, also referred to as entity matching, which is to determine, given Σ, G and a pair (e_1, e_2) of entities in G, whether $(G, \Sigma) \models (e_1, e_2)$.

Theorem 2: *Entity matching is* NP*-complete.* □

One might think that non-recursive keys would make our lives easier. Unfortunately, this simple case already embeds the subgraph isomorphism problem, which is NP-complete (cf. [19]) and can be reduced to the simple case.

Lemma 3: *The entity matching problem remains* NP*-hard even when Σ consists of a single value-based key $Q(x)$, and when graph G is a DAG (directed acyclic graph).* □

Proof sketch of Theorem 2: The lower bound of Theorem 2 follows from Lemma 3. Its upper bound is much harder to prove. Given a set Σ of (possibly recursively-defined) keys and a (possibly cyclic) graph G, checking $(G, \Sigma) \models (e_1, e_2)$ needs a *fixpoint* computation in which *each step* involves two calls for checking *subgraph isomorphism*.

To show the upper bound, we introduce a notion of proof graphs that are "witnesses" of proving $(G, \Sigma) \models (e_1, e_2)$. We show that $(G, \Sigma) \models (e_1, e_2)$ iff there exists a proof graph that is a DAG with at most N^2 nodes and can be checked in PTIME in $|G|$ and $|\Sigma|$, where N is the number of entities in G. Based on this property, we give an NP algorithm: guess a DAG G_f with at most N^2 nodes, and check whether G_f is a proof graph of $(G, \Sigma) \models (e_1, e_2)$ in PTIME. □

3.3 Recursion and Parallelization

Recursively defined keys introduce challenges beyond subgraph isomorphism. As a result, it is hard to find an efficient parallel algorithm for entity matching. To see this, recall that a datalog program has *the polynomial fringe property* (PFP) if all true facts have a proof tree such that the number of its leaves is polynomial in the data size (cf. [4]). It is known that datalog programs with PFP can be processed in *polylog parallel computation rounds* via recursive doubling, *i.e.,* in $\log^k N$ rounds for a constant k, where N is the size of the input data. We say that a problem has PFP if it has an algorithm with PFP. It is also known that transitive closure (TC), for instance, has PFP. As a result, TC can be computed in logarithmic MapReduce rounds.

Unfortunately, entity matching is harder than TC. Recursively defined keys impose *dependency* on the order of entities to be processed. This leads to chains C of dependent entity pairs such that to identify a pair (e_1, e_2) in C, we have to either wait for pairs preceding (e_1, e_2) in C to be identified, or incur exponentially many possible matches. In contrast, TC can be computed "partially" in PTIME.

Theorem 4: *Entity matching (1) has no PFP, and (2) cannot be parallelized in logarithmic rounds, even on trees.* □

Proof: To prove (1), we show that there exist a tree G_c with $(4N+3)$ entities and a set Σ_c consisting of three keys such that to identify a specific entity pair in G_c, there exists a unique proof tree in which the number of leaves is exponential in N. Hence entity matching does not have PFP.

We prove (2) by reduction from the *Monotone Circuit Value* problem [5]. Given a Boolean circuit as a DAG with INPUT, AND and OR gates, the latter problem is to decide whether the output of the circuit is true. Given a Boolean circuit C, we construct a tree G in logarithmic space in the size of C, and define a set Σ'_c of four keys. We show that the output value of a gate l is true iff $(G, \Sigma'_c) \models (e_l, e'_l)$, where (e_l, e'_l) is a pair of entities uniquely identified by l. Since the monotone circuit value problem cannot be solved in logarithmic rounds [5], neither can entity matching. □

When G is a tree, entity matching is tractable , as opposed to Lemma 3. However, it remains hard to be parallelized, as we have shown in Theorem 4.

Proposition 5: *On trees, entity matching is in* PTIME. □

Parallel scalability. Not all is lost. Despite Theorems 2 and 4, we will show that there are effective parallel algorithms for entity matching. To assess the effectiveness of parallel algorithms, we introduce a simple notion.

We say that an algorithm $\mathcal{A}$ for entity matching is *parallel scalable* if its worst-case time complexity is $O(t(|G|, |\Sigma|)/p)$, where p is the number of processors used by $\mathcal{A}$, and t is a function in $|G|$ and $|\Sigma|$, the size of the input. We assume *w.l.o.g.* that $p \ll |G|$ as commonly found in practice.

This suffices. For if $\mathcal{A}$ is parallel scalable, then for given G and Σ, the more processors are used (*i.e.,* the larger p is), the less time $\mathcal{A}$ takes. Indeed, $t(,)$ is independent of p. Hence entity matching is feasible in big G by increasing p. Many parallel algorithms do not have provable guarantee for speedup *no matter how many processors* are added.

4. A MAPREDUCE ALGORITHM

We show that entity matching is feasible in big graphs.

Theorem 6: *There exist parallel scalable algorithms in MapReduce for entity matching.* □

As a proof, we present a parallel scalable algorithm in Section 4.1, followed by optimization strategies in Section 4.2.

4.1 Algorithm and Parallel Scalability

The algorithm, referred to as $\mathsf{EM}_{\mathsf{MR}}$ and shown in Fig. 4, takes as input a graph G and a set Σ of keys. It returns $\mathsf{chase}(G, \Sigma)$, the set of all pairs (e_1, e_2) if $(G, \Sigma) \models (e_1, e_2)$.

As opposed to subgraph isomorphism algorithms, $\mathsf{EM}_{\mathsf{MR}}$ has to compute the transitive closure (TC) of relation Eq, in which each step involves two subgraph isomorphism checks. By Theorem 4, this cannot be done in logarithmic rounds. Nonetheless, $\mathsf{EM}_{\mathsf{MR}}$ *combines* isomorphism checking and TC computation into the same MapReduce process. It ensures parallel scalability. Better still, it checks whether $(G, \Sigma) \models (e_1, e_2)$ *without* enumerating all isomorphic mappings.

$\mathsf{EM}_{\mathsf{MR}}$ starts with a set Eq consisting of (e, e) for all entities e in G, and a set L of *candidates*, *i.e.,* all entity pairs (e_1, e_2) having the same *type* on which at least one key in Σ is *defined*. We say that a key $Q(x)$ is defined on e if x and e have the same type. For all $(e_1, e_2) \in L$, it checks whether (e_1, e_2) is in Eq, or $(G, \Sigma) \models (e_1, e_2)$, *in parallel*. If so, it adds (e_1, e_2) to Eq, and incrementally extends the TC of Eq. Note that $(G, \Sigma) \models (e_1, e_2)$ *once* (e_1, e_2) can be identified by *one* key in Σ, no matter how many keys are defined on it. The process iterates until Eq no longer grows, *i.e.,* $\mathsf{chase}(G, \Sigma) = \mathsf{Eq}$. It takes at most $|\mathsf{Eq}|$ rounds of iterations.

Driver: $\mathsf{Driver}_{\mathsf{MR}}$

Input: Graph G and a set Σ of keys.
Output: $\mathsf{chase}(G, \Sigma)$.

1. construct candidate set L and d-neighbor G^d for each e in L;
2. initialize a set $\mathsf{Eq} := \{(e, e) \mid e \in G\}$;
3. **repeat**
4. call MapEM; ReduceEM;
5. **until** Eq no longer changes;
6. **return** Eq;

Mapper: MapEM

Input: A key/value pair $((e_1, e_2), (\mathsf{Flag}))$.
Output: Intermediate key/value pairs.

1. **if** Flag = True or $(G_1^d \cup G_2^d, \mathsf{Eq}, \Sigma) \models (e_1, e_2)$ **then**
2. emit $((e_1), (e_1, e_2, \mathsf{True}))$; emit $((e_2), (e_1, e_2, \mathsf{True}))$;
3. **else** emit $((e_1), (e_1, e_2, \mathsf{False}))$;

Reducer: ReduceEM

Input: A list of key/value pairs $((e), ([v_1, v_2, \cdots]))$.
Output: Key/value pairs $((e_1, e_2), (\mathsf{Flag}))$.

1. initialize $\mathsf{Eq}(e)$ and $L(e)$ with $\emptyset$;
2. **for each** v_i in $[v_1, v_2, \cdots]$ **do**
3. **if** $v_i = (e_1, e_2, \mathsf{True})$ **then** $\mathsf{Eq}(e) := \mathsf{Eq}(e) \cup \{(e_1, e_2)\}$;
4. **if** $v_i = (e_1, e_2, \mathsf{False})$ **then** $L(e) := L(e) \cup \{(e_1, e_2)\}$;
5. $\mathsf{Eq} := \mathsf{Eq} \cup \mathsf{Eq}(e)$;
6. **for each** (e_1, e_2) by joining pairs in $\mathsf{Eq}(e)$ and Eq, $(e_1, e_2) \notin \mathsf{Eq}$
7. emit $((e_1, e_2), (\mathsf{True}))$; $\mathsf{Eq} := \mathsf{Eq} \cup \{(e_1, e_2)\}$;
8. **for each** $(e_1, e_2) \in L(e)$ and $(e_1, e_2) \notin \mathsf{Eq}$ **do**
9. emit $((e_1, e_2), (\mathsf{False}))$;

Figure 4: Algorithm $\mathsf{EM}_{\mathsf{MR}}$

$\mathsf{EM}_{\mathsf{MR}}$ capitalizes on the following notions.

(1) The d-neighbor G^d of entity e. Let d be the maximum radius of those keys $Q(x)$ in Σ that are defined on e, and V_d be the set of nodes in G that are within d-hops of e. The *d-neighbor* of e is the subgraph of G induced by V_d.

To check $(G, \Sigma) \models (e_1, e_2)$, $\mathsf{EM}_{\mathsf{MR}}$ inspects the d-neighbors (G_1^d, G_2^d) of (e_1, e_2), *not* the entire G. Indeed, one can verify the *data locality*: $(G, \Sigma) \models (e_1, e_2)$ iff $(G_1^d \cup G_2^d, \Sigma) \models (e_1, e_2)$.

We check $(G_1^d \cup G_2^d, \Sigma) \models (e_1, e_2)$ by using Eq computed so far (see Section 3), denoted by $(G_1^d \cup G_2^d, \mathsf{Eq}, \Sigma) \models (e_1, e_2)$.

(2) Transitivity closure (TC). $\mathsf{EM}_{\mathsf{MR}}$ computes the TC of Eq with the following rule: if (e_1, e'_1), (e_2, e'_2) and (e'_1, e'_2) are in Eq, then so is (e_1, e_2); similarly for (e'_1, e_1) and (e'_2, e_2).

Algorithm. We now present the details of $\mathsf{EM}_{\mathsf{MR}}$. It is controlled by a non-MapReduce driver $\mathsf{Driver}_{\mathsf{MR}}$. $\mathsf{Driver}_{\mathsf{MR}}$ first identifies candidate set L (line 1). For each entity e appearing in L, it constructs d-neighbors G^d also in MapReduce, by revising breadth-first search starting from e, with bound d. To avoid the cost of shipping *invariant input* data in MapReduce, these d-neighbors G^d and keys Σ are *cached physically* in the disk of processors, along the same lines as Haloop [9]. In addition, it stores a "global variable" Eq in HDFS, to keep track of entity pairs identified by Σ (line 2).

It then triggers MapEM with key/value pairs $((e_1, e_2), (\mathsf{False}))$ for all $(e_1, e_2) \in L$ (line 4), with (e_1, e_2) as its key. MapReduce functions MapEM and ReduceEM then iterate to expand Eq. $\mathsf{Driver}_{\mathsf{MR}}$ *terminates* the process when there is no

change to Eq (line 5), and return Eq as chase(G, Σ) (line 6).

Mapper. Given a key/value pair $((e_1, e_2)$, (Flag)), MapEM first checks whether Flag = True (*i.e.,* $(e_1, e_2) \in$ Eq) or $(G_1^d \cup G_2^d$, Eq, $\Sigma) \models (e_1, e_2)$ (line 1) by invoking a procedure $\mathsf{Eval}_{\mathsf{MR}}$ (to be presented shortly). If so, MapEM emits value $(e_1, e_2,$ True) with keys e_1 and e_2, for computing TC (line 2). Otherwise, it emits value $(e_1, e_2,$ False) with key e_1 only (line 3), indicating the result of checking in this round.

Reducer. The input to ReduceEM is $(e,$ list), where list includes all *newly identified* and un-identified pairs, and are collected in Eq(e) and $L(e)$, respectively (lines 1-4). ReduceEM then adds Eq(e) to Eq (line 5) and *joins* pairs in Eq(e) and Eq (line 6), to compute TC following the rule we have seen earlier. For those *newly joined* pairs (e_1, e_2) not in Eq, ReduceEM emits $((e_1, e_2),$ True) to expand TC in the next round, and Eq is updated by including (e_1, e_2) (line 7). For each (e_1, e_2) in $L(e)$ but not in Eq (line 8-9), $((e_1, e_2),$ (False)) is emitted for checking in the next round. Note that for each pair $(e_1, e_2) \in$ Eq, if (e_1, e_2) is not newly identified in this round, (e_1, e_2) is no longer in the process.

One can verify the following by induction on the length of chasing sequences for $(G, \Sigma) \models (e_1, e_2)$ (see Section 3).

Proposition 7: *If* $(G, \Sigma) \models (e_1, e_2)$*, then* (e_1, e_2) *is identified by* $\mathsf{EM}_{\mathsf{MR}}$ *following the shortest chasing sequence.* □

Example 8: Algorithm $\mathsf{EM}_{\mathsf{MR}}$ works on G_1 and Σ_1 of Example 7 as follows. $\mathsf{Driver}_{\mathsf{MR}}$ triggers MapEM with $((\mathsf{alb}_i, \mathsf{alb}_j)$, (False)) and $((\mathsf{art}_i, \mathsf{art}_j)$, (False)), where $i, j \in [1, 3]$, $i < j$. Note that $d = 1$ for Q_1, Q_2 and Q_3 of Fig. 1.

Round 1. MapEM identifies alb_1 and alb_2 with key $Q_2(x)$ by procedure $\mathsf{Eval}_{\mathsf{MR}}$. ReduceEM adds $(\mathsf{alb}_1, \mathsf{alb}_2)$ to Eq, and joins it with Eq. They emit key/value pairs as follows.

MapEM	Emitted pairs	ReduceEM	Emitted pairs
$(\mathsf{alb}_1, \mathsf{alb}_2)$	$((\mathsf{alb}_1)$, $(\mathsf{alb}_1, \mathsf{alb}_2,$ T)) $((\mathsf{alb}_2)$, $(\mathsf{alb}_1, \mathsf{alb}_2,$ T))	alb_1	$((\mathsf{alb}_1, \mathsf{alb}_3)$, (F))
$(\mathsf{alb}_1, \mathsf{alb}_3)$	$((\mathsf{alb}_1)$, $(\mathsf{alb}_1, \mathsf{alb}_3,$ F))	alb_2	$((\mathsf{alb}_2, \mathsf{alb}_3)$, (F))
$(\mathsf{alb}_2, \mathsf{alb}_3)$	$((\mathsf{alb}_2)$, $(\mathsf{alb}_2, \mathsf{alb}_3,$ F))		
$(\mathsf{art}_1, \mathsf{art}_2)$	$((\mathsf{art}_1)$, $(\mathsf{art}_1, \mathsf{art}_2,$ F))	art_1	$((\mathsf{art}_1, \mathsf{art}_2)$, (F)) $((\mathsf{art}_1, \mathsf{art}_3)$, (F))
$(\mathsf{art}_1, \mathsf{art}_3)$	$((\mathsf{art}_1)$, $(\mathsf{art}_1, \mathsf{art}_3,$ F))	art_2	$((\mathsf{art}_2, \mathsf{art}_3)$, (F))
$(\mathsf{art}_2, \mathsf{art}_3)$	$((\mathsf{art}_2)$, $(\mathsf{art}_2, \mathsf{art}_3,$ F))		

Round 2. MapEM identifies $(\mathsf{art}_1, \mathsf{art}_2)$ by key $Q_3(x)$, and ReduceEM updates Eq. Note that no key/value pair for $(\mathsf{alb}_1, \mathsf{alb}_2)$ is in this round since it is in Eq already.

MapEM	Emitted pairs	ReduceEM	Emitted pairs
$(\mathsf{alb}_1, \mathsf{alb}_3)$	$((\mathsf{alb}_1)$, $(\mathsf{alb}_1, \mathsf{alb}_3,$ F))	alb_1	$((\mathsf{alb}_1, \mathsf{alb}_3)$, (F))
$(\mathsf{alb}_2, \mathsf{alb}_3)$	$((\mathsf{alb}_2)$, $(\mathsf{alb}_2, \mathsf{alb}_3,$ F))	alb_2	$((\mathsf{alb}_2, \mathsf{alb}_3)$, (F))
$(\mathsf{art}_1, \mathsf{art}_2)$	$((\mathsf{art}_1)$, $(\mathsf{art}_1, \mathsf{art}_2,$ T)) $((\mathsf{art}_2)$, $(\mathsf{art}_1, \mathsf{art}_2,$ T))	art_1	$((\mathsf{art}_1, \mathsf{art}_3)$, (F))
$(\mathsf{art}_1, \mathsf{art}_3)$	$((\mathsf{art}_1)$, $(\mathsf{art}_1, \mathsf{art}_3,$ F))	art_2	$((\mathsf{art}_2, \mathsf{art}_3)$, (F))
$(\mathsf{art}_2, \mathsf{art}_3)$	$((\mathsf{art}_2)$, $(\mathsf{art}_2, \mathsf{art}_3,$ F))		

Round 3. There is no newly identified entity pair, and Eq is not updated; $\mathsf{Driver}_{\mathsf{MR}}$ thus terminates the process, and returns chase(G_1, Σ_1) with $(\mathsf{alb}_1, \mathsf{alb}_2)$ and $(\mathsf{art}_1, \mathsf{art}_2)$. □

Procedure $\mathsf{Eval}_{\mathsf{MR}}$. We next show how to check $(G_1^d \cup G_2^d, \mathsf{Eq}, \Sigma) \models (e_1, e_2)$ with a key $Q(x)$ in Σ, in MapEM. A naive method is to first enumerate all matches of $Q(x)$ at e_1 in G_1^d and e_2 in G_2^d by using a subgraph isomorphism algorithm (*e.g.,* VF2 [13], TurboIso [24]), and then check whether any those matches *coincide* (see Section 2). This involves two calls for a subgraph isomorphism algorithm, each of exponential cost. In other words, it is *not practical* to conduct the checking by using *any existing algorithm.*

To reduce the cost, we propose algorithm $\mathsf{Eval}_{\mathsf{MR}}$ that *combines* the two processes of computing (isomorphic) mappings into *a single process*, and allows *early termination*, *i.e.,* $\mathsf{Eval}_{\mathsf{MR}}$ terminates as soon as (e_1, e_2) is identified.

$\mathsf{Eval}_{\mathsf{MR}}$ conducts search *guided by* $Q(x)$, to instantiate nodes in $Q(x)$ with candidate pairs (s_1, s_2) in (G_1^d, G_2^d). We use a vector m to record the instantiation, combining mappings ν_1 and ν_2 from variables or values of $Q(x)$ to entities or values in G_1^d and G_2^d, respectively, and mapping μ for coinciding the two (see Section 2). For each node s_Q in $Q(x)$, (a) either $m[s_Q] = (s_1, s_2)$ when $s_1 = \nu_1(s_Q)$, $s_2 = \nu_2(s_Q)$, and $s_1 = \mu(s_2)$; (b) or $m[s_Q] = \perp$ if s_Q has no match yet.

(1) Initialization. More specifically, $\mathsf{Eval}_{\mathsf{MR}}$ initializes m with $m[x] = (e_1, e_2)$ and $m[s_Q] = \perp$ for all the rest. It then instantiates nodes of m one by one, as follows.

(2) Feasibility checking. To extend m with $m[s_Q] = (s_1, s_2)$, it checks the following *feasibility conditions.*

(1) Injective: s_1 and s_2 do not appear in m already.

(2) Equality: (a) if s_Q is y, then $(s_1, s_2) \in$ Eq; (b) if s_Q is $y*$, then $s_1 = s_2$ (values); (c) if s_Q is $\_y$, then s_1 and s_2 are entities of the same type; and (d) if s_Q is d, then $s_1 = s_2 = d$ (values).

(3) Guided expansion: for all triples $(s_Q, p_Q, o_Q) \in Q(x)$, if o_Q is already instantiated, *i.e.,* $m[o_Q] = (o_1, o_2)$, then $(s_1, p_Q, o_1) \in G_1^d$ and $(s_2, p_Q, o_2) \in G_2^d$; similarly for all triples (s'_Q, p_Q, s_Q) in $Q(x)$.

$\mathsf{Eval}_{\mathsf{MR}}$ sets $m[s_Q] = (s_1, s_2)$ if *all* feasibility conditions are satisfied. Otherwise, it *backtracks* with other instantiation. Intuitively, m encodes a *partial* injective mapping from nodes in $Q(x)$ to candidate pairs in (G_1^d, G_2^d).

(3) Verification. When m is *fully instantiated*, *i.e.,* it contains no $\perp$, $\mathsf{Eval}_{\mathsf{MR}}$ concludes that $(G_1^d \cup G_2^d, \mathsf{Eq}, \{Q(x)\}) \models (e_1, e_2)$ and returns True. Otherwise, False.

Lemma 8: $(G, \{Q(x)\}) \models (e_1, e_2)$ *if and only if* m *can be fully instantiated by* $\mathsf{Eval}_{\mathsf{MR}}$*, using key* $Q(x)$. □

When Σ contains multiple keys, $\mathsf{Eval}_{\mathsf{MR}}$ identifies common sub-structures of keys along the same lines as [30]. It terminates *once there exists a key* $Q(x)$ that identifies (e_1, e_2).

Example 9: Continuing with Example 8, $\mathsf{Eval}_{\mathsf{MR}}$ identifies art_1 and art_2 with $Q_3(x)$ in round 2, after alb_1 and alb_2 are identified by $Q_2(x)$ in round 1. It initializes $m[x] = (\mathsf{art}_1, \mathsf{art}_2)$, and extends m with $m[\mathsf{name}*] =$ ("The Beatles", "The Beatles"), and $m[\mathsf{album}] = (\mathsf{alb}_1, \mathsf{alb}_2)$ after feasibility check. As m is fully instantiated, $\mathsf{Eval}_{\mathsf{MR}}$ returns True. □

Parallel scalability. To complete the proof of Theorem 6, we show that $\mathsf{EM}_{\mathsf{MR}}$ is parallel scalable. Let G_m^d be the largest d-neighbor of all entities in G, and p be the number of processors used. Then for each round of $\mathsf{EM}_{\mathsf{MR}}$, MapEM takes at most $O(t(|G_m^d|, |\Sigma|)|L|/p)$ time, and ReduceEM takes $O(|\mathsf{Eq}|^2/p)$ time, where $t(|G_m^d|, |\Sigma|)$ is the cost of $\mathsf{Eval}_{\mathsf{MR}}$. Moreover, at most $O(|\mathsf{Eq}|)$ rounds are needed since in each round, at least one pair is identified. Furthermore, $\mathsf{Driver}_{\mathsf{MR}}$ constructs all G_d's in $O((|G_m^d||L| + |\Sigma|)/p)$ time. Putting these together, $\mathsf{EM}_{\mathsf{MR}}$ is parallel scalable.

4.2 Optimization Strategies

From the analysis above, we can see that the cost of algorithm $\mathsf{EM}_{\mathsf{MR}}$ is dominated by (a) the length of L, (b) the size

of d-neighbors, and (c) redundant MapReduce computation. Below we study optimization strategies to reduce the cost.

Reducing L. Each $(e_1, e_2) \in L$ involves (repeated) isomorphism checking. Thus we filter those pairs that cannot be identified as follows. Given a key $Q(x)$, we say that (e_1, e_2) *can be paired by* $Q(x)$ if there exists a ternary relation P^Q on nodes of $(G_1^d, G_2^d, Q(x))$ such that (1) $(e_1, e_2, x) \in P^Q$, (2) for each triple $(s_1, s_2, s_Q) \in P^Q$, (a) s_1 and s_2 are entities with same type if s_Q is y or _y, $s_1 = s_2$ if s_Q is $y*$, or $s_1 = s_2 = d$ if $s_Q = d$; and (b) for each $(s_Q, p_Q, o_Q) \in Q(x)$, there exist (s_1, p_Q, o_1) in G_1^d and (s_2, p_Q, o_2) in G_2^d such that $(s_Q, p_Q, o_Q) \mapsto (s_1, p_Q, o_1)$, $(s_Q, p_Q, o_Q) \mapsto (s_2, p_Q, o_2)$, and $(o_1, o_2, o_Q) \in P^Q$; similarly for $(s'_Q, p_Q, s_Q) \in Q(x)$. We refer to P^Q as a *pairing relation* of Q at (e_1, e_2).

One can verify that pairing is a necessary condition for (e_1, e_2) to be identified by key $Q(x)$. Hence we include in L only those pairs that are paired by *some* key $Q(x) \in \Sigma$.

Proposition 9: *For any pair (e_1, e_2), (a) if e_1 and e_2 cannot be paired by any key in Σ, then $(G, \Sigma) \not\models (e_1, e_2)$; and (b) if (e_1, e_2) can be paired by a key $Q(x)$, then there exists a unique maximum pairing relation P^Q of $Q(x)$ at (e_1, e_2), and P^Q can be computed in $O(|Q||G_1^d||G_2^d|)$ time.* □

Reducing (G_1^d, G_2^d). For each $(e_1, e_2) \in L$, we reduce (G_1^d, G_2^d) such that they are subgraphs induced by those nodes that are in the maximum pairing relation P^Q at (e_1, e_2) by some key $Q(x)$ of Σ. Extending Proposition 9, one can verify that $(G, \Sigma) \models (e_1, e_2)$ if and only if (e_1, e_2) can be identified by keys in (G_1^d, G_2^d) constructed in this way.

Reducing redundant MapReduce computation. We develop two optimization strategies by leveraging the dependency imposed by recursively defined keys. We say that a pair (e_1, e_2) *depends on* (e'_1, e'_2) if (e'_1, e'_2) is (a) in d-neighbors of (e_1, e_2); and (b) has the same type as y, where y is a variable in a recursive key in Σ defined on (e_1, e_2).

Entity dependency. $\mathsf{Driver}_{\mathsf{MR}}$ collects a set L_0 with pairs $(e_1, e_2) \in L$, such that only value-based keys in Σ are defined on. $\mathsf{Driver}_{\mathsf{MR}}$ triggers MapEM with pairs in L_0 *only*, instead of the entire L. In each MapReduce round, a new pair (e'_1, e'_2) is emitted only when (e'_1, e'_2) depends on (e_1, e_2), and if (e_1, e_2) has been already proceeded.

Incremental checking. We revise MapEM such that $(G_1^d \cup G_2^d, \mathsf{Eq}, \Sigma) \models (e_1, e_2)$ is checked only in the first round or when some pairs (e'_1, e'_2) on which (e_1, e_2) depends are *identified in the last round*, to reduce the expensive checking. This is done by adding a flag Changed to the pairs in Eq.

5. A VERTEX-CENTRIC ALGORITHM

The performance of algorithm $\mathsf{EM}_{\mathsf{MR}}$ is hampered by (1) the maintenance of global variable Eq; and (2) stragglers in each round that may hold up the process of a chain of entity pairs on which dependencies are imposed by recursively defined keys. Such costs are inherent to the I/O bound property and the synchronization policy of MapReduce.

To reduce the costs, we develop an algorithm for entity matching in the vertex-centric model of [31]. As opposed to MapReduce, [31] is based on a vertex program that is executed in parallel *on each vertex*, and interacts with the neighbors of the vertex via asynchronous message passing. There is no need for a global variable Eq, or for synchronizing the computation into rounds. We show the following.

Theorem 10: *There exist parallel scalable algorithms in the vertex-centric model of [31] for entity matching.* □

As a proof, we present such an algorithm (Section 5.1), and develop optimization strategies (Section 5.2).

5.1 Algorithm and Parallel Scalability

The algorithm, referred to as $\mathsf{EM}_{\mathsf{VC}}$, computes $\mathsf{chase}(G, \Sigma)$ when given a graph G and a set Σ of keys. For all $(e_1, e_2) \in L$, it checks whether $(G, \Sigma) \models (e_1, e_2)$. Similar to $\mathsf{EM}_{\mathsf{MR}}$, $\mathsf{EM}_{\mathsf{VC}}$ adds (e_1, e_2) to Eq *once* it is identified by any key in Σ. In contrast to $\mathsf{EM}_{\mathsf{MR}}$, $\mathsf{EM}_{\mathsf{VC}}$ follows *asynchronous message passing* [31]. To determine whether $(G, \Sigma) \models (e_1, e_2)$, it checks *different instantiations* of nodes in a key *in parallel* with multiple messages, for all keys defined on (e_1, e_2).

When $(G, \Sigma) \models (e_1, e_2)$ is confirmed, $\mathsf{EM}_{\mathsf{VC}}$ notifies those pairs $(s_1, s_2) \in L$ that depend on (e_1, e_2) by sending messages, so that $(G, \Sigma) \models (s_1, s_2)$ is checked "incrementally". The transitive closure (TC) of Eq is computed by message propagation at the same time. The process proceeds until no messages are active and Eq can no longer be changed.

The key ideas behind $\mathsf{EM}_{\mathsf{VC}}$ include *guided search* for verifying $(G, \Sigma) \models (e_1, e_2)$ and *expansion* of TC based on the dependency of entities, both via asynchronous message passing. To facilitate message passing, $\mathsf{EM}_{\mathsf{VC}}$ uses the following.

Product graph. Given G and Σ, $\mathsf{EM}_{\mathsf{VC}}$ constructs a *product graph* $G_p = (V_p, E_p)$, where each node in V_p is either (a) a pair (o_1, o_2) of entities or values that can be *paired* (see Proposition 9); or (b) a pair (e, e) of entities *only if* e is paired with another entity in V_p. There is an edge $((s_1, s_2), p, (o_1, o_2))$ in E_p from node (s_1, s_2) to (o_1, o_2) if (a) (s_1, p, o_1) and (s_2, p, o_2) are both in G; (b) (o_1, o_2) depends on (s_1, s_2) (see Section 4.2); here p is a special label dep; or (c) $o_1 \Leftrightarrow o_2$, and $o_1 \Leftrightarrow s_1$ or $o_1 \Leftrightarrow s_2$; p is labeled as tc in this case.

Intuitively, G_p encodes the topology of G, the dependency on entities *w.r.t.* Σ via dep edges, and the transitive closure of Eq via tc edges. We do not include (e, e) in G_p if e is not in L. In our experiments, we find that $|G_p| = 2.7 * |G|$ on average, *much smaller than* $|G|^2$.

For each (e_1, e_2) in G_p, a Boolean $\mathsf{Flag}(e_1, e_2)$ is used to indicate whether $(e_1, e_2) \in \mathsf{Eq}$, initially False unless $e_1 \Leftrightarrow e_2$.

Traversal order. For each key $Q(x)$ in Σ, $\mathsf{EM}_{\mathsf{VC}}$ defines a sorted list P_Q of triples in $Q(x)$ such that (a) *all nodes* in $Q(x)$ appear in some triples in P_Q, and (b) it encodes a "tour" of nodes in $Q(x)$, starting from x and ending at x.

Intuitively, $\mathsf{EM}_{\mathsf{VC}}$ propagates messages *guided by* P_Q. Together with feasibility checking to be seen shortly, a complete tour that starts from (e_1, e_2) guided by P_Q guarantees that (e_1, e_2) can be identified by $Q(x)$. There are multiple orders for a tour of $Q(x)$. However, finding an optimum order with a shortest tour is NP-complete, by reduction from *Chinese Postman Problem* (cf. [19]). In light of this, $\mathsf{EM}_{\mathsf{VC}}$ uses a greedy algorithm to decide P_Q.

Algorithm. $\mathsf{EM}_{\mathsf{VC}}$ first constructs G_p as above. Then at each node (e_1, e_2) in V_p, if a value-based key in Σ is defined on it, it triggers procedure $\mathsf{Eval}_{\mathsf{VC}}$ for subgraph isomorphism checking, propagates messages to activate other nodes in V_p guided by traversal order, and computes the TC of Eq. $\mathsf{EM}_{\mathsf{VC}}$ *terminates* when no messages are active, and it returns Eq of all pairs (e_1, e_2) with $\mathsf{Flag}(e_1, e_2) = \mathsf{True}$, as $\mathsf{chase}(G, \Sigma)$.

Procedure $\mathsf{Eval}_{\mathsf{VC}}$. At each node (s_1, s_2) in G_p, the actions of $\mathsf{EM}_{\mathsf{VC}}$ are summarized in $\mathsf{Eval}_{\mathsf{VC}}$, shown in Fig. 5.

Algorithm $\mathsf{Eval_{VC}}$ /* Executed at each node (s_1, s_2) */

(1) Initial messages at (s_1, s_2)

1. **for each** key $Q(x) \in \Sigma$ defined on (s_1, s_2) **do**
2. create an initial message $m_Q(s_1, s_2)$;
3. propagate $m_Q(s_1, s_2)$ guided by order P_Q;

(2) Upon receiving a message $m_Q(e_1, e_2)$ *following* (s_Q, p_Q, o_Q)

1. **if** $\mathsf{Flag}(e_1, e_2) = \mathsf{True}$ **then**
2. stop propagating $m_Q(e_1, e_2)$; **return**;
3. **if** $m_Q(e_1, e_2)$ is fully instantiated and $(e_1, e_2) = (s_1, s_2)$ **then**
4. $\mathsf{Flag}(e_1, e_2) := \mathsf{True}$; compute dependency and TC; **return**;
5. **if** either $m_Q(e_1, e_2)[s_Q]$ or $m_Q(e_1, e_2)[o_Q]$ is $\perp$ **then**
6. **if** $m_Q(e_1, e_2)$ satisfies all feasibility conditions at (s_1, s_2) **then**
7. extend $m_Q(e_1, e_2)$ by instantiating a node with (s_1, s_2);
8. **else** drop $m_Q(e_1, e_2)$; **return**;
9. propagate $m_Q(e_1, e_2)$ guided by order P_Q;

(3) Compute dependency and TC when $\mathsf{Flag}(e_1, e_2)$ *becomes* True

1. **if** $((e_1, e_2), \mathsf{dep}, (s_1, s_2)) \in G_p$, and $\mathsf{Flag}(s_1, s_2) = \mathsf{False}$ **then**
2. propagate increment message $m_{Q'}(s_1, s_2)$ for each $Q'(x)$ of Σ;
3. **if** $((e_1, e_2), \mathsf{tc}, (s_1, s_2)) \in G_p$ **then**
4. compute transitive closure of Eq;

Figure 5: Algorithm $\mathsf{Eval_{VC}}$

(1) Initial message. When $\mathsf{Eval_{VC}}$ is activated at a node (s_1, s_2) in G_p, for each key $Q(x) \in \Sigma$ defined on (s_1, s_2), an *initial message* $m_Q(s_1, s_2)$ is created (lines 1-2, (1), Fig. 5), with $m_Q(s_1, s_2)[x] = (s_1, s_2)$ and $m_Q(s_1, s_2)[s_Q] = \perp$ for all other nodes in $Q(x)$. The message is a vector that encodes a partial injective mapping from nodes in $Q(x)$ to nodes in G_p, similar to those used by procedure $\mathsf{Eval_{MR}}$ (Section 4.1).

Then *guided* by the *first* triple (x, p_Q, o_Q) (or (s_Q, p_Q, x)) of P_Q, a *copy* of $m_Q(e_1, e_2)$ is "forked" to propagate to each neighbor (o_1, o_2) of (s_1, s_2), following edge $((s_1, s_2), p_Q, (o_1, o_2))$ in G_p (line 3), for feasibility check (see (4) below).

(2) Early cancellation. Upon receiving a message $m_Q(e_1, e_2)$ at (s_1, s_2), (s_1, s_2) first checks whether $\mathsf{Flag}(e_1, e_2)$ is True, by sending a message to (e_1, e_2), whose ID is in $m_Q(e_1, e_2)$. If so, $\mathsf{Eval_{VC}}$ stops the propagation of $m_Q(e_1, e_2)$ (lines 1-2, (2), Fig. 5), since (e_1, e_2) is already identified.

(3) Verification. If $\mathsf{Flag}(e_1, e_2)$ is False, but $m_Q(e_1, e_2)$ is *fully instantiated*, *i.e.,* it does not contain $\perp$, and moreover, if (e_1, e_2) is (s_1, s_2), *i.e.,* $m_Q(e_1, e_2)$ has completed its propagation and is sent back to (e_1, e_2), guaranteed by the guided order P_Q, then we can conclude that $(G, \{Q(x)\}) \models (e_1, e_2)$ (see Lemma 11 below). Hence $\mathsf{Flag}(e_1, e_2)$ is set True, (e_1, e_2) *notifies* nodes that depend on (e_1, e_2) following edges labeled dep, and activates those nodes following edges labeled tc, to compute the TC of Eq (lines 3-4, see (6) and (7) below).

(4) Feasibility checking. Otherwise, assume that $m_Q(e_1, e_2)$ is sent to (s_1, s_2) following triple (s_Q, p_Q, o_Q) in P_Q. If $m_Q(e_1, e_2)[s_Q] = \perp$ (similarly for $m_Q(e_1, e_2)[o_Q] = \perp$), $\mathsf{Eval_{VC}}$ checks whether $m_Q(e_1, e_2)[s_Q]$ can be instantiated with (s_1, s_2) (lines 5-6) based on the same feasibility conditions of $\mathsf{Eval_{MR}}$ (*injective*, *equality* and *guided expansion*; Section 4.1), except that when s_Q is a variable y, it requires $\mathsf{Flag}(s_1, s_2) = \mathsf{True}$. If it does not pass the check, $m_Q(e_1, e_2)$ is *dropped* (line 8), as $m_Q(e_1, e_2)$ cannot be expanded. Otherwise $\mathsf{Eval_{VC}}$ sets $m_Q(e_1, e_2)[s_Q] = (s_1, s_2)$ (line 7).

(5) Guided propagation. Now, both $m_Q(e_1, e_2)[s_Q]$ and $m_Q(e_1, e_2)[o_Q]$ are instantiated. Then (s_1, s_2) propagates message $m_Q(e_1, e_2)$ guided by the *next* triple (s'_Q, p'_Q, o'_Q) in P_Q, *i.e.,* the successor of (s_Q, p_Q, o_Q) in P_Q (line 9). Assuming that $m_Q(e_1, e_2)[s'_Q] = (s_1, s_2)$ (the case is similar if $m_Q(e_1, e_2)[o'_Q] = (s_1, s_2)$), $\mathsf{Eval_{VC}}$ does the following.

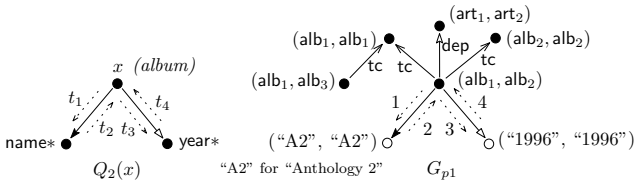


Figure 6: Message propagation in $\mathsf{Eval_{VC}}$

(a) If $m_Q(e_1, e_2)[o'_Q] = (o_1, o_2)$, *i.e.,* message $m_Q(e_1, e_2)$ has already been instantiated with (o_1, o_2), then $m_Q(e_1, e_2)$ is sent "back" to (o_1, o_2) directly.

(b) If $m_Q(e_1, e_2)[o'_Q] = \perp$, a *copy* of $m_Q(e_1, e_2)$ is propagated to each neighbor (o_1, o_2) of (s_1, s_2), following edge $((s_1, s_2), p_Q, (o_1, o_2))$ in G_p. If no such neighbor exists, $m_Q(e_1, e_2)$ is dropped.

One can verify that $m_Q(e_1, e_2)$ is propagated only *within* the d-neighbor of (e_1, e_2) in G_p as it is guided by P_Q.

(6) Dependency. When $\mathsf{Flag}(e_1, e_2)$ is set True, $\mathsf{Eval_{VC}}$ notifies all nodes (s_1, s_2) that depend on (e_1, e_2), by following edge $((e_1, e_2), \mathsf{dep}, (s_1, s_2))$ (see (3) above). Then $\mathsf{Eval_{VC}}$ is activated at (s_1, s_2). It checks whether $\mathsf{Flag}(s_1, s_2)$ is False (line 1, (3), Fig. 5). If so, $\mathsf{Eval_{VC}}$ triggers *increment messages* $m_{Q'}(s_1, s_2)$ for each $Q'(x)$ defined on (s_1, s_2), with $m_{Q'}(s_1, s_2)[x] = (s_1, s_2)$, $m_{Q'}(s_1, s_2)[y] = (e_1, e_2)$ and $m_{Q'}(s_1, s_2)[z_Q] = \perp$ for other nodes in $Q'(x)$, where y is a variable in $Q'(x)$ with the same type of (e_1, e_2) (line 2). These messages are propagated in the same way as above.

(7) Transitive closure. When $\mathsf{Flag}(e_1, e_2)$ is True, (s_1, s_2) is notified if $((e_1, e_2), \mathsf{tc}, (s_1, s_2)) \in G_p$. Assume *w.l.o.g.* that $(s_1, s_2) = (e_1, e_1)$. Then at (e_1, e_1), (e_1, e_2) is *joined* with (e_1, e), when either (a) $e \Leftrightarrow e_1$ or (b) $((e_1, e), \mathsf{tc}, (e_1, e_1)) \in G_p$ and $\mathsf{Flag}(e_1, e) = \mathsf{True}$; it sets $\mathsf{Flag}(e_2, e) = \mathsf{True}$. The newly identified nodes conduct the same process following tc edges, to further compute the TC (lines 3-4, (3), Fig. 5).

The correctness of $\mathsf{Eval_{VC}}$ is warranted by the following.

Lemma 11: $(G, \{Q(x)\}) \models (e_1, e_2)$ *if and only if there exists a message* $m_Q(e_1, e_2)$ *that can be fully instantiated by* $\mathsf{Eval_{VC}}$*; the message is propagated at most* $2|Q|$ *times.* □

Example 10: We show how $\mathsf{EM_{VC}}$ works on G_1 and Σ_1 of Example 7. A (partial) product graph G_{p_1} of G_1 is shown in Fig. 6, where $(\mathsf{art}_1, \mathsf{art}_2)$ depends on $(\mathsf{alb}_1, \mathsf{alb}_2)$.

For Q_2, the order P_{Q_2} is $[t_1, t_2, t_3, t_4]$, where t_1 and t_2 are $(x, \mathsf{name\_of}, \mathsf{name}*)^{+/-}$, and t_3 and t_4 are $(x, \mathsf{release\_year}, \mathsf{year}*)^{+/-}$, respectively; here $+$ and $-$ indicate forward and backward traversal, respectively. At $(\mathsf{alb}_1, \mathsf{alb}_2)$, $\mathsf{Eval_{VC}}$ constructs initial message m_{Q_2} for $Q_2(x)$, where $m_{Q_2}[x] = (\mathsf{alb}_1, \mathsf{alb}_2)$, and $\perp$ for the other nodes. As shown in Fig. 6, it propagates m as follows, guided by P_{Q_2}.

Node (m_{Q_2} visits)	Feasibility checking	P_{Q_2}
("A2", "A2")	$m_{Q_2}[\mathsf{name}*] =$ ("A2", "A2")	t_2
$(\mathsf{alb}_1, \mathsf{alb}_2)$	$m_{Q_2}[x]$ is instantiated	t_3
("1996", "1996")	$m_{Q_2}[\mathsf{year}*] =$ ("1996", "1996")	t_4
$(\mathsf{alb}_1, \mathsf{alb}_2)$	$\mathsf{Flag}(\mathsf{alb}_1, \mathsf{alb}_2) = \mathsf{True}$	

When m_{Q_2} is sent back to $(\mathsf{alb}_1, \mathsf{alb}_2)$, it is fully instantiated, and $\mathsf{Flag}(\mathsf{alb}_1, \mathsf{alb}_2)$ is set True. $\mathsf{Eval_{VC}}$ then notifies node $(\mathsf{art}_1, \mathsf{art}_2)$ via edge labeled dep, triggers an increment message m_{Q_3} for $Q_3(x)$ there, and identifies $(\mathsf{art}_1, \mathsf{art}_2)$ along the same lines. While some other nodes are notified by following tc edges for computing TC, no new entity pairs are derived. At this point, no message is in transit, and $\mathsf{EM_{VC}}$ returns all entity pairs with $\mathsf{Flag} = \mathsf{True}$. □

Parallel scalability. We show that algorithm $\mathsf{EM}_{\mathsf{VC}}$ is parallel scalable. The total amount of computation by $\mathsf{EM}_{\mathsf{VC}}$ is at most $O(t(|G_p^d|, |\Sigma|)|L||\mathsf{Eq}|)$, where G_p^d is the maximum d-neighbor of entity pairs in G_p and $O(t(|G_p^d|, |\Sigma|))$ is the time for checking $(G, \Sigma) \models (e_1, e_2)$ via message passing. Each pair may be checked $|\mathsf{Eq}|$ times in the worst case. Assume that the work is distributed evenly across p processors, *i.e.,* the resources of an idle node are re-allocated to process other nodes as conducted in the vertex-centric model [31], and that $p \ll |G|$. Then $\mathsf{EM}_{\mathsf{VC}}$ is in $O(t(|G_p^d|, |\Sigma|)|L||\mathsf{Eq}|/p)$ time.

From this and Lemma 11, Theorem 10 follows.

5.2 Optimization Strategies

$\mathsf{Eval}_{\mathsf{VC}}$ may fork excessive messages and incur redundant computation. To reduce the cost, we adopt prior optimizations [30] to extract common sub-structures of keys in Σ. Moreover, we present another two strategies to reduce it.

Bounded messages. To check $(G, \{Q(x)\}) \models (s_1, s_2)$, $\mathsf{Eval}_{\mathsf{VC}}$ generates at most k messages, for a (user-defined) constant k. To do this, we revise $\mathsf{Eval}_{\mathsf{VC}}$ as follows.

(1) When $\mathsf{Eval}_{\mathsf{VC}}$ is activated at (s_1, s_2), a variable $K_Q(s_1, s_2)$ is defined to keep track of the number of copies of $m_Q(s_1, s_2)$ that are active, initially 1 for the initial message.

(2) Suppose that $m_Q(e_1, e_2)[s_Q]$ is instantiated with (s_1, s_2) (while $m_Q(e_1, e_2)[o_Q] = \perp$). $\mathsf{Eval}_{\mathsf{VC}}$ propagates $m_Q(e_1, e_2)$ guided by a triple (s_Q, p_Q, o_Q) in P_Q as follows.

- If $K_Q(e_1, e_2) < k$, for each edge $((s_1, s_2), p_Q, (o_1, o_2))$ in G_p that is yet *unmarked* with (s_Q, p_Q, o_Q) for $m_Q(e_1, e_2)$, a new copy of $m_Q(e_1, e_2)$ is propagated to (o_1, o_2), and $K_Q(e_1, e_2)$ is increased by 1, until $K_Q(e_1, e_2) = k$ or all unmarked edges are covered.
- Otherwise (if there is no budget for new copies), $m_Q(e_1, e_2)$ is propagated following an unmarked edge $((s_1, s_2), p_Q, (o_1, o_2))$, without forking new copies.

Those edges $((s_1, s_2), p_Q, (o_1, o_2))$ that message $m_Q(e_1, e_2)$ follows are *marked* with (s_Q, p_Q, o_Q) for $m_Q(e_1, e_2)$, to avoid repeated checking. The process is similar if $m_Q(e_1, e_2)[o_Q] = (s_1, s_2)$ and $m_Q(e_1, e_2)[s_Q] = \perp$.

(3) When there are no nodes to propagate, or the feasibility conditions are not satisfied, $m_Q(e_1, e_2)$ will *backtrack* to check other instantiation, instead of being dropped.

In this way, to check whether $(G, \Sigma) \models (e_1, e_2)$, at most $O(k\|\Sigma\||\mathsf{Eq}|)$ messages are generated and propagated.

Prioritized propagation. When $\mathsf{Eval}_{\mathsf{VC}}$ picks an unmarked edge to propagate message $m_Q(e_1, e_2)$ from (s_1, s_2), it selects an edge with the highest *potential* that can make $m_Q(e_1, e_2)$ fully instantiated. This is estimated based on the number of neighbors of (o_1, o_2) that have the same types and values as those variables in $m_Q(e_1, e_2)$ to be instantiated. Such information is collected when constructing G_p.

6. EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we conducted three sets of experiments on $\mathsf{EM}_{\mathsf{MR}}$ and $\mathsf{EM}_{\mathsf{VC}}$ to evaluate the impacts of (1) the number p of processors used; (2) the size of graph G; and (3) the complexity of keys Σ (see below). The results verify that the algorithms are parallel scalable and can efficiently identify entities in reasonably large graphs.

Experimental setting. We used two real-life graphs: (a) *Google+* [21] (*Google* in short), a social network with 2.6 million nodes and 17.5 million edges (relationships such as *friend*), where 30 types of entities are determined by its node attributes, *e.g., major*, *university*, *place* and *employer*; and (b) *DBpedia* [1], a knowledge base with 4.3 million nodes and 40.3 million links, including 495 types of entities.

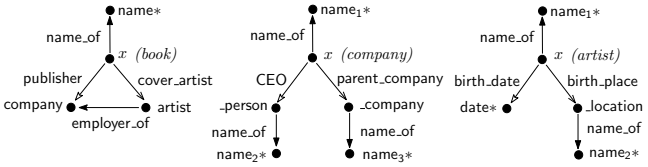


Figure 7: Keys defined on DBpedia

We also developed a generator to produce synthetic graphs G, controlled by the number of entities $\mathcal{E}$ and data values $\mathcal{D}$. Predicates $\mathcal{P}$ and entity types Θ were drawn from an alphabet $\mathcal{L}$ of 6000 labels. The size of G is up to 95 million entities (100 million nodes) and 500 million edges.

Key generator. We generated keys Σ controlled by the maximum radius d and the length c of longest *dependency chains* from recursively defined keys in Σ. (1) We constructed 30 and 100 keys for *Google* and *DBpedia*, respectively, with attributes and predicates from the data graphs. Some keys for *DBpedia* are shown in Fig. 7. (2) For synthetic graphs, we randomly generated 500 keys for different types of entities in Θ, with values from $\mathcal{D}$ and predicates from $\mathcal{P}$.

Algorithms. We implemented the following algorithms: (1) MapReduce algorithms on Hadoop 1.2.1: (a) $\mathsf{EM}_{\mathsf{MR}}$ of Section 4.1, (b) $\mathsf{EM}_{\mathsf{MR}}^{\mathsf{VF2}}$, which replaces $\mathsf{Eval}_{\mathsf{MR}}$ of $\mathsf{EM}_{\mathsf{MR}}$ with VF2 [13] by enumerating all matches without early termination; (c) $\mathsf{EM}_{\mathsf{MR}}^{\mathsf{Opt}}$, a revision of $\mathsf{EM}_{\mathsf{MR}}$ by supporting the optimization strategies of Section 4.2. (2) Vertex-centric algorithms on GraphLab [31]: (a) $\mathsf{EM}_{\mathsf{VC}}$ of Section 5.1, and (b) $\mathsf{EM}_{\mathsf{VC}}^{\mathsf{Opt}}$, which optimizes $\mathsf{EM}_{\mathsf{VC}}$ by using $k = 4$ messages and prioritized message propagation strategy (Section 5.2). Conventional algorithms for subgraph isomorphism algorithms and entity resolution do not work on entity matching and graphs, respectively, and hence, cannot be compared with.

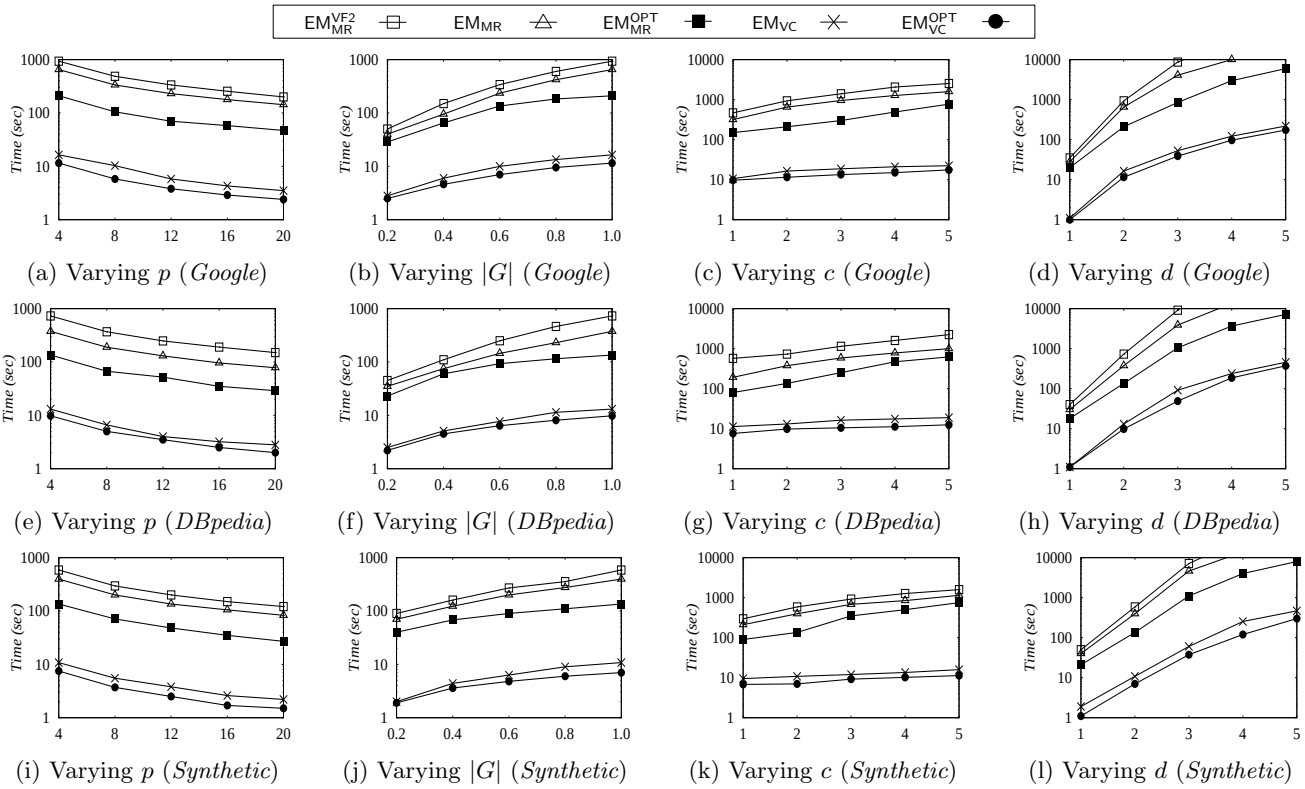
Distributed sites. We deployed the graphs, keys and algorithms on $p \in [4, 20]$ machines of Amazon EC2 Compute-Optimized Instance c4.4xlarge. Each experiment was run 3 times and the average is reported here.

Experimental results. We next report our findings. In all the experiments, we used 30, 100 and 500 keys for *Google*, *DBpedia* and *Synthetic* respectively.

Exp-1: Varying p. Fixing $c = 2$ and $d = 2$, we first evaluated the parallel scalability of these algorithms by varying p from 4 to 20. The results are reported in Figures 8(a), 8(e) and 8(i), for *Google*, *DBpedia* and *Synthetic* (fixing $G = (100M, 500M)$), respectively, in which we use *logarithmic scale* for the y-axis. We find the following.

Parallel scalability. On a given graph, these algorithms took less time proportional to the increase of processors. For instance, $\mathsf{EM}_{\mathsf{VC}}^{\mathsf{Opt}}$ (resp. $\mathsf{EM}_{\mathsf{MR}}^{\mathsf{Opt}}$) are 4.8, 4.7 and 5 times faster (resp. 4.6, 4.7 and 4.8) when p increases from 4 to 20 on *Google*, *DBpedia* and *Synthetic*, respectively. We find that $\mathsf{EM}_{\mathsf{VC}}^{\mathsf{Opt}}$ scales the best among all the algorithms: it takes 2.4 seconds to identify all entities in *Google* with 20 processors.

We also experimented with p up to 32. The results are consistent with Figures 8(a), 8(e) and 8(i): the algorithms are 1.5 times faster than the setting with $p = 20$ on average.



(a) Varying p (*Google*) (b) Varying $|G|$ (*Google*) (c) Varying c (*Google*) (d) Varying d (*Google*)

(e) Varying p (*DBpedia*) (f) Varying $|G|$ (*DBpedia*) (g) Varying c (*DBpedia*) (h) Varying d (*DBpedia*)

(i) Varying p (*Synthetic*) (j) Varying $|G|$ (*Synthetic*) (k) Varying c (*Synthetic*) (l) Varying d (*Synthetic*)

Figure 8: Performance evaluation

These experimentally verify Theorems 6 and 10.

MapReduce vs. vertex-centric. Algorithm $\mathsf{EM}_{\mathsf{VC}}$ outperforms all the MapReduce algorithms, even $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$. It is at least 12.1, 10.9 and 13.5 times faster on *Google*, *DBpedia* and *Synthetic*, respectively. For instance, it takes 5.8 seconds on *Google* when $p = 12$, while $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ takes 70 seconds. This verifies that $\mathsf{EM}_{\mathsf{VC}}$ reduces the inherent costs of the I/O bound and the synchronization policy of MapReduce.

We developed and evaluated $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ and $\mathsf{EM}_{\mathsf{MR}}$ because of the prevalent use of the MapReduce framework. Moreover, $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ may be advantageous to $\mathsf{EM}_{\mathsf{VC}}$ when $\mathsf{EM}_{\mathsf{VC}}$ requires a product graph much larger than G (see Section 5 and below).

Effectiveness of optimization. (1) $\mathsf{EM}_{\mathsf{MR}}$ is 1.5, 1.9 and 1.4 times faster than $\mathsf{EM}^{\mathsf{VF2}}_{\mathsf{MR}}$ on average on *Google*, *DBpedia* and *Synthetic*, respectively. This verifies the effectiveness of procedure $\mathsf{Eval}_{\mathsf{MR}}$ (Section 4.1) that employs guided expansion and early termination for subgraph isomorphism checking.

(2) Compared with $\mathsf{EM}_{\mathsf{MR}}$, $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ is at least 3.2, 2.9 and 3 times faster on *Google*, *DBpedia* and *Synthetic*, respectively. These verify the effectiveness of our optimization strategies: on average, (a) L is reduced 52%, 38% and 45%, (b) G^d is 2.5, 1.7 and 2.1 times smaller; and (c) it reduces 23%, 15% and 20% of redundant subgraph isomorphism checking *in each MapReduce round* by leveraging dependency and incremental checking, on the three datasets, respectively.

(3) Compared with $\mathsf{EM}_{\mathsf{VC}}$, $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ is 1.5 times faster on average when $k = 4$ on *Google*; similarly for *DBpedia* and *Synthetic*. These verify the effectiveness of bounded messages and prioritized message propagation (Section 5.2).

Table 2 shows the numbers of candidate and confirmed matches checked by $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ in the three datasets.

Datasets	Candidate Matches		Confirmed Matches
	$\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$	$\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$	
Google	24500	11760	1620
DBpedia	22615	15380	1357
Synthetic	20000	11000	1000

Table 2: Candidate matches vs. confirmed matches

Exp-2: Varying $|G|$. Fixing $p = 4$, $c = 2$ and $d = 2$, we varied $|G|$ with scale factors from 0.2 to 1 for *Google*, *DBpedia* and *Synthetic*. As shown in Figures 8(b), 8(f) and 8(j), (1) all the algorithms take longer on larger $|G|$, as expected; (2) $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ performs the best among all of them, and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ outperforms the other MapReduce algorithms; these are consistent with the results of Exp-1; (3) for product graphs G_p used by $\mathsf{EM}_{\mathsf{VC}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$, $|G_p| = 2.7 * |G|$ on average, which is much smaller than $|G|^2$; and (4) $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ are reasonably efficient: when $G = (40M, 200M)$ for *Synthetic*, they take 68 and 3.6 seconds respectively, with 4 processors; the results are similar on *Google* and *DBpedia*.

Exp-3: Varying Σ. Finally, we evaluated the impact of Σ, by varying the longest chain c and maximum radius d in Σ.

Varying c. Fixing $p = 4$ and $d = 2$, we varied c from 1 to 5. As shown in Figures 8(c), 8(g) and 8(k) for *Google*, *DBpedia* and *Synthetic* ($|G| = (100M, 500M)$), respectively, (1) all the algorithms take longer on larger c, (2) the number of MapReduce rounds increases from 2 to 9, for all MapReduce algorithms; and (3) $\mathsf{EM}_{\mathsf{VC}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ are less sensitive to c; this is because by asynchronous message passing, these algorithms do not separate computation into "rounds" and avoid the "blocking" of stragglers in each MapReduce round.

Varying d. Fixing $p = 4$ and $c = 2$, we varied d from 1 to 5. As reported in Figures 8(d), 8(h) and 8(l) for *Google*, *DBpedia* and *Synthetic* ($|G| = (100M, 500M)$), respectively, (1) d

is a major factor for the costs: all the algorithms take longer on larger d; and (2) the pairing strategy is effective as the d-neighbors of $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ are 60%, 42%, 53% smaller than those of $\mathsf{EM}_{\mathsf{MR}}$, and it makes $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ 4.8, 3.7 and 4.2 times faster than $\mathsf{EM}_{\mathsf{MR}}$ on average, when $d = 3$, on the three graphs, respectively. We find that keys often have a small radius in real life. This is analogous to real-life SPARQL queries: 98% of them have radius 1, and 1.8% have radius 2 [18].

Summary. We find the following. (1) Our algorithms scale well with the increase of processors: $\mathsf{EM}_{\mathsf{MR}}$, $\mathsf{EM}_{\mathsf{VC}}$, $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ are $4.8, 4.8, 4.7$ and 4.9 times faster on average when p increases from 4 to 20. (2) Our algorithms perform well on large graphs and complex Σ: on graphs with $G = (100M, 500M)$, Σ with 500 keys, $c = 2$, $d = 2$, $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ take 27 and 1.5 seconds on average with 20 processors, respectively. (3) Our optimization techniques are effective: $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ are 3 and 1.5 times faster than $\mathsf{EM}_{\mathsf{MR}}$ and $\mathsf{EM}_{\mathsf{VC}}$ on average, and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ is 4.8 times faster than $\mathsf{EM}^{\mathsf{VF2}}_{\mathsf{MR}}$. (4) $\mathsf{EM}_{\mathsf{VC}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{VC}}$ perform better than $\mathsf{EM}_{\mathsf{MR}}$ and $\mathsf{EM}^{\mathsf{Opt}}_{\mathsf{MR}}$ by reducing unnecessary costs inherent to MapReduce.

7. CONCLUSIONS

We have proposed a class of keys for graphs. We have shown that entity matching with keys is NP-complete and hard to parallelize. Despite these, we have provided two parallel scalable algorithms, and our experimental results have verified that entity matching is feasible in practice.

One topic for future work is to develop efficient algorithms for discovering keys. Another topic is to adapt keys to various applications with different pattern matching semantics.

Acknowledgments. Fan is supported in part by NSFC 61133002, 973 Program 2012CB316200 and 2014CB340302, ERC-2014-AdG 652976, Guangdong Innovative Research Team Program 2011D005, Shenzhen Peacock Program 1105100030834361, EPSRC EP/J015377/1 and EP/M025268/1, NSF III 1302212, and a Google Faculty Research Award.

8. REFERENCES

[1] Dbpedia. *http://wiki.dbpedia.org/Downloads2014*.
[2] Full version. *http://homepages.inf.ed.ac.uk/s1368930/keys.pdf*.
[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[4] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, 2011.
[5] F. N. Afrati and C. H. Papadimitriou. The parallel complexity of simple logic programs. *J. ACM*, 40(4), 1993.
[6] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using Dedupalog. In *ICDE*, 2009.
[7] O. Benjelloun, H. Garcia-Molina, H. Gong, H. Kawai, T. Larson, D. Menestrina, and S. Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *ICDCS*, 2007.
[8] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007.
[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 2010.
[10] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. In *WWW*, 2001.
[11] P. Buneman and G. Silvello. A Rule-Based Citation System for Structured and Evolving Datasets. *IEEE Data Eng. Bull.*, 33(3):33–41, 2010.
[12] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *TKDE*, 24, 2012.
[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.
[14] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. *SIGMOD*, 2005.
[15] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *PVLDB*, 2014.
[16] X. L. Dong, K. Murphy, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD*, 2014.
[17] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDBJ*, 2011.
[18] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *USEWOD workshop*, 2011.
[19] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
[20] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. *PVLDB*, 5(12), 2012.
[21] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. IMC '12, 2012.
[22] E. L. Goodman and D. Grunwald. Using vertex-centric programming platforms to implement SPARQL queries on large graphs. IA3, pages 25–32, 2014.
[23] R. V. Guha. Communicating and resolving entity references. *http://arxiv.org/abs/1406.6973*.
[24] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348, 2013.
[25] M. Herschel, F. Naumann, S. Szott, and M. Taubert. Scalable iterative graph duplicate detection. *TKDE*, 2012.
[26] S.-H. Kim, K.-H. Lee, H. Choi, and Y.-J. Lee. Parallel processing of multiple graph queries using MapReduce. In *DBKDA*, pages 33–38, 2013.
[27] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient deduplication with hadoop. *PVLDB*, 2012.
[28] N. Korula and S. Lattanzi. An efficient reconciliation algorithm for social networks. *PVLDB*, 7(5), 2014.
[29] N. Lao, T. Mitchell, and W. W. Cohen. Random walk inference and learning in a large scale knowledge base. In *EMNLP*, 2011.
[30] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *ICDE*, 2012.
[31] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
[32] P. Malhotra, P. Agarwal, and G. Shroff. Graph-parallel entity resolution using LSH & IMM. In *EDBT/ICDT Workshops*, 2014.
[33] N. Pernelle, F. Saïs, and D. Symeonidou. An automatic key discovery approach for data linking. *J. Web Sem.*, 23, 2013.
[34] N. Preda, G. Kasneci, F. M. Suchanek, T. Neumann, W. Yuan, and G. Weikum. Active knowledge: dynamically enriching RDF knowledge bases by web services. In *SIGMOD*, 2010.
[35] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee. PGX.ISO: Parallel and efficient in-memory engine for subgraph isomorphism. GRADES, 2014.
[36] V. Rastogi, N. Dalvi, and M. Garofalakis. Large-scale collective entity matching. *PVLDB*, 2011.
[37] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 2013.
[38] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 2012.