Keys for Graphs

Citation for published version:

Digital Object Identifier (DOI):
10.14778/2824032.2824056

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the VLDB Endowment (PVLDB)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Keys for Graphs

Wenfei Fan\textsuperscript{1,2}, Zhe Fan\textsuperscript{3}, Chao Tian\textsuperscript{1,2}, Xin Luna Dong\textsuperscript{4}
\textsuperscript{1}University of Edinburgh, \textsuperscript{2}Beihang University, \textsuperscript{3}Hong Kong Baptist University, \textsuperscript{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 $\Sigma$ of keys, is to find all pairs of entities (vertices) in $G$ that are identified by keys in $\Sigma$. 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.

Music. Consider a knowledge base $G_1$ consisting of triples $(s, p, o)$, indicating subject, predicate and object, respectively; e.g., (\textit{album}, \textit{recorded} by, \textit{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 \textit{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\footnote{We require exact match in the examples for simplicity; nonetheless, we can easily relax the constraints to similarity match.}. 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 \textit{artist} may not be identified by its \textit{name}. Indeed, there are 6 artists or bands named “Everest”. Nonetheless, a key for \textit{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 \textit{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 \textit{album} entities $x_1$ and $x_2$ have the same name and are \textit{recorded} by the same \textit{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 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 $\Sigma$ of keys for graphs, entity matching is to find all pairs of entities (vertices) in $G$ that can be identified by keys in $\Sigma$. 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 fix-point 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 $t(G)$ 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 techniques to reduce message passing.

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.

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

A graph $G$ is a set of triples $t = (s, p, o)$, where subject $s$ is an entity in $E$, $p$ is a predicate in $P$, and object $o$ is either an entity in $E$ or a value in $D$. 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 $E$: $e_1 \Leftrightarrow e_2$, if entities $e_1$ and $e_2$ have the same ID, i.e., they refer to the same entity; and

(b) value equality on $D$: $d_1 = d_2$ 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 $(\text{art}_1, \text{name}_{of}, \text{"The Beatles"})$, subject $\text{art}_1$ is an entity of type $\text{artist}$, and object "The Beatles" is a value; in $G_1$, both are represented as nodes, and the triple is presented as an edge labeled $\text{name}_{of}$ from $\text{art}_1$ 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 $y$ or a variable $x$, and $p_q$ is a predicate in $P$. 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, $Q_1(x)$ represents triples (x, name$_{of}$, name$*$), (com$_{company}$, name$_{of}$, name$*$) and (com$_{company}$, parent$_{of}$, x). Here $x$ is the designated variable.
A valuation of $Q(x)$ in a set $S$ of triples is a mapping $ν$ from $Q(x)$ to $S$ that preserves values in $D$ and predicates in $P$, and maps variables $y$ and $y'$ to entities of the same type. More specifically, for each triple $(ν(s), ν(p), ν(o))$ in $Q(x)$, there exists $(s, p, o)$ in $S$, written as $(ν(s), ν(p), ν(o)) \mapsto (s, p, o)$ or simply $(ν(s), ν(p), ν(o)) \mapsto (s, p, o)$, where

(a) $ν(s) = s$, $p = p$, $ν(o) = o$;
(b) $o$ is an entity if $o_Q$ is a variable 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 $S$ and $ν$ such that $ν(x) = e$ 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 $G_2$: $(\{com_1$, name.of, “AT&T”$\}$, $(com_3$, name.of, “AT&T”$\})$, $(com_2$, parent.of, com_3$$\})$, $(com_3$, parent.of, com_4$\})$. Then $S_1$ is a match of $Q_4(x)$ in $G_2$ at com_3, which maps variable $x$ to com_3, name+ to “AT&T”, wildcard $\_company$ to com_1, and company to com_3.

### 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, $Q_3$ and $Q_5$ 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 $S_1(e_1) \equiv_Q S_2(e_2)$, if a bijection $μ$ between $S_1$ and $S_2$ can be derived from $ν_1$ and $ν_2$, preserving node identity and value identity. That is, for each $(ν(s), ν(p), ν(o))$ in $Q(x)$ such that $(ν(s), ν(p), ν(o)) \mapsto (s, p, o)$ and $(ν(s), ν(p), ν(o)) \mapsto (s, p, o)$, it satisfies conditions:

(a) if $s$ is a variable $s_Q$ that is distinct from $x$, then $s \Leftrightarrow s_Q$; similarly for $o_Q$; and
(b) if $o_Q$ is a variable $y$, then $o_1 = o_2$.
3.1 Entity Matching with Keys

Example 5 shows that $G_2 \not\equiv Q_4(x)$, since $S_1(\text{com}_4) \not\subseteq Q_4$, $S_2(\text{com}_3)$, but $\text{com}_2$ and $\text{com}_3$ are distinct. However, key $Q_4(x)$ tells us that $\text{com}_2$ and $\text{com}_3$ 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 $\Sigma$ of keys and a graph $G$. We use $\text{Eq}$ to denote the equivalence relation of a set of pairs $(e_1, e_2)$ of entities in $G$ of the same type that have been identified by keys in $\Sigma$, i.e., $\text{Eq}$ is reflexive, symmetric and transitive. We denote by $\text{Eq}_0$ the node identity relation $\Leftrightarrow$, i.e., the set of pairs $(e_1, e_2)$ for all $e_i$ 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 $\nu_1$ and $\nu_2$, respectively. We define $S_1(e_1) \equiv_{\text{Eq}}^\Sigma S_2(e_2)$ by using $\text{Eq}$ instead of relation $\Leftrightarrow$ in the definition of $S_1(e_1) \equiv_{\text{Eq}} S_2(e_2)$. More specifically, for each triple $(s_0, p_0, q_0)$ in $Q$, if $(s_0, p_0, q_0) \mapsto_{\nu_1} (s_1, p_1, o_1)$ and $(s_0, p_0, q_0) \mapsto_{\nu_2} (s_2, p_2, o_2)$, then

(a) if $s_0$ is a variable $y$ distinct from $x$, then $(s_1, s_2) \in \text{Eq}$ (instead of $s_1 \Leftrightarrow s_2$); similarly for $q_0$; and
(b) if $q_0$ is a variable $y$, then $o_1 = o_2$.

We define a chase step of $G$ by $\Sigma$ as $\text{Eq}$ at

$$\text{Eq} = \{(e_1, e_2) \mid e_1, e_2 \in \text{Eq}_0\},$$

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

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

A chasing sequence of $G$ by $\Sigma$ is a sequence

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

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

Chasing keys of the Church-Rosser property.

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

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

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

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

Problem. The entity matching problem is stated as follows.

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

3.2 The Complexity of Entity Matching

Given a set of keys on a relation $R$, it is in $\text{PTIME}$ to find all pairs of tuples in $R$ that are identified by the keys. In contrast, the entity matching problem is non-trivial. To see this, consider its decision problem, also referred to as entity matching, which is to determine, given $\Sigma$, $G$ 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 $\Sigma$ 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 $\Sigma$ 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 $\text{PTIME}$ 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 $\text{PTIME}$.

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 $\text{PTIME}$. 

\[ \begin{align*}
\text{com}_1 & \equiv (\text{com}_2, \text{com}_0, \text{com}_3) \\
\text{com}_2 & \equiv (\text{com}_1, \text{com}_2) \\
\end{align*} \]
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 $G_e$ with $(4N + 3)$ entities and a set $Σ$ consisting of three keys such that to identify a specific entity pair in $G_e$, 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 $Σ$ of four keys. We show that the output value of a gate $l$ is true iff $(G, Σ) ⊨ (e_l, e_i)$, where $(e_l, e_i)$ 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 $A$ for entity matching is parallel scalable if its worst-case time complexity is $O(t(|G|, |Σ|)/p)$, where $p$ is the number of processors used by $A$, and $t$ is a function in $|G|, |Σ|$, the size of the input. We assume w.l.o.g. that $p \ll |G|$ as commonly found in practice. This suffices. For if $A$ is parallel scalable, then for given $G$ and $Σ$, the more processors are used (i.e., the larger $p$ is), the less time $A$ takes. Indeed, $t(\cdot)$ 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 EM$_{MR}$ and shown in Fig. 4, takes as input a graph $G$ and a set $Σ$ of keys. It returns chase($G, Σ$), the set of all pairs $(e_1, e_2)$ if $(G, Σ) ⊨ (e_1, e_2)$.

As opposed to subgraph isomorphism algorithms, EM$_{MR}$ 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, EM$_{MR}$ combines isomorphism checking and TC computation into the same MapReduce process. It ensures parallel scalability. Better still, it checks whether $(G, Σ) ⊨ (e_1, e_2)$ without enumerating all isomorphic mappings.

EM$_{MR}$ 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 $(e_1, e_2) ∈ L$, it checks whether $(e_1, e_2)$ is in Eq, or $(G, Σ) ⊨ (e_1, e_2)$, in parallel. If so, it adds $(e_1, e_2)$ to Eq, and incrementally extends the TC of Eq. Note that $(G, Σ) ⊨ (e_1, e_2)$ 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., chase($G, Σ$) = Eq. It takes at most $|Eq|$ rounds of iterations.

EM$_{MR}$ 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 $(G, Σ) ⊨ (e_1, e_2)$, EM$_{MR}$ 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: $(G, Σ) ⊨ (e_1, e_2)$ iff $(G^1_d, G^2_d, Σ) ⊨ (e_1, e_2)$.

We check $(G^1_d ∪ G^2_d, Σ) ⊨ (e_1, e_2)$ by using Eq computed so far (see Section 3), denoted by $(G^1_d ∪ G^2_d, Σ) ⊨ (e_1, e_2)$.

(2) Transitivity closure (TC). EM$_{MR}$ 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 $(e_1', e_2)$ and $(e_2', e_2)$.

Algorithm. We now present the details of EM$_{MR}$. It is controlled by a non-MapReduce driver DRIVER$_{MR}$. DRIVER$_{MR}$ first identifies candidate set $L$ (line 1). For each entity $e$ appearing in $L$, it constructs $d$-neighbors $G^d$ also in MapReduce, by reversing 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 Hadoop [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 $(e_1, e_2)$, (False)) for all $(e_1, e_2) ∈ L$ (line 4), with $(e_1, e_2)$ as its key. MapReduce functions MapEM and ReduceEM then iterate to expand Eq. DRIVER$_{MR}$ 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 ((e₁, e₂), (Flag)), MapEM first checks whether Flag = True (i.e., (e₁, e₂) ∈ Eq) or (G₁, G₂, Eq, Σ) = (e₁, e₂) (line 1) by invoking a procedure EvalMR (to be presented shortly). If so, MapEM emits value (e₁, e₂, True) with key e₁ and e₂, for computing TC (line 2). Otherwise, it emits value (e₁, e₂, False) with key e₁ 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 (e₁, e₂) not in Eq, ReduceEM emits ((e₁, e₂), True) to expand TC in the next round, and Eq is updated by including (e₁, e₂) (line 7). For each (e₁, e₂) in L(e) but not in Eq (line 8-9), ((e₁, e₂), (False)) is emitted for checking in the next round. Note that for each pair (e₁, e₂) ∈ Eq, if (e₁, e₂) is not newly identified in this round, (e₁, e₂) is no longer in the process.

One can verify the following by induction on the length of the sequence for (G, Σ) = (e₁, e₂) (see Section 3).

**Proposition 7:** If (G, Σ) = (e₁, e₂), then (e₁, e₂) is identified by EMₘᵣ following the shortest chasing sequence. □

**Example 8:** Algorithm EMₘᵣ works on G₁ and Σ₁ of Example 7 as follows. DRIVERₘᵣ triggers MapEM with ((alb₁, alb₂), (False)) and ((art₁, art₂), (False)) where i,j ∈ [1, 3], i ≤ j. Note that d = 1 for Q₁, Q₂ and Q₃ of Fig. 1.

Round 1. MapEM identifies alb₁ and alb₂ with key Q₁(x) by procedure EvalMR. ReduceEM adds (alb₁, alb₂) to Eq, and joins it with Eq. They emit key/value pairs as follows.

<table>
<thead>
<tr>
<th>MapEM</th>
<th>Emitted pairs</th>
<th>ReduceEM</th>
<th>Emitted pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(alb₁, alb₂)</td>
<td>((alb₁, alb₁, alb₂, 1))</td>
<td>alb₂</td>
<td>((alb₁, alb₁), (F))</td>
</tr>
<tr>
<td>(alb₁, alb₂)</td>
<td>((alb₁, alb₂, alb₂, 1))</td>
<td>alb₁</td>
<td>((alb₂, alb₂), (F))</td>
</tr>
<tr>
<td>(alb₁, alb₂)</td>
<td>((alb₂, alb₂, alb₁, 1))</td>
<td>alb₂</td>
<td>((alb₁, alb₁), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₁, art₁, art₂, 2))</td>
<td>art₁</td>
<td>((art₁, art₂), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₂, art₁, art₂, 2))</td>
<td>art₁</td>
<td>((art₂, art₂), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₂, art₁, art₂, 2))</td>
<td>art₁</td>
<td>((art₂, art₁), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₁, art₂, art₂, 3))</td>
<td>art₂</td>
<td>((art₁, art₂), (F))</td>
</tr>
</tbody>
</table>

Round 2. MapEM identifies (art₁, art₂) by key Q₂(x), and ReduceEM updates Eq. Note that no key/value pair for (alb₁, alb₂) is in this round since it is in Eq already.

<table>
<thead>
<tr>
<th>MapEM</th>
<th>Emitted pairs</th>
<th>ReduceEM</th>
<th>Emitted pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(alb₁, alb₂)</td>
<td>((alb₁, alb₁, alb₂, 1))</td>
<td>alb₁</td>
<td>((alb₁, alb₁), (F))</td>
</tr>
<tr>
<td>(alb₁, alb₂)</td>
<td>((alb₂, alb₁, alb₂, 1))</td>
<td>alb₂</td>
<td>((alb₂, alb₂), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₁, art₁, art₂, 2))</td>
<td>art₁</td>
<td>((art₁, art₂), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₂, art₁, art₂, 2))</td>
<td>art₂</td>
<td>((art₂, art₂), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₂, art₁, art₂, 2))</td>
<td>art₁</td>
<td>((art₂, art₁), (F))</td>
</tr>
<tr>
<td>(art₁, art₂)</td>
<td>((art₁, art₂, art₂, 3))</td>
<td>art₁</td>
<td>((art₁, art₂), (F))</td>
</tr>
</tbody>
</table>

Round 3. There is no newly identified entity pair, and Eq is not updated; DRIVERₘᵣ thus terminates the process, and returns chase(G₁, Σ₁) with (alb₁, alb₂) and (art₁, art₂). □

**Procedure EvalMR.** We next show how to check (G₁, G₂, Eq, Σ) = (e₁, e₂) with a key Q₁(x) in Σ, in MapEM. A naive method is to first enumerate all matches of Q₁(x) at e₁ in G₁ and e₂ in G₂ by using a subgraph isomorphism algorithm (e.g., VF2 [13], TurboReo [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 Evalₘᵣ that combines the two processes of computing (isomorphic) mappings into a single process, and allows early termination, i.e., Evalₘᵣ terminates as soon as (e₁, e₂) is identified. Evalₘᵣ conducts search guided by Qₙ(x), to instantiate nodes in Qₙ(x) with candidate pairs (s₁, s₂) in (G₁, G₂). We use a vector m to record the instantiation, combining mappings v₁ and v₂ from variables or values of Qₙ(x) to entities or values in G₁ and G₂, respectively, and mapping μ for coinciding the two (see Section 2). For each node sₙ in Qₙ(x), (a) either m[sₙ] = (s₁, s₂) when s₁ = v₁(sₙ), s₂ = v₂(sₙ), and s₁ = μ(s₂); (b) or m[sₙ] = ⊥ if sₙ has no match yet.

1. **Initialization.** More specifically, Evalₘᵣ initializes m with m[x] = (e₁, e₂) and m[sₙ] = ⊥ for all the rest. It then instantiates nodes of m one by one, as follows.

2. **Feasibility checking.** To extend m with m[sₙ] = (s₁, s₂), it checks the following feasibility conditions.

   (1) Injective: s₁ and s₂ do not appear in m already.

   (2) Equality: (a) if sₙ is y, then (s₁, s₂) ∈ Eq; (b) if sₙ is y*, then s₁ = sₙ = s₂ (values); (c) if sₙ is y, then s₁ and s₂ are entities of the same type; and (d) if sₙ is d, then s₁ = s₂ = d (values).

   (3) Guided expansion: for all triples (s₉, p₉, o₉) in Qₙ(x), if o₉ is already instantiated i.e., m[o₉] = (o₁, o₂), then (s₁, p₉, o₁) ∈ G₁ and (s₂, p₉, o₂) ∈ G₂; similarly for all triples (s₉, p₉, s₉) in Qₙ(x).

   Evalₘᵣ sets m[sₙ] = (s₁, s₂) if all feasibility conditions are satisfied. Otherwise, it backtracks with other instantiation. Intuitively, m encodes a partial injective mapping from nodes in Qₙ(x) to candidate pairs in (G₁, G₂).

   (3) Verification. When m is fully instantiated, i.e., it contains no ⊥, Evalₘᵣ concludes that (G₁ ∪ G₂, Eq, (Qₙ(x))) = (e₁, e₂) and returns True. Otherwise, False.

   **Lemma 8:** (G, {Qₙ(x)}) = (e₁, e₂) if and only if m can be fully instantiated by Evalₘᵣ, using key Qₙ(x).

When Σ contains multiple keys, Evalₘᵣ identifies common sub-structures of keys along the same lines as [30]. It terminates once there exists a key Qₙ(x) that identifies (e₁, e₂).

**Example 9:** Continuing with Example 8, Evalₘᵣ identifies art₁ and art₂ with Q₃(x) in round 2, after alb₁ and alb₂ are identified by Q₂(x) in round 1. It initializes m[x] = (art₁, art₂), and extends m with m[album] = ("The Beatles", "The Beatles"), and m[album] = (alb₁, alb₂) after feasibility check. As m is fully instantiated, Evalₘᵣ returns True. □

**Parallel scalability.** To complete the proof of Theorem 6, we show that EMₘᵣ is parallel scalable. Let Gₘᵣ be the largest d-neighbor of all entities in G, and p be the number of processors used. Then for each round of EMₘᵣ, MapEM takes at most O(t(Gₘᵣ, [Σ])) time, and ReduceEM takes O(|Eq|²/p) time, where t(Gₘᵣ, [Σ]) is the cost of EMₘᵣ. Moreover, at most O(|Eq|) rounds are needed since in each round, at least one pair is identified. Furthermore, DRIVERₘᵣ constructs all Gₘᵣ’s in O(|Gₘᵣ| + |Σ|)/p time. Putting these together, EMₘᵣ is parallel scalable.

**4.2 Optimization Strategies.** From the analysis above, we can see that the cost of algorithm EMₘᵣ 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 $P^{Q(x)}$ on nodes of $(G_1^p, G_2^p, Q(x))$ such that (1) $(e_1, e_2, x) \in P^{Q(x)}$, (2) for each triple $(s_1, s_2, s_q) \in P^{Q(x)}$, (a) $s_1$ and $s_2$ are entities with the same type if $s_q$ is $y$ or $\omega$, $s_1 = s_2$ 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 $(s_1, p_q, o_q)$ in $G_1^p$ and $(s_2, p_q, o_q)$ in $G_2^p$ such that $(s_q, p_q, o_q) \rightarrow (s_1, p_q, o_1)$, $(s_q, p_q, o_q) \rightarrow (s_2, p_q, o_2)$, and $(o_1, o_2, o_q) \in P^{Q(x)}$; similarly for $(s_q, p_q, o_q) \in Q(x)$. We refer to $P^{Q(x)}$ 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 $\Sigma$, 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 $P^{Q(x)}$ of $Q(x)$ at $(e_1, e_2)$, and $P^{Q(x)}$ can be computed in $O(|Q(x)|G_1^pG_2^p)$ time.

Reducing $(G_1^p, G_2^p)$. For each $(e_1, e_2) \in L$, we reduce $(G_1^p, G_2^p)$ such that they are subgraphs induced by those nodes that are in the maximum pairing relation $P^{Q(x)}$ at $(e_1, e_2)$ by some key $Q(x)$ of $\Sigma$. 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 $(G_1^p, G_2^p)$ 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 $d$-neighbor of $(e_1, e_2)$; and (b) has the same type as $y$, where $y$ is a variable in a recursive key in $\Sigma$ defined on $(e_1, e_2)$.

**Entity dependency.** Driver$_{MR}$ collects a set $L_0$ with pairs $(e_1, e_2) \in L$, such that only value-based keys in $\Sigma$ are defined on. Driver$_{MR}$ 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 $(G_1^p \cup G_2^p, Eq, \Sigma) \models (e_1, e_2)$ 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 EM$_{MR}$ 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 EM$_{VC}$, computes chase($G, \Sigma$) when given a graph $G$ and a set $\Sigma$ of keys. For all $(e_1, e_2) \in L$, it checks whether $(G, \Sigma) \models (e_1, e_2)$. Similar to EM$_{MR}$, EM$_{VC}$ adds $(e_1, e_2)$ to Eq once it is identified by any key in $\Sigma$. In contrast to EM$_{MR}$, EM$_{VC}$ 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, EM$_{VC}$ 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 EM$_{VC}$ 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, EM$_{VC}$ uses the following.

**Product graph.** Given $G$ and $\Sigma$, EM$_{VC}$ 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 $G_p$; (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. $\Sigma$ 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 $|G_p| \approx 2.7 \times |G|$ on average, much smaller than $|G|^2$.

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

**Traversal order.** For each key $Q(x)$ in $\Sigma$, EM$_{VC}$ 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, EM$_{VC}$ 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, EM$_{VC}$ uses a greedy algorithm to decide $P_Q$.

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

**Procedure EvalVC.** At each node $(s_1, s_2) \in G_p$, the actions of EM$_{VC}$ are summarized in EvalVC, shown in Fig. 5.
Algorithm Evalvc /* Executed at each node (s1, s2) */

1. Initial messages at (s1, s2)
   1. For each key Q(x) ∈ Σ defined on (s1, s2) do
   2. Create an initial message mQ(s1, s2);
   3. Propagate mQ(s1, s2) guided by order Pq;

2. Upon receiving a message mQ(e1, e2) following (sQ, pq, oQ)
   1. If Flag(e1, e2) = True then
      2. Stop propagating mQ(e1, e2); return;
      3. If mQ(e1, e2) is fully instantiated and (e1, e2) = (s1, s2) then
         4. Flag(e1, e2) := True; compute dependency and TC; return;
      5. If either mQ(e1, e2)[oQ] or mQ(e1, e2)[oQ] is ⊥ then
         6. Compute dependency and TC when Flag(e1, e2) becomes True
   7. If (e1, e2), dep, (s1, s2) ∈ Gp, and Flag(s1, s2) = False then
      8. Propagate incremental message mQ(e1, s2) for each Q(x) of Σ;
   9. Compute transitive closure of Eq;

Figure 5: Algorithm Evalvc

1. Initial message. When Evalvc is activated at a node (s1, s2) in Gp, for each key Q(x) ∈ Σ defined on (s1, s2), an initial message mQ(s1, s2) is created (lines 1-2, (1), Fig. 5), with mQ(s1, s2)[x] = (s1, s2) and mQ(s1, s2)[sQ] = ⊥ 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 Gp, similar to those used by Procedure Evalvc (Section 4.1).

Then guided by the first triple (x, pq, oQ) (or (sQ, pq, x)) of Pq, a copy of mQ(e1, e2) is “forked” to propagate to each neighbor (o1, o2) of (s1, s2), following edge ((s1, s2), pq, (o1, o2)) in Gp (line 3), for feasibility check (see (4) below).

2. Early cancellation. Upon receiving a message mQ(e1, e2) at (s1, s2), (s1, s2) first checks whether Flag(e1, e2) is True, by sending a message to (e1, e2), whose ID is in mQ(e1, e2). If so, Evalvc stops the propagation of mQ(e1, e2) (lines 1-2, (2), Fig. 5), since (e1, e2) is already identified.

3. Verification. If Flag(e1, e2) is False, but mQ(e1, e2) is fully instantiated, i.e., it does not contain ⊥, and moreover, if (e1, e2) is (s1, s2), i.e., mQ(e1, e2) has completed its propagation and is sent back to (e1, e2), guaranteed by the guided order Pq, then we can conclude that (m, {Q(x)}) = (e1, e2) (see Lemma 11 below). Hence Flag(e1, e2) is set True, (e1, e2) notifies nodes that depend on (e1, e2) 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 mQ(e1, e2) is sent to (s1, s2) following triple (sQ, pq, oQ) in Pq. If mQ(e1, e2)[oQ] = ⊥ (similarly for mQ(e1, e2)[oQ] = ⊥), Evalvc checks whether mQ(e1, e2)[oQ] can be instantiated with (s1, s2) (lines 5-6) based on the feasibility conditions of Evalvc (injective, equality and guided expansion; Section 4.1), except that when sQ is a variable y, it requires Flag(s1, s2) = True. If it does not pass the check, mQ(e1, e2) is dropped (line 8), as mQ(e1, e2) cannot be expanded. Otherwise Evalvc sets mQ(e1, e2)[oQ] = (s1, s2) (line 7).

5. Guided propagation. Now, both mQ(e1, e2)[oQ] and mQ(e1, e2)[oQ] are instantiated. Then (s1, s2) propagates message mQ(e1, e2) guided by the next triple (sQ, pq, oQ) in Pq, i.e., the successor of (sQ, pq, oQ) in Pq (line 9). Assuming that mQ(e1, e2)[oQ] = (s1, s2) (the case is similar if mQ(e1, e2)[oQ] = (s1, s2)), Evalvc does the following.

---

One can verify that mQ(e1, e2) is propagated only within the d-neighbor of (e1, e2) in Gp as it is guided by Pq.

6. Dependency. When Flag(e1, e2) is set True, Evalvc notifies all nodes (s1, s2) that depend on (e1, e2), by following edge ((e1, e2), dep, (s1, s2)) (see (3) above). Then Evalvc is activated at (s1, s2). It checks whether Flag(s1, s2) is False (line 1, (3), Fig. 5). If so, Evalvc triggers increment messages mQ(e1, s2) for each Q(x) defined on (s1, s2), with mQ(s1, s2)[x] = (s1, s2), mQ(s1, s2)[y] = (e1, e2), and mQ(s1, s2)[zQ] = ⊥ for other nodes in Q(x), when y is a variable in Q(x) with the same type of (e1, e2) (line 2). These messages are propagated in the same way as above.

7. Transitive closure. When Flag(e1, e2) is True, (s1, s2) is notified if ((e1, e2), tc, (s1, s2)) ∈ Gp. Assume w.l.o.g. that (s1, s2) = (e1, e1). Then at (e1, e1), (e1, e2) is joined with (e1, e1) when either a ∈ vte or (b) ((e1, e1), tc, (e1, e1)) ∈ Gp and Flag(e1, e2) = True; it sets Flag(e1, e2) = 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 Evalvc is warranted by the following.

Lemma 11: (G, {Q(x)}) |= (e1, e2) if and only if there exists a message mQ(e1, e2) that can be fully instantiated by Evalvc; the message is propagated at most 2|Q| times.

---

Example 10: We show how EMVC works on G1 and Σ1 of Example 7. A (partial) product graph Gp of G1 is shown in Fig. 6, where (art1, art2) depends on (alb1, alb2).

For Q2, the order Pq2 is {t1, t2, t3, t4}, where t1 and t2 are (x, name backbone, artist)+/−, and t3 and t4 are (y, release year, artist)+/−, respectively; here + and − indicate forward and backward traversal, respectively. At (alb1, alb2), Evalvc constructs initial message mQ2 for Q2(x), where mQ2(x) = (alb1, alb2), and ⊥ for the other nodes. As shown in Fig. 6, it propagates m as follows, guided by Pq2:

---

When Q2 is sent back to (alb1, alb2), it is fully instantiated, and Flag(alb1, alb2) is set True. Evalvc then notifies node (art1, art2) via edge labeled dep, triggers an increment message mQ2 for Q2(x) there, and identifies (art1, art2) 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 EMVC returns all entity pairs with Flag = True.

---
Parallel scalability. The total amount of computation by EMVC is parallel scalable. The total amount of computation by EMVC is at most $O(t((|G_p|, |\Sigma|) |L| |\text{Eq}|)$, where $G_p$ is the maximum $d$-neighbor of entity pairs in $G_p$ and $O((|G_p|, |\Sigma|)$ is the time for checking $(G, \Sigma) \models (e_1, e_2)$ via message passing. Each pair may be checked $|\text{Eq}|$ 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 EMVC is in $O((|G_p|, |\Sigma|) |L| |\text{Eq}|/p)$ time.

From this and Lemma 11, Theorem 10 follows.

5.2 Optimization Strategies

EMVC 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 $\Sigma$. Moreover, we present another two strategies to reduce it.

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

1. When EMVC is activated at $(s_1, s_2)$, a variable $K_Q(s_1, s_2)$ is defined to keep track of the number of messages 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 $m_Q(e_1, e_2)[o_Q] = \bot$). EMVC 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 $m_Q(e_1, e_2)[s_Q] = \bot$.

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 $O(k |\Sigma| |L| |\text{Eq}|)$ messages are generated and propagated.

Prioritized propagation. When EMVC 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 EMVR and EMVC to evaluate the impacts of (1) the number $p$ of processors used; (2) the size of graph $G$; and (3) the complexity of keys $\Sigma$ (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.

We also developed a generator to produce synthetic graphs $G$, controlled by the number of entities $E$ and data values $D$. Predicates $P$ and entity types $\Theta$ were drawn from an alphabet $L$ 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 $\Sigma$ controlled by the maximum radius $d$ and the length $c$ of longest dependency chains from recursively defined keys in $\Sigma$. (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 $\Theta$, with values from $D$ and predicates from $P$.

Algorithms. We implemented the following algorithms: (1) MapReduce algorithms on Hadoop 1.2.1: (a) EMVR of Section 4.1, (b) EMVR, which replaces EMVR of EMVR with VF2 [13] by enumerating all matches without early termination; (c) EMOpt of Section 5.1. (2) Vertex-centric algorithms on GraphLab [31]: (a) EMVC of Section 5.1, and (b) EMOpt, which optimizes EMVC 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.xlarge. 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, EMOpt (resp. EMOpt) 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 EMOpt 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.

![Figure 7: Keys defined on DBPedia](image-url)
These experimentally verify Theorems 6 and 10.

MapReduce vs. vertex-centric. Algorithm $EM_{VC}$ outperforms all the MapReduce algorithms, even $EM_{MR}^{Opt}$. 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 $EM_{MR}^{Opt}$ takes 70 seconds. This verifies that $EM_{VC}$ reduces the inherent costs of the I/O bound and the synchronization policy of MapReduce.

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

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

(2) Compared with $EM_{MR}^{Opt}$, $EM_{MR}$ 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 $EM_{VC}$, $EM_{VC}^{Opt}$ 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 $EM_{VC}^{Opt}$ and $EM_{MR}^{Opt}$ in the three datasets.

![Figure 8: Performance evaluation](image)

**Table 2: Candidate matches vs. confirmed matches**

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Candidate Matches</th>
<th>Confirmed Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$EM_{VC}^{Opt}$</td>
<td>$EM_{MR}^{Opt}$</td>
</tr>
<tr>
<td>Google</td>
<td>22015</td>
<td>15380</td>
</tr>
<tr>
<td></td>
<td>1620</td>
<td>1357</td>
</tr>
<tr>
<td>DBpedia</td>
<td>20000</td>
<td>11000</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

**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) $EM_{VC}^{Opt}$ performs the best among all of them, and $EM_{MR}^{Opt}$ outperforms the other MapReduce algorithms; these are consistent with the results of Exp-1; (3) for product graphs $G_p$ used by $EM_{VC}$ and $EM_{MR}^{Opt}$, $|G_p| = 2.7 |G|$ on average, which is much smaller than $|G|^2$; and (4) $EM_{MR}^{Opt}$ and $EM_{VC}^{Opt}$ 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 $\Sigma$**. Finally, we evaluated the impact of $\Sigma$, by varying the longest chain $c$ and maximum radius $d$ in $\Sigma$.

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) $EM_{VC}$ and $EM_{VC}^{Opt}$ 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 $EM^\text{Opt}_{MR}$ are 60%, 42%, 53% smaller than those of $EM^\text{Opt}_{MR}$, and it makes $EM^\text{Opt}_{VC}$ 4.8, 3.7 and 4.2 times faster than $EM^\text{Opt}_{MR}$ 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: $EM^\text{MR}$, $EM^\text{VC}$, $EM^\text{Opt}_{MR}$ and $EM^\text{Opt}_{VC}$ 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 $\Sigma$: on graphs with $G = (100M, 500M)\Sigma$ with 500 keys, $c = 2, d = 2, EM^\text{MR}$ and $EM^\text{Opt}$ take 27 and 1.5 seconds on average with 20 processors, respectively. (3) Our optimization techniques are effective: $EM^\text{Opt}_{MR}$ and $EM^\text{Opt}_{VC}$ are 3 and 1.5 times faster than $EM^\text{MR}$ and $EM^\text{VC}$ on average, and $EM^\text{Opt}_{MR}$ is 4.8 times faster than $EM^\text{Opt}_{VC}$.

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 11051000030834361, EPSRC EP/J015377/1 and EP/M025268/1, NSF III 1302212, and a Google Faculty Research Award.

8. REFERENCES