# KeyMiner: Discovering Keys for Graphs

Morteza Alipourlangouri
Dept. of Computing and Software
McMaster University
alipoum@mcmaster.ca

Fei Chiang
Dept. of Computing and Software
McMaster University
fchiang@mcmaster.ca

## ABSTRACT

Keys allow us to uniquely identify entities in a graph database. They have applications in object identification, entity resolution, knowledge fusion, and social network reconciliation. Keys also serve as a data quality constraint to not only identify duplicates, but to restrict updates to those satisfying the key graph patterns. In this paper, we present our work in mining keys over graphs. We propose an efficient algorithm that discovers keys in a graph via frequent subgraph expansion, and present two properties that define a meaningful key. Lastly, we discuss challenges for key discovery over dynamic graphs in spatio-temporal settings.

**PVLDB Reference Format:**
Morteza Alipourlangouri, Fei Chiang. Discovering Keys for Graphs. *PVLDB*, 11 (5): xxxx-yyyy, 2018.
DOI: https://doi.org/TBD

## 1. INTRODUCTION

Keys are a fundamental integrity constraint used in database systems to define the set of attributes that uniquely identify an entity. Keys serve a vital role in databases to maintain data quality standards by preventing incorrect insertions and updates as the data naturally evolves over time. In addition, keys provide clues for duplicate detection (also referred to as entity resolution), one of the most common data quality issues facing large organizations [6]. While keys are often defined by a domain analyst according to application and domain requirements, manual specification of keys is expensive and laborious for large-scale datasets. Existing techniques have explored mining for keys in relational data (as part of functional dependency discovery) [10], and in XML data [4].

The proliferation of graph databases has lead to the study of integrity constraints over graphs, including functional dependencies [9], and keys [8]. These constraints have shown to have wide applications to deduplication, citation of digital objects and knowledge fusion and knowledge base expansion [6]. Evolving graphs such as knowledge bases and citation graphs require keys to uniquely identify objects to ensure reliable and accurate deduplication and query answering. In these dynamic settings, where object properties change frequently and new objects are added, manual specification of keys is expensive and labor intensive. Automated solutions are needed to discover keys. Although recent work has proposed techniques to find keys over RDF data [2], these techniques are not applicable for graphs as they do not support: (i) topological constraints; and (ii) recursive keys (a distinct feature in graph keys). To the best of our knowledge, there is no existing work that discovers keys for directed graphs. Consider the following example where keys help to identify entities in an evolving database.


*Proceedings of the VLDB Endowment,* Vol. 11, No. 5

DOI: https://doi.org/TBD

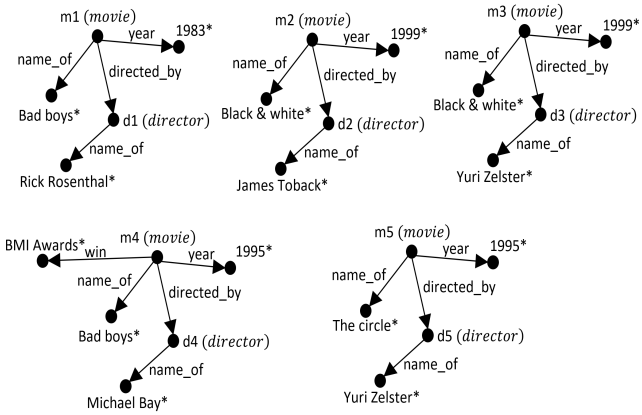


**Figure 1: Movie instances from IMDB database.**

EXAMPLE 1.1. *Figure 1 shows a sample of five movies from the IMDB movie graph database [1]. Intuitively, a key can be defined by a topological pattern, and a set of edge labels, and node types [8]. We can define a key for a type of entity using a graph pattern, and apply pattern matching algorithms to identify unique entities in a graph. Figure 2 shows a sample of possible keys to uniquely identify a movie:*

$Q_1(x)$ : By the movie name (title of the movie).
$Q_2(x)$ : By movie name, and year of release.
$Q_3(x)$ : By movie name, and a director.
$Q_4(x)$ : By movie name, and awards won.
$Q_5(x)$ : By movie name, and a specific director.
$Q_6(x)$ : A director can be uniquely identified by her name.

The example highlights that many keys are possible, and this often depends on the data and its semantics. According

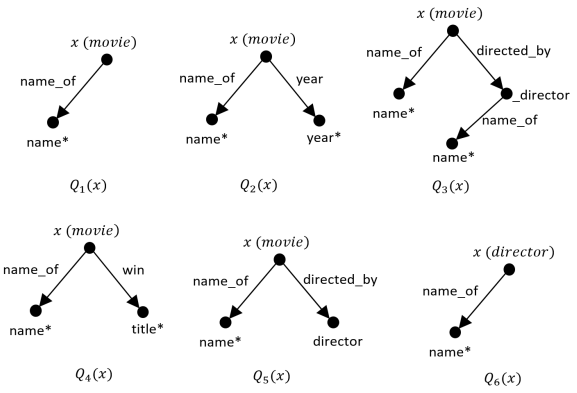


**Figure 2: Possible keys for *movie* and *director*.**

to $Q_1(x)$, all movies with the same name resolve to the same movie. This of course does not hold true over time, as Figure 1 shows two different movies titled 'Bad Boys' in 1983 ($m1$) and in 1995 ($m4$). Secondly, the domain semantics influence the quality of a key. For example, $Q_4(x)$ indicates a movie is uniquely identified by its name and awards won. However, not all movies win awards, and the second condition will lead to null values for many movies (that have not won awards), thereby leading to poor coverage and representation across all movies. To effectively identify keys in graphs, we consider the following questions:

1. What properties define a meaningful key in a graph?
2. How can we efficiently discover such keys?
3. How can these keys be effectively used in applications?

The example demonstrates the need for automatic discovery of keys over graphs, and to refine these keys over time as the data evolves. In our preliminary evaluation over knowledge bases, we found similar examples of postulated keys that became stale as new data is inserted. For example, a music database used musician name as an initial key. However, this was no longer sufficient after two musicians named 'Mick Jones' (from two different bands 'Foreigner' in 1976 and 'Big Audio Dynamite' in 1984) posed a conflict.

**Contributions.** In this paper, we address the above questions, and present *KeyMiner*, an algorithm that discovers keys in graphs. We propose two desirable properties for keys, and an algorithm for discovering keys in large-scale graphs. Lastly, we discuss open questions for further exploration, particularly for spatio-temporal graphs.

# 2. PRELIMINARIES

A *directed graph* is defined as $G(V, E, T, L)$, where $V$ is a finite set of vertices, $T$ is a finite set of types, and $L$ is a finite set of labels. A set of edges is denoted as $E \in V \times L \times V$, i.e., $e = (u, l, v)$ represents an edge from $u$ to $v$ with the label $l$ that is not equal to edge $(v, l, u)$. Each node $v \in V$ has a type $t$ from $T$ (nodes with no type have $t = \emptyset$), and each node $v$ has a numeric *id*, denoted by $v_{id}$. For example, in Figure 1, 'movie' and 'director' are node types that are common among all the instances, and each instance has a unique id for movies and directors, $m1 - m5$, and $d1 - d5$, respectively.

## 2.1 Graph Patterns

A *graph pattern* is a connected, directed graph $Q(x) = (V_Q, E_Q, T_Q, L_Q)$ where $V_Q$ is a finite set of entities; $E_Q$ is a finite set of edges; $T_Q$ is a function which assigns a specific type to each $v \in V_Q$; and $L_Q$ is a set of labels from which each edge $e \in E_Q$ has a label $l \in L_Q$. A graph pattern $Q(x)$ is matched in a graph $G$ if there is a subgraph isomorphism $i$ from $Q$ to $G$ where (i) for each node $v \in V_Q, v \leftrightarrow i(v)$; and (ii) for each edge $e \in E_Q, L_Q(e) = L_Q(i(e))$. For a graph pattern $Q(x)$, each node $v \in V_Q$ can be represented in one of three formats:

- **wildcard _*v*:** A node $v \in V_Q$ that is a wildcard (modeling an entity $x$) is matched to $v' \in V$ if their respective types are equal (no matching on node ids nor values is performed). e.g., `_director` in $Q_3(x)$.
- **variable *v*:** A variable node $v$ (modeling entity $x$), $v \in V_Q$ is matched to $v' \in V$ ($v \leftrightarrow v'$) if their respective node ids and types are equal, e.g., `director` in $Q_5(x)$.
- **value $v^*$:** For two value-based nodes, $v^* \in V_Q$ and $v' \in V$, $v^*$ matches $v'$ if their respective values are equal, e.g., `name*` in $Q_5(x)$ matches *Rick Rosenthal*.

For example, to find unique movies in Figure 1 using $Q_5(x)$, we check for equality of values using `name*`, and equality of node ids and types using `director`.

## 2.2 Keys for Graphs

A key for a graph is defined using a pattern $Q(x)$ for an entity $x$ [8]. If two subgraph isomorphisms $i_1$ and $i_2$ of $Q(x)$ are found in graph $G$, then $i_1(x)_{id} = i_2(x)_{id}$, if $\{\forall v \in V_Q, i_1(v) \leftrightarrow i_2(v) \wedge \forall e \in E_Q, L_Q(i_2(e)) = L_Q(i_2(e))\}$. This means the two instances refer to the same entity. For example, consider candidate key $Q_2(x)$ in Figure 2, that defines a unique movie based on its name and release year. By using $Q_2(x)$ to match the instances in Figure 1, we obtain the following classification of movies: $\{\{m1\}, \{m2, m3\}, \{m4\}, \{m5\}\}$. Since $m2$ and $m3$ share the same movie title `Black & White` and release year `1995`, they are considered duplicates, while movies $m1$, $m4$ and $m5$ are distinct.

Keys in graphs may contain entities as part of the original key definition, leading to *recursive* definitions. For example, Figure 2 shows that $Q_5(x)$ defines a key for a movie based on the entity *director*. To correctly interpret $Q_5(x)$, we must first define a key for the *director* entity (using the director's name), as shown in $Q_6(x)$. By applying $Q_6(x)$ to Figure 1, we obtain the following classification for directors $\{\{d1\}, \{d2\}, \{d3, d5\}, \{d4\}\}$, revealing two references to the same director `Yuri Zelster`. We can then apply $Q_5(x)$ as a candidate key to uniquely identify movies in Figure 1, revealing that all movies $m1$ - $m5$ are unique.

Since keys for graphs are defined using graph patterns, graph pattern mining and matching algorithms form a basis for key discovery algorithms. In our work, we use the well-known pattern mining algorithm, GRAMI [7] that has shown to be efficient in non-distributed settings. We are exploring the use of distributed graph mining techniques such as Arabesque [11] that are built on top of Hadoop and Spark, to increase scalability to extremely large graphs.

# 3. KeyMiner: DISCOVERING KEYS

Discovering meaningful keys in graphs relies on defining key properties independent of the application domain and the data, especially when multiple candidate keys are possible to identify an entity. We propose two key properties: *minimality* and *coverage* (along with preliminary ideas of

how to quantify these properties), and a key discovery algorithm over graphs.

## 3.1 Property 1: Minimality

In traditional relational databases, a key is defined as the set of minimal attributes that uniquely identifies a tuple. More attributes can be added to a key and still serve as a key, but these additional attributes are not necessary. We extend this intuition and define *minimal* keys in graphs as those with a simple graph pattern. Specifically, we seek keys that (i) contain a minimal number of nodes; and (ii) a minimal number of entity *variables* (as defined in Section 2.1). Key patterns containing variables elicit recursive keys, which require longer matching times as pattern matching must be performed against a graph $G$ for each sub-entity. The number of *wildcards* and *values* can be captured by the number of nodes since each of these formats does not evoke recursive keys. We propose Equation (1) as an initial step to quantify the minimality of a key.

$$min(Q(x)) = 1 - \frac{(\phi \times E_1) + ((1-\phi) \times E_2)}{m} \quad (1)$$

Let $E_1$ represent the sum of *value* and *wildcard* nodes, and $E_2$ be the number of *variable* nodes in the graph pattern $Q(x)$. The non-zero parameter $\phi$ is a user-defined weight denoting the preference towards simpler (wildcard and value based) nodes. Let $m$ be a user-defined value representing an upper bound on the (maximum) number of nodes. The *min* values range from (0,1), where larger values indicate more minimal keys.

## 3.2 Property 2: Coverage

For a candidate key $Q$, we define *coverage* to represent the number of entities in a graph $G$ captured by $Q$. We seek keys that model a maximum number of instances in $G$. Consider our earlier example with candidate key $Q_4(x)$ that uniquely defines movies based on its title and awards. Movies that have not received awards are not covered by $Q_4(x)$. Since each key $Q$ elicits a classification of the instances, where instances within a class are considered duplicates, we search for keys that partition the instances into a maximal number of classes. For example, consider $Q_3(x)$ and $Q_4(x)$, which classify the entity instances into the following classes: $\Pi_{Q_3(x)} = \{\{m1\}, \{m2\}, \{m3\}, \{m4\}, \{m5\}\}$ and $\Pi_{Q4(x)} = \{\{m4\}\}$. We prefer candidate key $Q_3(x)$ as a maximum number of classes is achieved. We define *cov(Q(x))* that measures the coverage for a key $Q(x)$.

$$cov(Q(x)) = \frac{I \times C}{N^2} \quad (2)$$

Let $I$ denote the number of instances captured by $Q(x)$, $C$ is the number of classes after using $Q(x)$ to partition the $I$ instances, and $N$ is the total number of instances for entity $x$ in graph $G$. The values in *cov(Q(x))* range between [0, 1], where a value of 0 indicates no instances are captured by $Q(x)$, and 1 indicates all instances are uniquely identified. We revisit our candidate keys $Q_3(x)$ and $Q_4(x)$, where $cov(Q_3(x)) = 1$, and $cov(Q_4(x)) = 0.04$, indicating $Q_3(x)$ has better coverage than $Q_4(x)$. Our coverage measure finds similarity to existing support-based metrics used in association rule mining over dynamic graphs where support is defined as the coverage w.r.t. a time window.

To rank the set of discovered keys, a ranking function, $rank(Q(x))$ is needed to preferentially select minimal keys with maximum coverage. We intend to explore various ranking functions as our next steps. This includes functions that consider the spatial and temporal locality of data values, i.e., in dynamic settings, some key values have greater longevity and are more likely to hold true for longer periods of time.

## 3.3 KeyMiner Algorithm

We present, KeyMiner, an algorithm for discovering keys in a graph database $G$. KeyMiner mines for keys for each type $t \in T$, by creating a search tree rooted at node $x$ of type $t$, and then expands and traverses the tree to define a candidate key. Candidate keys are evaluated based on a ranking function (`rank`) that evaluates the quality of the key based on the minimality and coverage properties. Algorithm 1 presents KeyMiner, which takes input graph $G$ and types $T$, and returns *keys* for each type $t \in T$. The mining step is done via the `Discovery` algorithm, given in Algorithm 2.

**Algorithm 1** *KeyMiner*

**Input:** A graph $G$ and set $T$ of types
**Output:** Candidate keys for all types

```
1: keys = CreateSet(T);
2: for all type t in T do
3:     if keys[t] == null then
4:         keys[t]=Discovery(G, t);
5: return keys
```

To mine for keys for an entity $e$ of type $t$, the *Discovery* Algorithm first creates a graph pattern consisting of a single node $x$ of type $t$. This pattern is expanded upon by computing the set $G_t^d$, containing all $d$-neighbors within a user-defined radius $d$ of $x$ (line 2), where $G_t^d = \bigcup_{e \in G} G_e^d$, and $G_e^d$ is the $d$-neighbor graph of entity $e$. We define candidate keys by iteratively expanding the pattern with a neighbor from $G_t^d$, and consider adding *values*, *wildcard* and entity *variables* one at a time (line 4). We then check whether the new candidate key is recursive (line 8-12), and use a directed acyclic graph (DAG) to track dependencies between entities. If a newly added node contains an entity variable of type $t'$ ($t' \neq t$), we add these two nodes $t$ and $t'$ into the DAG with a connecting edge to denote their dependency (line 9). Cycles in the DAG are broken by removing the last added dependency ($t \rightarrow t'$). If there are no cycles, we recursively call *Discovery* on the new node of type $t'$. Candidate keys are selected based on satisfying a user-defined rank threshold (lines 15-17). We implement an optimization that prunes candidate keys that do not satisfy a minimum number of instances (line 18).

Example 3.1. *Figure 3 shows a sample search tree for movies. Each level of the tree is generated by adding a node to the sub-tree from the previous level. In the first level, we add all possible node types to the movie entity, and obtain five candidate keys. For an entity such as* director*, we generate keys that can match any graph using either variables (key k4) or wildcards (k3). Recall in the wildcard case, the existence of an entity (director) is a sufficient condition for a match. During the search, a candidate key k is returned if $rank(k) >$ min_rank threshold and no further expansions of*

**Algorithm 2** *Discovery*

**Input:** A graph $G$ and type $t$
**Output:** Candidate key for entities of type t

```
1: p = CreatePattern(x, t);
2: G_t^d = ComputeAllNeighbors(G, t);
3: while keys[t] ≠ null do
4:    candidates = Expand(p, G_t^d); //expand p with one
      neighbor from G_t^d
5:    if candidates == null then
6:       return nokey;
7:    for all pattern p' in candidates do
8:       if IsRecursive(p') then
9:          Insert(DAG, t, t'); // p' contains t' and (t' ≠ t &
            keys[t'] == null)
10:         if IsLoop(DAG) then
11:            ResolveLoop(DAG);
12:            continue;
13:         else
14:            keys[t']=Discovery(G, t');
15:      if Rank(p')≥ min_rank then
16:         Delete(DAG, t);
17:         return p' ;
18:      if Instance(p')≥ min_instance then
19:         Prune(p');
20: return keys
```

*the key is done. If a candidate key does not satisfy the minimum number of instances (min_instance), we prune all expansions of this key (k5). Candidate key k4 contains a recursive reference to sub-entity 'director', and we add the 'movie' to 'director' reference to the DAG, and recursively mine for* director *keys. Once a key is found for the sub-entity* director *(e.g., $Q_6(x)$) we resume key mining for* movies. *Keys such as k8 are preferred for its minimality and maximal coverage.*

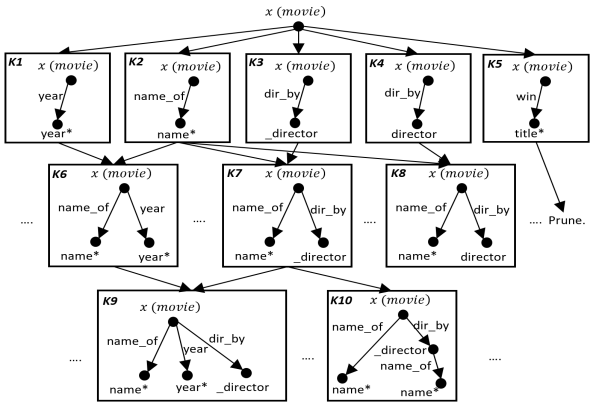


**Figure 3: Search tree for the *movie* entity.**

# 4. CHALLENGES & NEXT STEPS

Discovering keys over dynamic graphs poses additional challenges that require modeling the frequency and type of changes (topological vs. labels), and identifying spatial and temporal correlations among these updates. We outline challenges in each of these areas as our next steps.

**Key mining over dynamic graphs.** Dynamic graphs require us to develop a change model that tracks information such as the type of change (topological, labels, values), the frequency of change, and the longevity of values in the graph (i.e, some values may persist for longer periods of time). This data serves as input into an extension of KeyMiner that considers *incremental repairs* to already discovered keys, similar to constraint repairs over relational data [5]. During candidate key selection, the `rank` function can be updated to prefer keys containing persistent graph patterns to minimize the need for future repairs, or to identify keys with sufficiently high coverage.

KeyMiner can also be used to discover keys that hold under different interpretations. Similar to ontology FDs that interpret FDs w.r.t. the concepts and relationships in an ontology [3], a key may hold only under a given context. Knowledge bases may contain entities that are defined w.r.t. a geographic or domain-specific interpretation. For example, the data properties to uniquely identify a prescription drug are often country specific according to federal drug administration rules, how the drug is used to treat illnesses, and regional conventions used to market prescription drugs.

**Temporal keys:** Changes to the graph may exhibit spatial correlations where specific nodes are frequently updated together, e.g., movie name and year. Such updates often follow a *temporal locality* where values that were recently updated are likely to be updated again the near future. We explore *temporal key constraints*: graph keys that hold over a specific window of time. These temporal constraints are defined as part of the schema, and serve as a tool for understanding the evolution of keys and the graph.

# 5. REFERENCES


[1] *IMDB Movie Graph Database, https://datasets.imdbws.com*, 2018 (accessed Apr. 20).

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

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

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

[5] F. Chiang and R. Miller. Active repair of data quality rules. In *Intl. Conf. on Information Quality (ICIQ)*, pages 174–188, 2011.

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

[7] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a large graph. *VLDB*, pages 517–528, 2014.

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

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

[10] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[11] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440, 2015.