TAPER: query-aware, partition-enhancement for large, heterogeneous graphs

Hugo Firth¹ · Paolo Missier¹

Published online: 2 May 2017

© The Author(s) 2017. This article is an open access publication

Abstract Graph partitioning has long been seen as a viable approach to addressing Graph DBMS scalability. A partitioning, however, may introduce extra query processing latency unless it is sensitive to a specific query workload, and optimised to minimise *inter*-partition traversals for that workload. Additionally, it should also be possible to incrementally adjust the partitioning in reaction to changes in the graph topology, the query workload, or both. Because of their complexity, current partitioning algorithms fall short of one or both of these requirements, as they are designed for offline use and as one-off operations. The *TAPER* system aims to address both requirements, whilst leveraging existing partitioning algorithms. *TAPER* takes any given initial partitioning as a starting point, and iteratively adjusts it by swapping chosen vertices across partitions, heuristically reducing the probability of inter-partition traversals for a given path queries workload. Iterations are inexpensive thanks to time and space optimisations in the underlying support data structures. We evaluate *TAPER* on two different large test graphs and over realistic query workloads. Our results indicate that, given a hash-based partitioning, *TAPER* reduces the number of inter-partition traversals by $\sim 80\%$; given an unweighted Metis partitioning, by $\sim 30\%$. These reductions are achieved within eight iterations and with the additional advantage of being workload-aware and usable online.

Keywords Graph databases · Graph repartitioning · Workload mining

✉ Hugo Firth
h.firth@ncl.ac.uk; me@hugofirth.com

Paolo Missier
paolo.missier@ncl.ac.uk

¹ School of Computing Science, Newcastle University, Newcastle upon Tyne, UK

1 Introduction

Path queries over labelled graphs are increasingly common in many applications. These include fraud detection [27], recommender systems [9] and social analysis [2] amongst others. Such a labelled graph has the form $G = (V, E, L_V, l)$, where each vertex v is annotated with a label $l(v) \in L_V$ from a predefined set L_V of labels (e.g. Purchase, Person, etc...). In this work we address the problem of efficiently and incrementally **improving path query performance over k -way partitionings** of large, heterogeneous, labelled graphs. A k -partitioning $P_k(G)$ of G is a family of disjoint sets $\{V_1, V_2, \dots, V_k\}$, with $V_1 \cup \dots \cup V_k = V$. The heterogeneity of G refers to the diversity in the labels L_V associated to the vertices, e.g., a social graph with $L_V = \{Person, Post\}$ is more heterogeneous than a web graph with $L_V = \{Url\}$.

Partitioning large graphs is a recognised approach to addressing scalability issues in graph data management. However, if these partitionings are of a low quality then the performance of path queries (indeed, general pattern matching queries), greatly decreases [18]. Intuitively, any measure of this partitioning quality should correspond to the number of inter-partition traversals, or *ipt* for short, i.e., the number of times that inter-partition edges $(v_i, v_j) \in E$ with $v_i \in V_i, v_j \in V_j, i \neq j$ are traversed during query execution. Current systems for improving graph partition quality either optimise data placement (graph partitioners) [8, 12, 23, 26], or are based on selective vertex replication [18, 21, 32]. We will improve on the output of graph partitioners, without considering vertex replication, i.e. $V_i \cap V_j = \emptyset$ for $i \neq j$.

Existing graph partitioners have two main drawbacks: Firstly, due to their computational complexity [26], non-streaming methods [8, 12, 23] are only suitable as offline operations, typically performed ahead of analytical workloads. For online, non-analytical workloads, they require complete re-execution, i.e. after a series of graph updates, which may be impractical [10]. Simpler methods, such as grouping vertices by some hash of their ids, are efficient [18] but yield poor *ipt* scores when queried. Methods meant to partition graph streams [26, 28] lie between these two extremes, both in terms of efficiency and quality. However, they make strong assumptions about the order of graph streams and the availability of neighbourhood information for new vertices. As a result they are also largely confined to offline application.

Secondly, the partitioners are agnostic to query workloads as they optimise for producing the minimum number or weight of inter-partition edges (*min edge-cut*). This is equivalent to assuming uniform, or at least constant, likelihood of traversal for each edge throughout query processing. This assumption is unrealistic as a workload may traverse a limited subset of edges, which is specific to its query patterns and subject to change. To appreciate the importance of query-sensitive partitioning, consider the graph of Fig. 1. The partitioning $\{A, B\}$ is optimal following a balanced min edge-cut approach [12], but it may not be optimal when query patterns are taken into account.

Following common practice, we express queries using a Regular Path Queries [1, 17] (RPQ) formalism, which can be expressed using a restricted form of regular path expressions over the set of vertex labels. For example, expression $c \cdot (b|d)$ evaluates to paths $(3, 2)$, $(3, 4)$, $(5, 2)$, $(5, 4)$ over the graph in Fig. 1.

Notice that computing each of these paths requires 1 *ipt*. However, it is easy to see that with the alternative partitioning $V_1 = \{1, 3, 6\}$, $V_2 = \{2, 4, 5\}$, only paths

Fig. 1 Illustrative example graph

(3, 2), (5, 4) require traversing a partition boundary, although this partitioning is not optimal with respect to min edge-cut. Mature research of query-sensitive database partitioning is currently confined to relational DBMS [3,22].

1.1 The *TAPER* re-partitioner

In this paper we present *TAPER*, a graph re-partitioning system that is sensitive to evolving query workloads. Let $\mathcal{Q} = \{(Q_1, n_1) \dots (Q_h, n_h)\}$ denote a query workload, where n_i is the relative frequency of Q_i in $\mathcal{Q}$, and let $P_k(G)$ be an existing k -way partitioning of G . This could be for instance a simple hash-based partitioning, or one based on an established method such as Metis [12] multilevel partitioning [12] or spectral recursive octasection [8].

The goal of *TAPER* is to enhance $P_k(G)$, by computing a new partitioning $P'_k(G, \mathcal{Q})$ from $P_k(G)$ that takes $\mathcal{Q}$ into account. The new partitioning is obtained by swapping vertices across the partitions of $P_k(G)$, using heuristics that attempt to minimise the total *probability of ipt*, denoted *total extroversion*, that occur during execution of any of the queries in $\mathcal{Q}$. As this method only involves moving relatively few vertices from one partition to another, it is much less expensive than a complete re-partitioning, even after many iterations.

Furthermore, by virtue of its incremental nature, *TAPER* is able to track changes $\mathcal{Q} \rightarrow \mathcal{Q}'$ in the workload, by re-partitioning its own partitioning, i.e.,

$$P_k(G, \mathcal{Q}) \xrightarrow{\mathcal{Q}'} P'_k(G, \mathcal{Q}') \quad (1)$$

In general, given an initial, possibly workload-agnostic, and non-optimal initial partitioning $P_k^0(G)$, *TAPER* can be used to compute a progression of partitionings:

$$P_k^0(G) \xrightarrow{\mathcal{Q}_1} P_k^1(G, \mathcal{Q}_1) \xrightarrow{\mathcal{Q}_2} P_k^2(G, \mathcal{Q}_2) \dots \quad (2)$$

These partitionings have the property that each $P_k^i(G, \mathcal{Q}_i)$ exhibits lower extroversion than $P_k^0(G)$ given $\mathcal{Q}_i$: it is approximately optimised for that workload.

TAPER makes use of space-efficient main-memory data structures to encode $\mathcal{Q}$ and to associate estimates of traversal probability with the edges in G . These are then used

to calculate the extroversion of each vertex in its partition. A *TAPER* re-partitioning step, as in Definition 1, is actually several internal iterations of a vertex-swapping procedure aimed at reducing extroversion one vertex at a time. Each time a new *TAPER* invocation is required (1) we update our data structures for $\mathcal{Q}'$ and G' , recompute vertex extroversion and begin a new round of iterations.

1.2 Contributions

Our specific contributions are as follows: **Firstly**, from the notion of *stability* of a partition [4] we derive an operational metric of partitioning quality, expressed in terms of extroversion for each vertex; **Secondly**, we describe an encoding of traversal probabilities for each edge in G , given $\mathcal{Q}$, which is space-efficient and show how they can be updated following the evolution of $\mathcal{Q}$; **Thirdly**, we show how *TAPER* makes use of these structures to iteratively achieve a re-partitioning step (Definition 1).

We present an extensive evaluation of the *TAPER* system using both real and synthetic graphs of varying sizes, and compare its performance and scalability against one-off workload-agnostic partitionings obtained using the popular Metis approach,¹ without edge weights. In our experiments we use both a simple hash-based partitioning as well as a Metis partitioning as a starting point $P_k^0(G)$ for one invocation of *TAPER*. Our results show that such an invocation converges to a stable quality within 6–7 internal iterations, and that the resulting new partitioning $P_k^{i+1}(G, \mathcal{Q})$ exhibits 70% quality improvement when a hash-based $P_k^0(G)$ starting point is used, and about 30% improvement when using a Metis initial partitioning.

Finally, we show experimentally how the quality of a partitioning degrades following successive simulated changes in $\mathcal{Q}$, and how it is successfully restored by repeatedly invoking *TAPER* on the current partitioning and the new workload.

1.3 Related work

Two main strands of prior work are relevant to our study: (1) workload aware replication and data placement in distributed databases; and (2) graph partitioning.

In the context of online applications, distributed database queries that traverse multiple partitions are expensive [20], incurring high communication cost and, in some implementations, resource contention. In order to achieve good latencies, distributed databases must find a data placement strategy which minimises these transactions, and the overhead they cause, whilst maintaining a balanced load across all machines involved. This is also known as reducing average query *span*, i.e. the number of partitions involved in answering a query.

For graph data, *balanced graph partitioning* has been exhaustively studied in literature since the 1970s [8, 12, 13, 16, 23, 25, 26, 28, 30], and a number of practical implementations are available [12, 23]. We do not seek a new graph partitioning algorithm; rather, to propose a workload-driven method for improving partitions that already exist.

¹ Metis: <http://bit.ly/1tqUcSQ>.

This goal is orthogonal to that of the Loom [7] system, also proposed by the present authors. Loom uses a variant of the traversal probability encoding presented here (Sect. 4) to produce a workload-aware partitioning of a graph *stream*. This may seem to support our goal with *TAPER*, however, like other streaming graph partitioners [26, 28], once Loom has assigned a vertex to a partition it will never reassign it. This has several implications: firstly, Loom assumes that a workload Q is known *a priori* with respect to vertex assignment; secondly, Loom's partitionings are vulnerable to aforementioned degrading quality given changes to Q over time; and thirdly, Loom may only partition a graph fully, rather than take an existing partitioning as input. This is in contrast with *TAPER*, which tracks an ongoing workload $Q \rightarrow Q'$ and may continuously adapt an existing graph partitioning for good performance given a random $Q_i \in Q'$.

Note that *TAPER*'s goal of refining an existing partitioning is supported by ParMetis [13], a parallel implementation of the established Metis partitioner [12]. However, this process is communication and computation intensive and as such is often only used as a “one-off” step, rather than for repeated repartitioning of a graph [29]. Additionally, like other existing partitioners, ParMetis is agnostic to a changing query workload. Although ParMetis may be provided with weights which correspond to the traversals of individual edges by a query workload, such weights are much too expensive to maintain over time [32].

Aside from Loom and ParMetis, Curino et al. [3, 20] have proposed systems to tackle a problem related to our own: that of workload-aware data placement in distributed RDBMS. In particular, *Schism* [3] captures a query workload over a period of time, modelling it as a graph, each edge of which represents tuples involved in the same transaction. This graph is then partitioned using existing “one-off” techniques to achieve a min edge-cut. Mapped back to the original database, this partitioning represents an arrangement of records which causes a minimal number of transactions in the captured workload to be distributed. In *SWORD*, Quamar et al. [22] build upon the ideas presented in *Schism* [3]. They use a compressed representation of the workload graph and perform incremental re-partitioning to improve the partitioning's scalability and sensitivity to workload changes.

Although the goal of these works and our own is similar, there exist major differences in approach. For instance, in *Schism*, edges directly represent the elements accessed by a query, rather than their labels as we do. However this fine grained approach produces very large graphs, which are expensive to both partition and store, and may impact scalability [22]. Furthermore, these works are focused on a relational data model, where typical workloads overwhelmingly consist of short, 1–2 “hop” queries. This justifies Quamar et al.'s simplifying decision, when repartitioning a graph, to consider only queries which span a single partition. However, this assumption does not hold for general graph path and pattern matching queries. It is unclear how *SWORD*'s approach would perform given a workload containing many successive join operations, equivalent to the traversals required for graph pattern matching.

Other systems, such as LogGP [30] and CatchW [25], are explicitly workload-aware graph partitioners, but focus on improving analytical workloads designed for the *bulk synchronous parallel* (BSP) model of computation. In the BSP model a graph processing job is performed in a number of supersteps, synchronised between partitions. Both systems keep a historical log of edges traversed in previous supersteps and

use this to predict future traversals and move involved vertices so that their edges are internal to partitions, reducing *ipt* in subsequent supersteps.

Yet more partitioning systems [29,31] consider other aspects of a graph's workload, besides edge traversals. Xu et al. [31] note that, for many BSP workloads, reducing the total communication time of a workload by simply minimising a graph's inter-partition edges may not be an appropriate optimisation. This is due to the fact that a cluster of machines may be heterogeneous, i.e. machines may have different levels of computing power, or be virtualised across various physical hosts, affording different levels of inter-machine communication. Xu et al. therefore propose modelling the topology of a cluster upon which a graph partitioning will reside, as a graph itself. Subsequently, given some basic information about a workload's communication and computation characteristics, vertices and edges of the data graph are assigned to appropriate machines, minimising workload execution time.

The work of Vaquero et al. [29] is perhaps the most relevant to our own with TAPER. They propose a system of iterative vertex swapping to adapt to graph changes over time (e.g. vertex additions and removals). This is highly effective at maintaining a good partitioning over time, *w.r.t* min edge-cut. However, *because* the system optimises for min edge-cut, it does not consider the heterogeneity of a graph, or that of its query workload. Thus the work we present here could complement that of Vaquero et al: they consider changes to the graph whilst we consider changes to its workload.

Not only are all these systems [3,22,25,29–31] focused on different types of workload than we are with TAPER, but in an online setting, workload summaries which directly measure individual edge traversals [3,25,30] are vulnerable to overfitting. Path queries in particular are often only executed over a subset of the graph, i.e. find *Post* connected to *Person a*. If this subset changes (e.g. to *Person j*) then such a granular workload summary would be inaccurate. By considering instead the vertex labels in a workload's queries and the graph structure local to a given edge *e*, TAPER is able to predict *e*'s traversal probability even when it has not yet been traversed.

Further prior work has focused on exploiting statistical properties of a query workload, to efficiently manage graph data through replication [18,21,32]. Whilst [18,21] focus on 1-hop queries over social networks, Yang et al. [32] propose algorithms to efficiently analyse general online query workloads and to dynamically replicate “hotspots” (cross partition clusters of vertices that are being frequently traversed), thereby temporarily dissipating network load. Whilst highly effective at dealing with unbalanced query workloads, Yang et al. focus solely upon the replication of vertices and edges using temporary *secondary* partitions. They do not improve upon the initial partitioning, nor do they consider workload characteristics when producing it. This can result in replication mechanisms doing far more work than is necessary over time, adversely affecting the performance of a system. As a result, the enhancement techniques we present here would also complement many workload aware replication approaches, such as that proposed by Yang et al.

2 Definitions

In a labelled graph $G = (V, E, L_V, l)$, function $l : V \rightarrow L_V$ associates a label $l(v)$ from a given set L_V to each vertex $v \in V$. A **path-query** q over G is a regular expression over symbols in L_V . We use a type of Regular Path Queries (RPQ) [17], defined by the following expression language over L_V :

$$E ::= \tau \mid (E \cdot E) \mid (E + E) \mid (E \mid E) \mid E^* \quad (3)$$

where $\tau \in L_V$, and as usual “+” represents union, “|” exclusive disjunction, and “*” the Kleene closure operator.

Let $L(Q)$ denote the regular language defined by a query Q . The result of executing Q is a set of subgraphs $G_i = (V_i, E_i, L_V, l)$, where $V_i = \{v_{i_1} \dots v_{i_n}\} \subset V$ consists of all and only the vertices such that $l(v_{i_1}) \dots l(v_{i_n})$ is a valid expression in $L(Q)$. $E_i \subset E$ is the set of edges $e \in E$ that connect the vertices v_{i_j} in G .

Note that queries that include more complex topologies, such as branching and cycles, typically require conjunctions between expressions, or other extensions to RPQs, such as those proposed by Barcelo et al. [1]. For simplicity, these extensions are not covered by the RPQ fragment defined by expression language (3), and are not within the scope of this work.

2.1 Stability of a graph partitioning

The broad goal of *TAPER* is to increase the quality of a k -way partitioning (Sect. 1.1). Here we define the measure of partition *quality* which we aim to increase. For this, we extend the notion of partition *stability*, first introduced by Delvenne et al. [4] in the context of multi-resolution community detection in graphs; stability is described in terms of network flow. The main intuition is that, when a partition is *stable*, a flow that originates from a point within a partition and moves randomly along paths should be trapped within the same partition for a long time t . Network flow in graphs is modelled as a random walk, where discrete time t is measured as the number of steps. More precisely, the stability of a partition V_i is defined as the probability that it contains the same random walker both at time t_0 and at time $t_0 + t$, less the probability for an independent walker to be in $V_i : p(V_i, t_0, t_0 + t) - p(V_i, t_0, \infty)$. Note that this definition allows for the possibility of a walker crossing multiple partition boundaries before returning to its initial partition at any time during the $[t_0, t_0 + t]$ interval. The overall stability of a partitioning $P_k(G)$ is the sum of the stability of all partitions V_i where $1 \leq i \leq k$. In other words, the greater the stability of a partitioning, the higher the probability that a random walker, having traversed t steps, will be in the same partition where it started.

2.2 Workload-aware stability

We extend stability by creating a new measure of partition quality which we will refer to as *workload-aware stability*. Our extensions are driven by two main requirements. Firstly, *TAPER* aims to improve the *quality* of a graph partitioning by minimising

the probability of expensive *inter*-partition traversals, when executing a **given query workload** $\mathcal{Q}$ (Definition 1 in Sect. 1.1). Using stability, which models network flow as random walkers that traverse paths in a graph, gives us more flexibility than other measures of partition quality, such as *edge-cut*, when we try to incorporate information on a query workload. Stability's 'walkers', represented by the probabilities in a transition matrix, may be modified to account for the specific graph patterns associated with the queries in $\mathcal{Q}$, along with their relative frequency. This will reveal different dominant traversal patterns and produce a measure of quality more closely correlated with the cost of executing $\mathcal{Q}$ over a particular graph partitioning. Secondly, the current definition of stability as given above is limited, as it does not account for the probability that a walker crosses partition boundaries multiple times within t steps. In contrast, we need to be able to estimate the probability that the walker **does not leave the partition** within the interval.

2.3 The visitor matrix: non-random walks with memory

We address both requirements by extending the well-known notion of a *biased* random walk over a graph. Rather than uniform transition probabilities, such a random walk assumes the more general Markov property; that is, the probability of a transition from vertex v_k to v_j only depends on the prior probability of being in v_k :

$$Pr(v_k \rightarrow v_j | v_i \rightarrow \dots \rightarrow v_k) = Pr(v_k \rightarrow v_j | v_k)$$

In this case, the probabilities $Pr(v_k \rightarrow v_j | v_k)$ are captured by a transition matrix M :

$$M[k, j] = Pr(v_k \rightarrow v_j | v_k)$$

and the probability of a t -steps walk from v_k to v_j is computed as $M^t[k, j]$. However, taking into account the query matching patterns, as per our requirements above, invalidates the Markov property, because the probability of a transition $v_k \rightarrow v_j$ now depends on the specific path through which we arrive at v_k :

$$Pr(v_k \rightarrow v_j | p \rightarrow v_k) \neq Pr(v_k \rightarrow v_j | p' \rightarrow v_k)$$

in general, for any two paths $p \neq p'$ leading to v_k .

In other words, in order to account for query matching patterns of length up to t , where t is defined by the query expressions in $\mathcal{Q}$, we use a *multi-step* (non-random) walk model over the graph, which has memory of the last t steps. Each transition probability $v_k \rightarrow v_j$ is now explicitly conditioned on the paths, of length up to t , which lead to v_k .

To represent these probabilities, we extend M to a set of matrices:

$$VM(t) \equiv \{VM^{(1)}, \dots, VM^{(t)}\} \quad (4)$$

where t denotes the longest query matching pattern in $\mathcal{Q}$, and $VM^{(k)}$ has dimension $1 \leq k \leq t$. We use the term **Visitor Matrix** (VM) to refer to (4).

Fig. 2 Visitor Matrix structure

The definition of VM is by induction, where the base cases are the prior probabilities $Pr(v_i)$ to be in v_i , for $VM^{(1)}$, and the normal transition matrix M , for $VM^{(2)}$. Formally:

$$\begin{aligned}
 VM^{(1)}[i] &= Pr(v_i) \\
 VM^{(2)}[i_1, i_2] &= Pr(v_{i_1} \rightarrow v_{i_2} | v_{i_1}) = M[i_1, i_2] \\
 VM^{(k)}[i_1, \dots, i_k] &= Pr(v_{i_{k-1}} \rightarrow v_{i_k} | v_{i_1} \rightarrow \dots \rightarrow v_{i_{k-1}})
 \end{aligned} \quad (5)$$

for $2 < k \leq t$. Figure 2 shows a representation of a Visitor Matrix with $t = 3$, using a 2-dimensional matrix layout where $VM^{(3)}$ is “appended” to $VM^{(2)}$. The cells in the matrix store probabilities for paths in the example graph to the right (originally Fig. 1), relative to query expression Q_1 . For example, path $1 \rightarrow 2 \rightarrow 3$ is an instance of query pattern abc , and its probability is stored in $VM^{(3)}[1, 2, 3]$ (similarly for the other highlighted elements in the matrix). A VM , like any finite transition matrix, is right-stochastic, i.e., each row sums to 1, and the cells represent all paths up to length t . We show how compute the elements of $VM(t)$ for a given query workload $\mathcal{Q}$ in Sect. 4.2.

In practice, we assume $VM(t)$ is partitioned into n sub-matrices $VM_i(t)$, one for each of n partitions, because we can always find a permutation of the rows and columns of VM such that $VM_i(t)$ is a contiguous sub-matrix of VM . Thus, in the following we use $VM_i(t)$ to refer the VM for partition V_i .

Note that the visitor matrix is impractically large to compute, with a space complexity of $O(|V|^t)$. In Sect. 5.2 we present heuristics that are designed to reduce both space complexity, as well as to avoid computing some of the cells in the VM .

3 Enhancing a partitioning

We exploit the VM structure, to compute a new partitioning $P(G, \mathcal{Q})$ from a partitioning $P(G)$, as in Eq. 1. First we identify a set of vertices in each partition with highest

likelihood of being the source of *inter*-partition traversals (extroversion). Subsequently we swap such high-extroversion vertices between partitions, internalising the common traversal paths resulting from Q in single partitions. As we will show experimentally, repeated iterations of these steps reduce the overall likelihood of inter-partition traversal across all partitions V_i , and thus, indirectly, increase *workload-aware stability*.² These iterations constitute one invocation of the *TAPER* algorithm; not to be confused with repeated invocations given a changing workload (as described in Eq. 2).

3.1 Increasing stability by vertex swapping

Informally, we define the *extroversion* of a vertex v to be the likelihood that it is the source of an *inter*-partition traversal, given any of the query patterns in Q . *TAPER* seeks to enhance a partitioning by determining a series of vertex swaps between graph partitions such that their total extroversion is minimised. This is an extension of the general graph partitioning problem, a classic approach to which is the algorithm KL/FM, proposed by Kernighan and Lin [14] and later improved upon by Fiduccia and Mattheyses [5]. They present techniques that attempt to find sets of vertices and edges which, when moved between two halves of a graph bisection, produce an arrangement that is globally optimal for some criteria (usually min edge-cut). Karypis and Kumar [12] subsequently generalise this technique to address the problem of k -way partitioning, in an algorithm that they call *Greedy Refinement*. Greedy Refinement selects a random boundary vertex³ and orders the partitions to which it is adjacent by the potential *gain* (reduction in edge-cut) of moving the vertex there, subject to some partition balance constraints. If a move does not satisfy chosen balance constraints, progressively less beneficial destination partitions are considered. Finally, the move will be performed. Greedy Refinement has been shown to converge within 4–8 iterations. It is this algorithm which we use as the basis for our *TAPER*'s vertex swapping procedure. However, rather than reduction in edge-cut, we use the reduction in extroversion as our measure of gain for evaluating vertex swaps.

There are some other key differences between our own approach, and that of Greedy Refinement. Firstly, Greedy Refinement considers vertices at random from the boundary set, whilst we consider only the set of most *extroverted* vertices, in descending order of extroversion. This reduces the number of swaps performed and so should improve performance. Secondly, Greedy Refinement is designed to operate on a graph compressed using a matching algorithm, so every vertex move corresponds to the movement of a cluster of vertices in the original graph. Without this trait, Greedy Refinement would be more susceptible to being trapped in local optimisation minima: as vertex clusters are iteratively moved across partition boundaries edge-cut may temporarily increase. We do not operate on a compressed graph; instead we opt for a simple flood fill approach, detailed in Sect. 5.5. Using traversal probabilities, precomputed in the visitor matrix, we identify a vertex v 's *family*: those vertices likely to be

² We never explicitly calculate stability, as it is an expensive global measure, unsuitable for use as a cost function.

³ A vertex with neighbours in ≥ 1 external partitions.

the source of traversals to v . This is the clique of vertices which should accompany a swapping candidate to a new partition.

3.2 Introversion and extroversion

We now formally define a vertex's extroversion (symmetrically, along with its *introversion*), in terms of the VM. Given $v \in V_i$, we have seen that a VM cell $VM_i^{(k+2)}[v_{i_1}, \dots, v_{i_k}, v, v']$ denotes the probability of a transition from v to v' , given a path $p = v_{i_1} \rightarrow \dots \rightarrow v_{i_k} \rightarrow v$ that matches a query pattern. Let $paths(v, V_i)$ denote the set of all such paths in V_i , i.e. those that match a query pattern in $\mathcal{Q}$ and end in v . We define *introversion*(v) of v as the total probability of such transition occurring, summed over every path $p \in paths(v, V_i)$ and every destination vertex $v' \in V_i$. Formally:

$$introversion(v) = \frac{1}{Pr(v)} \sum_p \left(Pr(p) \cdot \sum_{v' \in V_i} VM_i(t)[p, v'] \right) \quad \text{for all } p \in paths(v, V_i) \quad (6)$$

where for path $p = v_{i_1} \rightarrow \dots \rightarrow v_{i_k} \rightarrow v$ of length $k + 1$, we have:

$$\begin{aligned} Pr(p) &= Pr(v | v_{i_1} \rightarrow \dots \rightarrow v_{i_k}) \cdot \\ &Pr(v_{i_k} | v_1 \rightarrow \dots \rightarrow v_{i_{k-1}}) \cdot \dots \cdot Pr(v_{i_2} | v_{i_1}) \cdot Pr(v_{i_1}) = \\ &VM_i^{(k+1)}[v_{i_1}, \dots, v] \cdot \\ &VM_i^{(k)}[v_{i_1}, \dots, v_{i_k}] \cdot \dots \cdot VM_i^{(2)}[v_{i_1}, v_{i_2}] \cdot VM_i^{(1)}[v_{i_1}] \end{aligned}$$

and the total *intra*-partition traversal probability is divided by the total probability of all traversal paths to v :

$$Pr(v) = \sum_{p \in paths(v, V_i)} Pr(p)$$

to account for the percentage of the traversals from v that are internal.

Symmetrically, we define the extroversion of vertex v as the total likelihood of inter-partition traversal $v \rightarrow v'$, where $v \in V_i$ and $v' \in V_j$, $j \neq i$. As the VM is right stochastic and we may assume that a partition's VM forms a sub-matrix of the global VM, *inter*-partition probabilities are the complement to 1 of the *intra*-partition probabilities:

$$extroversion(v) = \frac{1}{Pr(v)} \sum_p \left(Pr(p) \cdot \left(1 - \sum_{v' \in V_i} VM_i(t)[p, v'] \right) \right) \quad \text{for all } p \in paths(v, V_i) \quad (7)$$

4 Prefix Trie encoding of query expressions

We use a prefix trie, which we have called the Traversal Pattern Summary Trie (*TPSTry*), to encode the set of path expressions defined by each new query Q in our workload $\mathcal{Q}$. Combined with continuous tracking of query frequencies over a time window t , the *TPSTry* gives us a compact way to represent legal paths that may lead to each vertex v in G , along with each path's current probability of being traversed. A trie is highly efficient at matching prefixes for multiple sequences or strings. From the stream of regular expressions which comprise the query workload $\mathcal{Q}$, we derive a dictionary set D of all label sequences (strings) described by these expressions. If a sequence of vertices $p_1, \dots, p_n$ is *connected*, such that $(p_n, p_{n-1}) \in E$, and its corresponding sequence of labels $l(p_1) \dots l(p_n)$ is a prefix of some sequence from D , then that sequence is considered *legal*.

The idea of using a trie is inspired by Li et al. [15] who use them to encode sequences of clicked hyperlinks over a web graph, summarising the top k most frequent patterns in web browsing sessions. In our context, a sequence of clicked hyperlinks is just a particular case of generic traversals over more general forms of graph data.

Instead of encoding all actual graph traversals, however, we only encode query patterns in terms of the labels associated with each vertex. Then we associate probabilities to each node in this, smaller, trie of labels. In practice, each path in the trie is an intentional representation of a (possibly very large) set of paths in the graph, namely those whose vertices match the sequence of labels in the trie branch. This representation is very compact, because this trie grows with $|L_V|^t$, where t is the length of the longest path expressed by queries in $\mathcal{Q}$ and L_V is typically small. Of course, one path in the trie now corresponds to a set of paths in the graph. We are going to take this one-many relationship into account when we convert the probabilities associated with nodes in the trie, into the probabilities associated with vertices in the graph, i.e., the elements of the VM.

Given a workload $\mathcal{Q}$, *TPSTry* is constructed by mapping each new regular expression $Q \in \mathcal{Q}$ to a set of strings, and adding these to a trie using standard trie insertion procedure. Each node in the *TPSTry* which corresponds to one of these added strings is then labelled with the expression Q , even if the node existed as the result of a distinct previous expression.⁴ The labels for each query in $\mathcal{Q}$ are hashes of the expressions themselves, as these are guaranteed to be unique.⁵ If an expression is not seen within the preceding time t (i.e. has a frequency of 0) then its label is removed from nodes in the trie; any node without **any** query labels is also removed. Such an infrequent expression is then treated as new in future. The mapping $s = str(Q)$ of a query expression Q to string s is straightforward and is defined as follows (*append*(x, y) simply appends string y to string x):

$$\begin{aligned} str(l) &= \{l\} \text{ for each } l \in L_V \\ str(e_1 \mid e_2) &= str(e_1) \cup str(e_2) \end{aligned}$$

⁴ *TPSTry* nodes may be labelled with multiple queries.

⁵ We use Q_i labels in examples for readability.

Fig. 3 Summary trie construction from queries

$$\begin{aligned} \text{str}(e_1 \cdot e_2) &= \{\text{append}(x, y) \mid x \in \text{str}(e_1), y \in \text{str}(e_2)\} \\ \text{str}(e^N) &= \text{str}(e \cdot e \dots e) \quad // N \text{ times} \end{aligned}$$

Example Consider again the graph in Fig. 1 and the expressions $Q_1 = a \cdot (b|c) \cdot (c|d)$ and $Q_2 = (c|a) \cdot c \cdot a$. These two expressions are encoded using the two prefix trees in Fig. 3a. The two trees are then further combined into the single prefix tree in Fig. 3b, with each node labelled with the set of queries it pertains to.

4.1 Associating probabilities to trie nodes

Given a trie, such as in Fig. 3b, we associate a probability to each node in the trie, reflecting the relative likelihood that a sequence of vertices with those labels will be traversed in the graph. These probabilities are periodically (re)calculated by considering both the individual contribution of each query Q to the trie structure, as well as the frequency with which Q appears in the workload during some preceding time t .

To understand these calculations, consider again Fig. 3, where we assume that Q_1 , Q_2 each occur once in $\mathcal{Q}$ over time t , i.e., they have the same relative frequency. Starting from root $\mathcal{E}$, consider transition $\mathcal{E} \rightarrow a$. Its probability can be expressed as:

$$\Pr(\mathcal{E} \rightarrow a) = \Pr(\mathcal{E} \rightarrow a|Q_1) \cdot \Pr(Q_1) + \Pr(\mathcal{E} \rightarrow a|Q_2) \cdot \Pr(Q_2)$$

where the conditional probabilities are computed using the labels on the nodes and the $\Pr(Q_i)$ are the relative frequencies of the Q_i . In the example we have $\Pr(Q_1) = \Pr(Q_2) = .5$, $\Pr(\mathcal{E} \rightarrow a|Q_1) = 1$ because a is the only possible first match in Q_1 's pattern, and $\Pr(\mathcal{E} \rightarrow a|Q_2) = .5$ because initially Q_2 can match both a and c , with equal probability. Thus, $\Pr(\mathcal{E} \rightarrow a) = 1 \cdot .5 + .5 \cdot .5 = .75$.

We can now use $\Pr(\mathcal{E} \rightarrow a)$ to compute $\Pr(\mathcal{E} \rightarrow a \rightarrow b)$ and $\Pr(\mathcal{E} \rightarrow a \rightarrow c)$:

$$\begin{aligned} \Pr(\mathcal{E} \rightarrow a \rightarrow b) &= \\ &\Pr(\mathcal{E} \rightarrow a \rightarrow b|Q_1) \cdot \Pr(Q_1) + \\ &\Pr(\mathcal{E} \rightarrow a \rightarrow b|Q_2) \cdot \Pr(Q_2) \end{aligned}$$

where

$$\begin{aligned} Pr(\mathcal{E} \rightarrow a \rightarrow b|Q_1) &= \\ Pr(a \rightarrow b|\mathcal{E} \rightarrow a, Q_1) \cdot Pr(\mathcal{E} \rightarrow a|Q_1) &= \\ .5 \cdot 1 &= .5 \end{aligned}$$

and $Pr(\mathcal{E} \rightarrow a \rightarrow b|Q_2) = 0$ because pattern $\mathcal{E} \rightarrow a \rightarrow b$ is not feasible for Q_2 . Thus, $Pr(\mathcal{E} \rightarrow a \rightarrow b) = .5 \cdot .5 = .25$.

Formally, we identify each node n in the trie by the sequence of k steps $(n_1, n_2, \dots, n_k)$ required to reach it from the root node, $\mathcal{E}$. A *probability label* $p(n)$ is then associated with each node, its value computed as follows:

$$\begin{aligned} p(n) &= Pr(\mathcal{E} \rightarrow \dots \rightarrow n_k) = \\ \sum_{Q_i \in \mathcal{Q}} Pr(\mathcal{E} \rightarrow \dots \rightarrow n_k|Q_i) \cdot Pr(Q_i) \end{aligned}$$

The individual terms of the sum are conditional probabilities over the path in the trie to node N . As we have seen in the example, these conditional probabilities over the paths are computed recursively on the length k :

$$\begin{aligned} Pr(\mathcal{E} \rightarrow \dots \rightarrow n_{k-1} \rightarrow n_k|Q_i) &= \\ Pr(n_{k-1} \rightarrow n_k|\mathcal{E} \rightarrow \dots \rightarrow n_{k-1}, Q_i) \cdot \\ Pr(\mathcal{E} \rightarrow \dots \rightarrow n_{k-1}|Q_i) \end{aligned}$$

4.2 Computing VM cells with the TPSTry

The *TPSTry* encodes the current likelihood of traversing from a vertex with some label, to any connected vertex with some other label (Sect. 4.1). This is an abstraction over the values we actually need for the visitor matrix, which are vertex-to-vertex transition probabilities. We may derive the desired vertex transition probabilities, from a path of previously traversed vertices $p = p_1, p_2, \dots, p_k$, as follows. First we look up the the path's corresponding sequence of vertex labels in the pattern summary trie. This returns a set of child trie nodes $n \in N$ which represent legal labels for the next vertex to be traversed, along with each label's associated probability $p(n)$. Subsequently, the traversal probabilities for each label are uniformly distributed amongst those neighbours of p_k which share that label. This produces a vector of traversal probabilities, one for each neighbour of the p_k . This vector corresponds to a row in the visitor matrix.

For each path of traversals with length $< t$, the VM assumes that a subsequent traversal is guaranteed, i.e. the total traversal probability in each row is 1, and the VM is stochastic. In reality some paths of traversals must have a total length $< t$, either because a query expression defines a path of a shorter length, or because a vertex does not have a neighbour with the label required by a query expression. A query execution engine would stop traversing in such a scenario. We represent this non-zero probability

Fig. 4 Visitor Matrix (left), TPSTry probabilities (right)

of no subsequent traversal from a vertex as probability to traverse to the same vertex,⁶ as this is equivalent to intra-partition traversal probability.

In Sect. 2.3 we described a VM cell as containing the probability of traversing to a vertex v given some preceding sequence of traversals $p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_{t-1}$. Formally, we compute the value of a cell $VM^{(t)}[p_1, \dots, p_{t-1}, v]$ as

$$\begin{aligned} Pr(p_{t-1} \rightarrow v | p_1 \rightarrow \dots \rightarrow p_{t-1}) = \\ Pr(l(p_{t-1}) \rightarrow l(v) | \mathcal{E} \rightarrow l(p_1) \rightarrow \dots \rightarrow l(p_{t-1})) \cdot \\ Pr(p_t = v | l(p_t) = l(v), p_t \in N_G(p_{t-1})) \end{aligned}$$

where $l : V \rightarrow L_V$ is the labelling function for a graph G , and $N_G : V \rightarrow V$ corresponds to the set of neighbours of v such that $(v, n) = e \in E$ for all $n \in N_G(v)$. The latter term of this definition uniformly distributes the traversal probability to a vertex with label l across all of p_{t-1} 's l labelled neighbours.

Example Given the graph in Fig. 1, consider the element $VM^{(3)}[1, 2, j]$ in its visitor matrix. The probability to be in vertex 2, having previously been in vertex 1, is given by the matrix's $VM^{(2)}[1, 2]$ th element. The labels of vertices 1 and 2 are a and b respectively. There exist two valid suffixes to the label sequence $a \rightarrow b$: c and d . From the query pattern summary trie in Fig. 4, we know that the relative frequency of c from $a \rightarrow b$ is 0.5.

$$Pr(b \rightarrow c | a \rightarrow b) = \frac{0.125}{0.25} = 0.5$$

The relative frequency of d from $a \rightarrow b$ is also 0.5. Vertex 2 has the neighbours 1, 3, 4 and 5 with the labels a , c , d and c respectively. As an example, the probability

⁶ We do not consider the possibility of self-referential edges; any probability to remain in the same vertex is equivalent to probability of no subsequent traversal.

of traversing to vertex 3 is the probability of traversing to a c labelled vertex, divided by the number of c labelled neighbours of 2.

$$\begin{aligned} VM^{(3)}[1, 2, 3] &= 0.5 \cdot Pr(j = 3 | l(j) = c, j \in N_G(2)) \\ &= 0.5 \cdot 0.5 = 0.25 \end{aligned}$$

Therefore, as shown in Fig. 4 (left), we have $VM^{(3)}[1, 2, *] = (0, 0, 0.25, 0.5, 0.25, 0)$.

In the previous section (Sect. 4.1) we mention that *TPSTry* probabilities are periodically updated to reflect query frequencies changing over time. We do not recompute VM cells for each change to the *TPSTry*, instead they are lazily re-evaluated each time a vertex swapping iteration (Sect. 3.1) is triggered. Additionally, we store a snapshot of the *TPSTry* at the point of the pervious vertex swapping iteration; if a trie node's probability remains the same between two iterations, we are able to safely avoid recomputing its associated VM cells.

5 Implementation

The *TAPER* system consists of a main algorithm for calculating elements of the Visitor Matrix and for deriving the most extroverted vertices for each vertex swapping iteration. The system also implements the *TPSTry* traversal pattern summery trie. In this section we present the *TAPER* prototype architecture, we discuss heuristics for managing the space and time complexity associated with the Visitor Matrix, and we describe in detail the vertex ranking and swapping algorithm that takes place at each iteration.

5.1 Architecture

We have implemented a system prototype on top of the Tinkerpop graph processing framework,⁷ which allows us to use any of several popular GDBMS to store G . Though our prototype, built using the Akka framework,⁸ is designed to be distributed across multiple hosts, in the current implementation input graph partitionings reside on a single host. Partitions are defined in terms of *vertex-cut*, as opposed to *edge-cut*: *inter*-partition connections are represented by flagging cut vertices and annotating them with the partitions they belong to. We have extended Tinkerpop so that multiple edge-disjoint subgraphs are treated as a single, global, graph and queried using the Gremlin query language.⁹ An *inter*-partition traversal is detected when a Gremlin query retrieves the external neighbours of a cut vertex. Our test architecture is shown in Fig. 5. It simulates a distributed deployment, where each partition is logically isolated, managed by a separate instance of the *TAPER* algorithm implementation. Each instance is responsible for updating the Visitor Matrix for its partition, and also determines the rank of extroverted vertices to evict at each iteration.

⁷ The Tinkerpop project: <http://bit.ly/1WNJ7HW>.

⁸ Akka concurrency framework: <http://bit.ly/1B6WXGG>.

⁹ The Gremlin query language: <http://bit.ly/1tqUpWk>.

Fig. 5 Architecture

5.2 Reducing the cost of the visitor matrix

As noted in Sect. 2.3, the space complexity of the VM for each partition V_i grows with the number of vertices in the partition, and exponentially with the length of the query patterns: $O(|V_i|^t)$. Here we discuss two heuristics, aimed at reducing the portion of the VMs that need to be explicitly represented or computed for each partition, reducing both the time and space complexity of the *TAPER* algorithm.

5.2.1 Space complexity

Firstly, we note that large graphs are typically sparse : i.e. $|E| \ll |V|^2$. As each vertex is only connected to a small number of neighbours, the adjacency and transition matrices representing such graphs contain many 0-value elements, which may be discarded, compressing the matrices. A VM, which is essentially a family of k dimensional transition matrices where $2 \leq k \leq t$ and t is the number of traversal steps we remember, can be compressed using this standard technique. Although in general we cannot be certain that the graphs against which *TAPER* is applied will be sparse, the only non-zero elements that may exist in a VM are those that correspond to label paths in the pattern summary trie. This serves to make the VM sparser relative to the corresponding adjacency matrix, **especially** well suited to compression.

Secondly, we avoid the costly computation and storage of many VM rows associated with vertices likely to be “safe”; i.e. vertices unlikely to have high extroversion. Remember that, with *TAPER*, we are only interested in identifying *highly* extroverted vertices. These are the most likely to be the source of *inter*-partition traversals and therefore good candidates for being swapped to another partition. From Eqs. 6 and 7 (Sect. 3.2), we know that such extroverted vertices will necessarily have a low total *intra*-partition traversal probability: low introversion. We therefore declare vertices with introversion above a configurable threshold “safe” and discard them, reducing the space complexity of the VM.

Consider for example vertex 3 (denoted v_3) of partition B in Fig. 1. Accounting for the *TPSTry* of Fig. 3, the traversal probabilities for v_3 are found in $VM_B(3)$ rows $VM_B^{(2)}[3, *]$, $VM_B^{(3)}[5, 3, *]$, $VM_B^{(3)}[6, 3, *]$ and so on. The probability to be in v_3 from

vertex 5 is computed as $VM_B^{(1)}[5] \cdot VM_B^{(2)}[5, 3]$. Extending this, the total intra-partition traversal probability **from** 3, given 5, is

$$VM_B^{(1)}[5] \cdot VM_B^{(2)}[5, 3] \cdot \sum_j VM_B^{(3)}[5, 3, j]$$

Given the values in Fig. 4, completing this process for paths $p \in \text{paths}(v_3, V_B)$ to all $j \in V_B$ gives v_3 an *intra*-partition traversal probability of 0.44. Doing the same for all $j \in V$ gives a **total** traversal probability through v_3 ($Pr(v_3)$) of 0.5. For any choice of introversion threshold less than $\frac{0.44}{0.5} = 0.88$, v_3 would be a safe vertex. We may discard any *VM* rows associated with v_3 except where necessary for paths through other, more extroverted, vertices.

5.2.2 Time complexity

In order to maximise the savings of the heuristic above, we would like to avoid computing some of the matrix rows we eventually discard. We rely upon the following observations to achieve this: as the probability of any given traversal from a vertex is usually less than 1.0, longer paths of traversals generally have a lower probability than shorter ones; the less likely a path of traversals through a vertex v , the less it will contribute to v 's introversion and extroversion; and the *VM* rows for each vertex v are computed in ascending order of the length of their associated paths (Sect. 4.2). Given these observations, we know that for the set of *VM* rows associated with a given vertex v : those rows computed earlier **should** contribute more to v 's introversion and extroversion than those compute later.

We may therefore compare v 's introversion to our chosen “safe” threshold after only having considered paths through v of length up to k , where k is less than the maximum length $k < t$. We then do not need to compute further *VM* rows for safe vertices. In effect this provides another configurable threshold, this time controlling time complexity at the potential expense of accuracy. The smaller the value of k the more likely the algorithm is to declare a vertex safe which actually has a **total** introversion below the “safe” threshold and might therefore have been an effective candidate for swapping to another partition.

Vertices without external neighbours represent a special case of this heuristic. They are guaranteed to be “safe” and have no extroversion. We do not calculate *VM* rows associated with these vertices, except where needed by other paths.

5.3 TPSTry implementation

TPSTry (Sect. 4) is implemented as two separate data structures: (i) a trie multimap, where each trie node maps to the set of queries which could be responsible for a traversal path with the associated sequence of vertex labels; and (ii) a sorted table mapping queries to their respective frequencies. These frequencies are approximated using a sketch datastructure which samples the occurrences of each query within a sliding window of time t .

5.4 Calculating a partial extroversion order

TAPER relies upon an ordering of the vertices in a partition by their likelihood to be the source of *inter*-partition traversals. In order to produce this order, we group the rows of a partition's visitor matrix by the final vertex of the paths they represent and then derive their extroversion (Sect. 3.2).

As a result of the heuristics defined above, not all vertices are represented in the visitor matrix. Therefore we refer to the sorted set of vertices produced as a *partial extroversion ordering*. Rather than grouping the rows of a pre-existing matrix, we define a corecursive algorithm to efficiently produce such rows consecutively during initial VM construction. This greatly simplifies the process of maintaining a running total of *intra*-partition transition probabilities for each vertex, as required for the heuristics presented in Sect. 5.2. A simplified version of the procedure is expressed in Algorithm 1. Consider again our earlier example of vertex 3 in partition *B* (Fig. 1), along with the pattern summary trie in Fig. 4 (right). Vertex 3 has the label *c*, which does exist as a prefix in the trie. It has local neighbours 5 and 6, along with external neighbours 2 and 4, labelled *c,a,b* and *d* respectively. The external transition probability from 3 given a path of (3) is $1 - \Sigma(0, 1, 0)$ multiplied by the probability to have made the sequence of traversals which that path represents ($\frac{0.25}{|c|} = 0.125$). In this case: 0.

Algorithm 1 Calculate the VM_i rows for a vertex *v*

```

path  $\leftarrow$  sequence of vertices (initially (v))
paths  $\leftarrow$  set of paths (initially {(v)})
transitions  $\leftarrow$  vector of probability values for v's neighbours
trie  $\leftarrow$  traversal pattern summary trie
threshold  $\leftarrow$  safe introversion value
length  $\leftarrow$  max length of a path in trie
rows  $\leftarrow$  map of path  $\rightarrow$  transitions vectors
introversion(rows)  $\leftarrow$  total introversion of a set of  $VM_i$  rows

```

```

calcVMRows(paths, transitions, rows)
  newPaths  $\leftarrow \emptyset$ 
  for path in paths do
    if path size > length then return rows
    if path in trie then
      transitions  $\leftarrow$  probabilities from trie given path
      rows  $\leftarrow$  rows + (path  $\rightarrow$  transitions)
      if introversion(rows) > threshold then
        rows  $\leftarrow \emptyset$ 
      stop calcVMRows
    neighbours  $\leftarrow$  path.head.neighbours
    for n in neighbours do
      newPaths  $\leftarrow$  newPaths + n prepended to path
  return calcVMRows(newPaths, transitions, rows)

```

The prefixes *cc* and *ac* also exist in the trie, therefore (5, 3) and (6, 3) are further potential paths through 3. The external transition probabilities from 3 given paths of (5, 3) and (6, 3) are $(1 - \Sigma(0, 0, 1)) \cdot 0.125$ and $(1 - \Sigma(0, 0.25, 0.5)) \cdot 0.25$ respectively.

Fig. 6 Offer/receive algorithm in each *TAPER* instance

Note that the total probability for the path $(6, 3, j)$, $j \in V_B$ is < 1 because acd is also a prefix in the pattern summary trie, and vertex 3 is adjacent to the “external” vertex 4. The final external transition probability from 3 is 0.06; its *extroversion* $0.06 \cdot \frac{1}{0.5} = 0.12$.

5.5 Vertex swapping

To achieve its aims, *TAPER* improves distributed query performance by reducing the probability of inter-partition traversals when answering queries. For each partition, given a sorted collection of the vertices with the highest extroversion, *TAPER* must reduce this probability without mutating the underlying graph structure.

To this end, we propose a simple variation on the k -way Kernighan-Lin algorithm proposed by Karypis and Kumar [12] (Sect. 3.1). This is a two-step, symmetric process, shown in Fig. 6: firstly, given a priority queue of candidate vertices with high extroversion, compute the preferred destination partition for each vertex, along with the clique of neighbours which should accompany it (its *family*); secondly, when offered a new group of vertices, a partition should compute potential gains in introversion and decide whether or not to accept the offer.

We determine a swapping candidate’s family set with a simple recursive procedure: Given each family member (initially just the candidate), we examine its local neighbours; if a traversal from a neighbour to the member is more likely than not, then it is added to the family. Once the family-set has been determined, we evaluate the total loss in introversion the sending partition would suffer from their loss. This process is highly efficient as all the relevant values are preserved in the visitor matrix, either from calculating the introversion of vertices, or from constructing a candidates set in the previous step.

When on the receiving end of a swap, a partition should calculate the total local introversion of a family, and compare it to the potential loss to the offering partition. Partitions are “cooperative” rather than greedy, so if the introversion gain for a receiving partition is not greater than the loss for a sending one, the swap is rejected. If a swap is rejected the offering partition will try less “preferable” destinations until all partitions adjacent to the candidate vertex are exhausted. In this event the candidate vertex and its family remain in their original partition.

This process runs independently for each partition, swapping extroverted vertices to their preferred neighbouring partitions. A vertex may only be swapped once per iteration of the algorithm. Note that, because partitions calculate potential vertex swaps independently of one another, it is possible that highly extroverted connected vertices would “flip-flop” between two partitions without ever improving *workload-aware stability*. In order to avoid this we adopt another technique from (Par)Metis [13]. First, every partition is assigned an ordered identifier. Subsequently, we split vertex swapping into two phases: Initially, given its queue of extroverted vertices, a partition will only attempt swaps whose destination partition has an id lower than itself. In the second phase, each partition will do the reverse, swapping vertices with partitions of a higher id. For each iteration, the ordering of these phases flips (higher→lower, lower→higher, ...). This process prevents vertices “chasing” one another continuously by ensuring that when the introversion loss is calculated by the second partition, both offending vertices will be present locally.

When swaps have been attempted for every vertex in each partition’s extroverted queue, the iteration is considered finished. Where necessary, we update each partition’s VM by adding and removing rows for swapped vertices, the resulting subgraph acts as input to a subsequent iteration of vertex-swapping. Repeated iterations of this process will produce the desired result: an enhanced partitioning with a lower overall probability for inter-partition traversals, better *workload-aware stability* given a query workload $\mathcal{Q}$.

At this point, recall that we do not compute or store all VM rows associated with *introverted* vertices, only those associated with paths through other more extroverted vertices. A nice property of this approach is that whilst these safe vertices may become extroverted as we swap their neighbours between partitions, the VM rows which contribute the most to this new extroversion are, by construction, those which already existed for computing the extroversion of swapped neighbours. As a result we do not need to compute new VM probabilities in each iteration, instead retaining the VM computed at the start of each distinct TAPER invocation.

6 Evaluation

Our evaluation aims to show how *TAPER* achieves and maintains high partitioning quality, measured as low *inter*-partition traversals. We present three main results. The first two on the effect of a single *TAPER* invocation given a workload snapshot $\mathcal{Q}$ (i.e. Eq. 1, from Sect. 1.1): (1) given a simple hash-partitioning $P_k^0(G)$ and a workload $\mathcal{Q}$, a single *TAPER* invocation achieves a quality comparable to that of a Metis-partitioning [12] in at most 8 iterations; (2) given the same workload, along with input partitionings generated by proven existing techniques, a *TAPER* invocation is still able to achieve significant quality improvements.

The third result is on the impact of a changing workload, given periodic *TAPER* invocations (i.e. Eq. 2 from Sect. 1.1): (3) given a workload stream $\mathcal{Q}_1, \mathcal{Q}_2, \dots$, our system maintains an up-to-date query summary in the *TPSTry*. As a result, repeated *TAPER* invocations are able to keep *ipt* below some desired minimum, despite any workload changes.

We use Metis [12] as our primary basis for comparison because, despite its age, it remains a gold standard for producing quality workload-agnostic partitionings of medium sized graphs [16,28]. As such it is a compelling yard-stick for our evaluation of the *TAPER* prototype, which will consider partitioning quality in terms of scale free metrics such as *ipt* and the number of vertex swaps. We avoid implementation dependent metrics because, as a prototype, *TAPER* is unlikely to exhibit realistic performance. For instance, without true distributed query processing (Sect. 5.1) across a network, query response times are meaningless as a measure of partitioning quality.

6.1 Experimental setup

For all experiments we initially partition the test graphs using either hash or Metis. Except where otherwise stated, these are 8-way partitions. As mentioned, we consider the quality of a partitioning as its workload-aware stability (Sect. 2.2), corresponding to the probability of inter-partition traversals when executing a workload Q . We measure this experimentally by executing snapshots of query workloads over partitioned graphs and counting (Sect. 5) the number of inter-partition traversals *ipt*. All algorithms, data structures and dataset pre-processing steps, including *calcVMRows* and the *TPSTry*, are publicly available.¹⁰ All our experiments are performed on a machine with a 3.1Ghz Intel Core i7 CPU and 16GB of RAM.

6.1.1 Test datasets

TAPER is designed to perform best on heterogeneous graphs (Sect. 4). When the graph is homogeneous, the uniformity of traversal probabilities renders min edge-cut an equally good measure of partition quality. Thus, we have tested the system on two heterogeneous graphs.

The first, **MusicBrainz**,¹¹ is a freely available database of music which contains curated records of artists, their affiliations and their works. This database currently stores over 950,000 artists and over 18 million tracks. When converted to a graph, the subset of data we use amounts to around 10 million vertices and more than 30 million edges. The graph is also highly heterogeneous, containing more than 12 distinct vertex labels.

The second test dataset is a synthetic provenance graph, generated using the **ProvGen** generator [6] and compliant with the PROV data model [19]. ProvGen is designed to produce arbitrarily large PROV graphs starting from small seed graphs and following a set of user-defined topological constraint rules. Provenance graphs are a form of metadata, which contains records of the history of entities, e.g. documents, complex artifacts, etc... They are exemplars of the large-scale heterogeneous graphs that *TAPER* is designed to partition. For the purpose of these experiments we generated a graph with about 1 million vertices and 3 million edges. As described in [19], PROV graphs naturally have three labels, representing the three main elements of provenance:

¹⁰ The *TAPER* repository: <http://bit.ly/1W3f0eH>.

¹¹ The MusicBrainz database: <http://bit.ly/1J0wlNR>.

Entity (data), *Activity* (the execution of a process that acts upon data), and *Agent*, namely the people, or systems, that are responsible for data and activities.

6.1.2 Test query workloads

For each dataset we need to create a corresponding query stream: an infinite sequence of path queries consisting of a small number of distinct graph patterns. The relative frequencies of each query pattern should shift continuously, representing workload changes with time. For our experiments, we selected a simple periodic model of workload change where the frequency of each query pattern grows and shrinks according to a constant, repeating pattern¹² and no new query patterns are added over time. These frequency changes are the complement of each other, so that the total frequency of all query patterns in the workload stream is always equal to 1. Note that *TAPER* **does not assume** any such distribution of query frequencies, and can refine a graph partitioning given arbitrary changes in workload.

We also define the set of distinct query patterns for each dataset. Regarding MusicBrainz, to the best of our knowledge there is no widely accepted corpus of benchmark queries. Thus, we define a small set of common-sense queries that focus on discovering implicit relationships in the graph, such as collaborations between artists, and migrations between geographical areas.

MQ₁ *Area · Artist · (Artist|Label) · Area*: searches for two distinct patterns which would indicate an artist has moved away from their country of origin.

MQ₂ *Artist · Credit · (Track|Recording) · Credit · Artist*: might be used to detect collaboration between 2 or more artists on a single track.

MQ₃ *Artist · Credit · Track · Medium*: would return a set of all the Mediums (e.g. Cd) which carry an Artist's work.

Regarding provenance graphs, several categories of typical path query have been proposed [11]. Using these categories, we propose four query patterns typical of provenance analysis.

PQ₁ *Entity · (Entity) * · Entity*: computes the transitive closure over a data derivation relationship.

PQ₂ *Agent · Activity · Entity · Entity · Activity · Agent*: identifies pairs of agents who have collaborated as data producer/consumers pairs.

PQ₃ *(Entity) * · Activity · Entity*: returns all entities and all activities involved in the creation of a given entity.

PQ₄ *Entity · Activity · (Agent)**: returns agents responsible for the creation of a given an entity.

¹² Details of the workload stream are elided for space.

Fig. 7 ipt per *TAPER* internal iteration

6.2 Results

6.2.1 Improvement over an initial hash partitioning

Figure 7 shows the improvement in partitioning quality which a single *TAPER* invocation achieves for each dataset, given static workload snapshots Q and initial hash-partitionings $P_g^0(G)$. The top dotted line bisecting the left y-axis is our baseline: the ipt required to execute Q over $P_g^0(G)$. The bottom dotted line indicates the ipt required to execute Q over an initial Metis partitioning. The chart shows how partitioning quality converges to within 10% of that over a Metis partitioned graph, after fewer than 8 internal iterations. Note that these iterations satisfy a maximum partition imbalance of 5%. Also note that, though we do not fully evaluate running time, the longest iteration over the MusicBrainz graph takes around 250s. This running time generally decreases with each successive iteration, with the average being less than 150s. We are confident that optimisation work will be able to reduce this time significantly.

Figure 7 also demonstrates that a *TAPER* invocation requires far less communication than a full Metis repartitioning. A **Lower bound** for the number of vertex swaps required for Metis to repartition a Hash partitioning of the ProvGen graph is around 500k. On the other hand, 5 iterations of *TAPER* over the ProvGen graph (Fig. 7a) require just 300k vertex swaps to produce ~80% enhancement.

We use a lower bound in our comparisons because, as mentioned (Sect. 1.3), there are multiple different implementations of Metis [12, 13] which will exhibit different numbers of vertex swaps during a repartitioning. Rather than compare to each of these systems, we simply calculate an absolute lower bound for their performance by observing that, for the ProvGen graph, a Metis partitioning cuts around 500k fewer edges than a Hash partitioning. Regardless of the repartitioning algorithm used, a reduction in the edge-cut of a partitioning by 1 must intuitively cost at least 1 vertex swap. In reality, the number of vertex swaps caused by Metis would be much higher. For instance, conventional Metis would have to collect the entire graph to a single partition in order to compute its partitioning, then redistribute. Even if the overhead of gathering the graph is not considered, Metis usually requires around 97% of vertices to be swapped when repartitioning [24]. Meanwhile, the number of vertex swaps caused

Fig. 8 *ipt* per approach

by ParMetis’ diffusion based repartitioning approach is $\sim 1.5X$ the edge-cut reduction produced [24].

Practically, a Metis repartitioning has a cost **at least 2X** that of a *TAPER* invocation in both our test cases, yet achieves only a small improvement in query performance. This suggests that, by performing swaps in extroversion order (Sect. 3.1), we are correctly prioritising those swaps that are more effective at reducing *ipt*, given a workload snapshot Q . This supports *TAPER*’s applicability to continuous re-partitioning in online settings, such as distributed graph DBMS, where other system requirements may severely limit the number of vertex swaps possible in a given timeframe.

6.2.2 Improving over other initial partitionings

Figure 8 illustrates that a *TAPER* invocation may achieve a quality improvement over not only an initial hash-partitioning, but also over initial partitionings produced with existing partitioning techniques, e.g. Metis. When improving upon a Metis partitioning (METIS + *TAPER* in the figure), *TAPER* averages a 30% reduction in *ipt*. As seen in the previous section, a *TAPER* invocation over an initial hash-partitioning achieves a quality less than an initial Metis partitioning. Thus we conjecture that the *TAPER* algorithm is sensitive to its starting input and, despite swapping vertex *family* cliques (Sect. 5.5), when starting from a Hash partitioning is gets trapped in local optimisation minima. Starting from a Metis partitioning, *TAPER* iteratively approaches a new minimum closer to the global.

TAPER’s ability to improve over Metis graphs may be explained by observing that in non-trivial partitionings, some edges must cross partition boundaries. As a workload-agnostic algorithm, Metis is optimising for a different cost function than *TAPER* and may cut edges which are likely to be frequently traversed, giving *TAPER* scope for its

Fig. 9 *ipt* versus k partitions (ProvGen)

improvement. Note that improvement is not necessarily possible when Metis is given an input graph with edge-weights corresponding to traversal likelihood given Q . In that instance, edge-weight cut is equivalent to inter-partition traversal probability: both Metis and *TAPER* are optimising for the same cost function. However, tracking an online workload with edge-weights is challenging and highly expensive [32]. Also, adapting to any workload changes with Metis would still require a repartitioning, which we know to be more costly than *TAPER* in terms of vertex-swaps.

6.2.3 The effect of differing numbers of partitions

Figure 9 demonstrates the applicability of a *TAPER* invocation to partitionings with different numbers of partitions (different values of k). The figure presents the *ipt* caused by executing a query workload Q over several distinct partitionings of the ProvGen graph: four for each of the partitioning approaches we consider, with k values 2, 8, 16 and 32. The improvement in *ipt* offered by *TAPER* is readily apparent across this range of k .

As the number of partitions k grows, there is a higher probability that parts of a graph traversed by the same query will be spread across multiple partitions. Combined with partition balance constraints,¹³ this results in an increase of *absolute ipt* when executing Q over a *TAPER* partitioning. However, increasing k also increases the general probability that any two vertices which share an edge are split between partitions, thus reducing the quality of Hash and Metis partitionings as well. This effect is particularly pronounced for Hash partitionings. Indeed, for a partitioning where $k = 2$,

¹³ Which prevent *TAPER* from amassing extroverted vertices to a single partition, reducing *ipt* regardless of k .

Fig. 10 *ipt* per query

TAPER achieves only around 50% reduction in *ipt* relative to Hash; for $k = 16$ this reduction is over 80%.

6.2.4 Optimising for frequent queries

Figure 10 demonstrates the effect of *TAPER*'s use of query frequencies within a workload to prioritise vertex swaps. The figure presents *ipt* over various partitionings of the MusicBrainz graph, given a workload snapshot with the relative frequencies of queries **MQ₁**, **MQ₂**, and **MQ₃** at 10, 20 and 70%, respectively. Relative to the Metis partitioning, a *TAPER* invocation achieves its worst quality for **MQ₁**, improving with **MQ₂** and surpassing the other system for **MQ₃**. This is because paths in a graph which form a full, or partial, match of a high frequency query afford their vertices and edges a higher probability of being traversed. When edges in the path cross partition boundaries, this traversal probability contributes to extroversion. Again, *TAPER* is prioritising vertex-swaps to internalise paths traversed by the most common queries to single partitions.

6.2.5 The effect of changes in query workloads

So far in our evaluation we have performed single invocations of *TAPER*: several iterations of vertex-swapping over an initial partitioning, given a static snapshot of queries. This is essentially **fitting** the distribution of vertices across partitions to a particular workload snapshot's dominant traversal patterns (Sect. 2.2). However, within a larger workload stream, query frequencies are likely to change continuously. Figure 11

Fig. 11 *ipt* versus workload change

trivially demonstrates that the quality of a fitted partitioning degrades in the presence of such workload change.

For simplicity this experiment was performed over the provenance dataset, with a finite workload stream comprised of two query patterns, traversing single edges guaranteed not to be incident to the same vertex: Q_a *Entity · Entity* and Q_b *Agent · Activity*. These queries were chosen to avoid overlap between the traversals required for a new query workload and those optimised for by *TAPER* given the previous one, thereby producing a clearer performance trend. At the head of the stream, the frequency of Q_a is 100%; throughout the stream the frequency of Q_a tends linearly to 0%, Q_b to 100%. The initial partitioning has been pre-improved with *TAPER*, assuming a workload of 100% Q_a queries. As the frequency of Q_b queries increases, so does the *ipt*. For comparison, the top dotted line in Fig. 11 shows the *ipt* required to execute solely Q_b queries over a hash-partitioning of the graph; the bottom line shows those required over a partitioning improved by *TAPER* correctly assuming $Q_b = 100\%$. In other words, in the presence of an unexpected change in workload, *TAPER*'s quality improvement may degrade to near that of a naive hash-partitioner.

However, the *TPSTry* is continually updated to reflect changing query frequencies (Sect. 4) and our experiments depicted in Fig. 7 demonstrate that *TAPER* invocations are inexpensive compared to a full re-partitioning operation. Therefore, by periodically executing *TAPER* invocations with the current partitioning as input, we are able to maintain our partitioning quality improvement even in the presence of a dynamic and changing workload stream. Figure 12 presents the *ipt* which occur when executing a full streaming query workload, generated as described (Sect. 6.1.2), over the MusicBrainz graph partitioning. Knowing the *ipt* required to execute each query pattern over a hash partitioning, the chart displays a derived trendline for baseline performance. We denote a single “period” of the periodic workload stream on the x -axis.

Fig. 12 *ipt* over time w. *TAPER* invocations

Each stream period starts with the highest frequency of the cheapest query pattern, hence baseline *ipt* starts at a minimum. As the frequency of queries which return more results rises, then falls, the *ipt* follows suit. Comparing against this baseline, Fig. 12 clearly demonstrates that, with periodic invocations, *TAPER* is able to prevent some performance decay over time. The highlighted areas of the chart indicate when *TAPER* has been executed; each followed by a drop in *ipt*, as we expect.

In this experiment we trigger *TAPER*'s execution at regular intervals, which is naive, as invocations may occur when a trend in the workload renders them unnecessary or detrimental. For instance, the second highlighted invocation acts on workload information which quickly becomes “stale”. This actually causes a slight *rise* in *ipt*; a risk when tracking values which change frequently. Identifying more effective trigger conditions is left as future work.

7 Conclusions

We have presented *TAPER*: a practical system for improving path query processing performance in partitioned graph data. By monitoring the traversals and frequencies associated with queries in a workload stream, we can calculate the likelihood for any vertex in a graph to be a source of costly inter-partition traversals - its extroversion. By using vertex labels as an intensional representation of traversal patterns, along with several other heuristics, the resource intensive challenge of identifying and relocating these most extroverted vertices becomes tractable.

Our experiments show that *TAPER* significantly reduces the number of inter-partition traversals (*ipt*) over a graph partitioning. It achieves improvements similar to quality existing partitioners, such as Metis, whilst requiring a lower total communication volume, even after many internal iterations of its vertex-swapping algorithm. Furthermore, as it is workload-aware, *TAPER* may even improve the quality of input partitionings already good *w.r.t.* some workload agnostic objective function, such as min edge-cut.

Our final experiments show that as a result of a continuous workload summary (*TPSTry*), and the incremental nature of *TAPER* invocations, we are able to maintain high partitioning quality (*workload-aware stability*) in the presence of a changing stream of queries. This renders *TAPER* suitable for use in online scenarios, where such dynamic workloads are common.

As mentioned (Sect. 2), in *TAPER* we only consider queries as single regular expressions over vertex labels when encoding traversal patterns in the *TPSTry*. For future work we plan to extend our workload summary, including edge labels and more complex query patterns. This will increase the accuracy of extroversion orderings, improving performance. We also plan to explore more sophisticated, predictive, trigger conditions for *TAPER* invocations when given a workload stream, as the current regular intervals are ineffective.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Barcelo, P., Hurtado, C.A., Libkin, L., Wood, P.T.: Expressive languages for path queries over graph-structured data. In: Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), pp. 3–14 (2010)
2. Chen, L.: Distance-join: pattern match query in a large graph. *Sci. Technol.* **2**(1), 886–897 (2009)
3. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach database replication and partitioning. *Proc. VLDB Endow.* **3**(1–2), 48–57 (2010)
4. Delvenne, Jc, Schaub, M.T., Yaliraki, S.N.: The stability of a graph partition: a dynamics-based framework for community detection. *Dyn. Complex Netw.* **2**, 221–242 (2013)
5. Fiduccia, C., Mattheyses, R.: A linear-time heuristic for improving network partitions. In: Proceedings of the 19th Design Automation Conference (1982)
6. Firth, H., Missier, P.: ProvGen: generating synthetic PROV graphs with predictable structure. In: 5th International Provenance and Annotation Workshop, (IPAW), pp. 16–27 (2014)
7. Firth, H., Missier, P.: Workload-aware streaming graph partitioning. In: Workshop Proceedings of the EDBT/ICDT 2016 Joint Conference (2016)
8. Hendrickson, B., Leland, R.: An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.* **16**(2), 452–469 (1995)
9. Huang, Z., Chung, W., Ong, T.H., Chen, H.: A graph-based recommender system for digital library. In: Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries, pp. 65–73 (2002)
10. Jindal, A., Dittrich, J.: Relax and let the database do the partitioning online. In: Enabling Real-Time Business Intelligence, pp. 65–80 (2012)
11. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying data provenance. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10), pp. 951–962. ACM, New York (2010)
12. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.* **47**(2), 109–124 (1997)
13. Karypis, G., Kumar, V.: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.* **48**(1), 71–95 (1998)
14. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* **49**(2), 291–307 (1970)
15. Li, H., Lee, S.: Mining top-K path traversal patterns over streaming web click-sequences. *J. Inf. Sci. Eng.* **1133**(95), 1121–1133 (2009)

16. Margo, D., Seltzer, M.: A scalable distributed graph partitioner. *Proc. VLDB Endow.* **8**(12), 1478–1489 (2015)
17. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. *SIAM J. Comput.* **24**(6), 1235–1258 (1995)
18. Mondal, J., Deshpande, A.: Managing large dynamic graphs efficiently. In: *Proceedings of the 2012 international conference on Management of Data*, pp. 145–156 (2012)
19. Moreau, L., Missier, P., Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., Miles, S., Myers, J., Sahoo, S., Tilmes, C.: PROV-DM: the PROV data model technical reports. In: *World Wide Web Consortium* (2012)
20. Pavlo, A., Curino, C., Zdonik, S.: Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: *Proceedings of the 2012 international conference on Management of Data*, p. 61 (2012)
21. Pujol, J.M., Erramilli, V., Siganos, G., Yang, X., Laoutaris, N., Chhabra, P., Rodriguez, P.: The little engine(s) that could. In: *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 375–386 (2010)
22. Quamar, A., Kumar, K.A., Deshpande, A.: SWORD: scalable workload-aware data placement for transactional workloads. In: *Proceedings of the 16th International Conference on Extending Database Technology*, p. 430. ACM Press, New York (2013)
23. Sanders, P., Schulz, C.: Think locally, act globally: highly balanced graph partitioning. In: *International Symposium on Experimental Algorithms*, pp. 164–175. Springer, New York (2013)
24. Schloegel, K., Karypis, G., Kumar, V.: Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.* **47**(2), 109–124 (1997)
25. Shang, Z., Yu, J.X.: Catch the Wind: graph workload balancing on cloud. In: *IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 553–564 (2013)
26. Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1222–1230 (2012)
27. Tong, H., Gallagher, B., Faloutsos, C., Eliassi-Rad, T.: Fast best-effort pattern matching in large attributed graphs. In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, p. 737 (2007)
28. Tsourakakis, C., Gkantsidis, C., Radunovic, B., Vojnovic, M.: FENNEL: streaming graph partitioning for massive scale graphs. In: *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pp. 333–342 (2014)
29. Vaquero, L.M., Cuadrado, F., Logothetis, D., Martella, C.: Adaptive partitioning for large-scale dynamic graphs. In: *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*, pp. 144–153 (2014)
30. Xu, N., Chen, L., Cui, B.: LogGP: a log-based dynamic graph partitioning method. *Proc. VLDB Endow.* **7**(14), 1917–1928 (2014)
31. Xu, N., Cui, B., Chen, L., Huang, Z., Shao, Y.: Heterogeneous environment aware streaming graph partitioning. *IEEE Trans. Knowl. Data Eng.* **27**(6), 1560–1572 (2015)
32. Yang, S., Yan, X., Zong, B., Khan, A.: Towards effective partition management for large graphs. In: *Proceedings of the 2012 International Conference on Management of Data*, pp. 517–528. ACM Press, New York (2012)