School of Computing Science

Workload-sensitive Approaches to Improving Graph Data Partitioning Online.

Hugo Firth

Submitted for the degree of Doctor of Philosophy in the School of Computing Science, Newcastle University

July 2018

Abstract

Many modern applications, from social networks to network security tools, rely upon the graph data model, using it as part of an offline analytics pipeline or, increasingly, for storing and querying data online, e.g. in a graph database management system (GDBMS). Unfortunately, effective horizontal scaling of this graph data reduces to the NP-Hard problem of "k-way balanced graph partitioning".

Owing to the problem's importance, several practical approaches exist, producing quality graph partitionings. However, these existing systems are unsuitable for partitioning online graphs, either introducing unnecessary network latency during query processing, being unable to efficiently adapt to changing data and query workloads, or both. In this thesis we propose partitioning techniques which are efficient and sensitive to given query workloads, **suitable for application to online graphs and query workloads**.

To incrementally **adapt** partitionings in response to workload change, we propose *TAPER*: a graph repartitioner. *TAPER* uses novel datastructures to compute the probability of expensive *inter*-partition traversals (*ipt*) from each vertex, given the current workload of path queries. Subsequently, it iteratively adjusts an initial partitioning by swapping selected vertices amongst partitions, heuristically maintaining low *ipt* and high partition quality with respect to that workload. Iterations are inexpensive thanks to time and space optimisations in the underlying datastructures.

To incrementally **create** partitionings in response to graph growth, we propose *Loom*: a streaming graph partitioner. *Loom* uses another novel datastructure to detect common patterns of edge traversals when executing a given workload of pattern matching queries. Subsequently, it employs a probabilistic graph isomorphism method to incrementally and efficiently compare sub-graphs in the stream of graph updates, to these common patterns. Matches are assigned within individual partitions if possible, thereby also reducing *ipt* and increasing partitioning quality *w.r.t* the given workload.

Both partitioner and **re**partitioner are extensively evaluated with real/synthetic graph datasets and query workloads. The headline results include that *TAPER* can reduce *ipt* by upto 80% over a naive existing partitioning and can maintain this reduction in the event of workload change, through additional iterations. Meanwhile, Loom reduces *ipt* by upto 40% over a state of the art streaming graph partitioner.

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University or any other institution.

Hugo Firth

January 2018

Publications

Significant portions of the work presented within this thesis have been documented in the following publications:

JOURNAL

1. **H. Firth** and P. Missier, *TAPER: query-aware, partition-enhancement for large, heterogenous graphs*, Distributed and Parallel Databases, 35(2) Special Issue: Distributed Graph Processing and Management, 85-115, June 2017

CONFERENCE

1. **H. Firth** and P. Missier, *Loom: Query-aware Partitioning of Online Graphs*, Proceedings of 21st International Conference on Extending Database Technology (*EDBT*), March 2018
2. **H. Firth** and P. Missier, *ProvGen: Generating Synthetic PROV Graphs with Predictable Structure*, Proceedings of 5th International Provenance and Annotation Workshop (*IPAW*), June 2014

WORKSHOP

1. **H. Firth** and P. Missier, *Workload-aware streaming graph partitioning*, Proceedings of the Joint EDBT/ICDT Workshops (*GraphQ*), March 2016

Acknowledgements

I'd like to start by thanking my PhD supervisor, Dr. Paolo Missier, for all his patience, advice and support throughout my studies. I apologise for my tendency to over-engineer: both sentences and programs.

More generally, I'd like to thank some of the people I have worked alongside: Jack Aiston, for his help understanding the arcane mysteries of combinatorics; Naomi Hannaford, both for being really supportive and for suffering through my loudness; finally, Thomas Cooper and Lauren Roberts, who kept things interesting with the occasional much needed distraction.

Thank you to my family, whose support and belief, though occasionally daunting, means the world to me. Huge thank yous to Rob Thompson and Wojciech Musialkiewicz, who are the best team. Finally, and most importantly, love and thanks to my partner Lisa, whose contributions towards who I am are elided for space.

Contents

List of Figures

List of Tables

1 Introduction

Contents

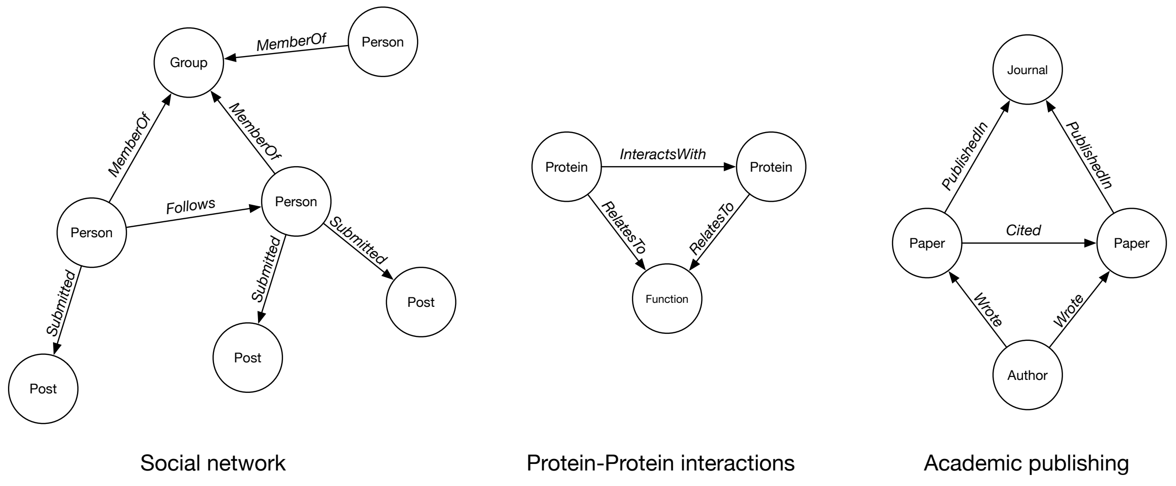


Figure 1.1: Example graph representations of various data models.

Structuring data as a graph, representing records as labelled vertices with interconnecting edges corresponding to relationships, is increasingly common in many application domains. These include social networks [16, 47, 122], Bioinformatics tools [10, 121] and structured document networks such as academic papers and the world wide, or semantic webs [4, 21, 39].

There are several reasons for this recent prevalence. Notably, graphs are a very natural representation of data, sidestepping issues of complex schema design [5]. Consider the examples presented in Figure 1.1. In a social network a *Person* record might have a *Submitted* relationship with several *Post* records, along with *MemberOf* relationships to several *Group* records. Meanwhile, Bioinformaticians often model the interaction between the proteins in living cells using *protein-protein interaction* (PPI) graphs [121], where vertices may represent individual proteins or the bodily functions to which proteins relate (e.g. cell growth). In a more traditional tabular format, such as found in relational database management systems (RDBMS), the edges in Fig. 1.1 would require structures like pivot tables or data duplication (i.e. denormalisation) to express.

Furthermore, storing data in a graph format also renders many classes of operation efficient and/or simple to express. A classic example of such an operation is the **shortest-path** query [69]: selecting the smallest possible sequence of edges which connect any two given vertices. Shortest-path queries are often used for physical route planning, or finding chains of social connections, known as "friend-of-friend". In a tabular format, each additional "of-friend" step requires an expensive scan of the data or index lookup: a **join** in RDBMS parlance.

On the other hand, when storing data in a graph format, such queries typically correspond to a **single** index lookup and subsequent traversal of a small number of graph edges. Edge traversal is analogous to pointer dereferencing, significantly faster than an index intensive join [100]. Specialised graph analysis frameworks[1], graph database management systems (GDBMS)[2] and certain RDF Stores[3] exist to exploit these advantages.

Note that, with the exception of simple aggregation tasks such as finding the average degree[4], the majority of graph operations are implemented using a large number of these efficient edge traversals.

Besides shortest-path, the famous PageRank algorithm [88] can be written as a graph operation [75]. Each document is represented as a vertex with an initial rank. For each vertex v in turn, the algorithm **traverses** all outgoing edges (links)[5] and updates the rank of all neighbours, based on the rank of v. This process is repeated until the ranks of vertices converge to stability.

Additionally, consider the common follower recommendation feature of social networks [47]. Given the example in Fig. 1.1, *Person a* might wish to be recommended other members of their *Group* who they do not *Follow*. This is known as a sub-graph pattern matching query, and is implemented in GDBMS as a series of edge-traversals [100] from a given starting vertex (e.g. *Person a*).

Pagerank is a classic example of an **offline analytical operation**: a slow running task, often executed as a one-off (offline), e.g. by an analyst, or very infrequently. Sub-graph pattern matching queries may also be executed infrequently, often in large batches, constituting offline analytical operations. However, they are more typically an example of an **online data-management operation**: a fast and relatively cheap operation often executed automatically and frequently (online). For example, as part

[1]e.g. Pregel [74], GraphX [124], GraphChi [66] and others
[2]e.g. Neo4j [86], TitanDB [114], Trinity [108]
[3]e.g. Apache Jena [14], Virtuoso [30]
[4]The average number of edges incident to each vertex in a graph.
[5]In some literature this traversal is referred to as message passing between vertices. For the purposes of this thesis, message passing and traversal are considered equivalent.

of a read request to a GDBMS, to find and return a small number of records; or as part of a write request, to find the area of a graph to update or add to.

Note that online data-management operations are often collectively referred to as Online Transactional Processing (OLTP). However, we do not use that acronym throughout this thesis, simply referring to operation types in full, or as online/offline. This avoids confusion with types of operation we do not consider, such as online analytical, and with the common misunderstanding that OLTP refers only to operations executed in strongly consistent relational database management systems.

Despite the clear advantages presented above, there are some potential drawbacks to the graph representation of data, namely concerning the efficiency of systems which make use of it at scale. It is the scalability of graph based applications which we aim to improve with this work, particularly in an online context.

1.1 Graph Partitioning

To understand the scalability concerns surrounding graphs, first note that applications from domains where we argue graphs are most useful produce large quantities of data, often continuously. This is especially true of social networks: a widely used Twitter dataset from 2010 [65] contains 1.5 billion *Follow* relationships, whilst Facebook's graph data may now contain as many as many as 13 billion vertices [45]. Given such quantities of data, systems must scale **up** or scale **out** regardless of representation.

Scaling up describes the process of increasing the effective resources of a single machine, using specialised hardware[6] or techniques, in order to process large amounts of data. Notably, this is the approach taken by the GraphChi analysis framework [66], which applies a "parallel sliding windows" approach to only process a small subset of a graph at any time. The downside to scaling up is that the specialised hardware involved is very expensive and still has some fundamental scalability limits which are difficult to overcome (e.g. network bandwidth).

Scaling out is more common, and usually implies data sharding: splitting data into a number of similar chunks in order to take advantage of the increased resources of

[6]i.e. "Supercomputers" with hundreds of cores and thousands of Gigabytes of RAM.

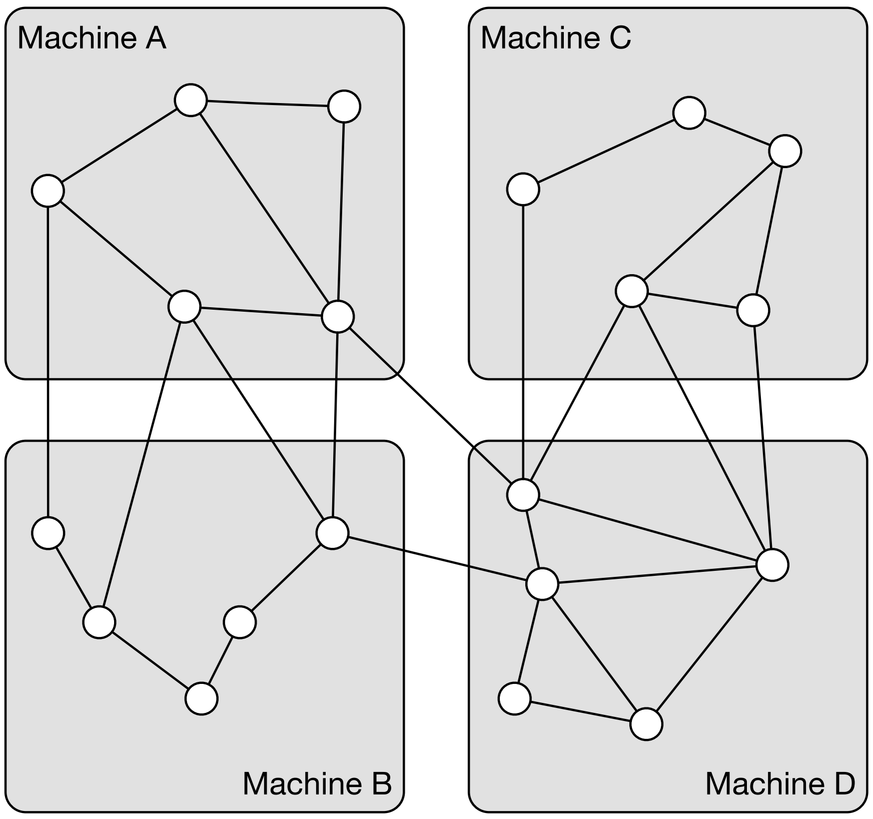


Figure 1.2: A 4-way graph partitioning distributed across a cluster of machines

a homogeneous cluster of machines. Over graph structured data, effective sharding is equivalent to the well-known NP-Complete [3] problem of "k-way balanced graph partitioning". Each of k machines in a cluster contains a distinct portion of the graph, known as a partition. Partitions are likely connected by some number of edges, in which case corresponding machines will also contain copies of the connecting elements. Figure 1.2 presents an example of such a partitioning.

To understand why scaling out effectively is challenging for graph based systems, consider Fig. 1.2 and recall that most workloads of graph operations require large numbers of edge traversals, regardless of whether they are offline or online in nature. When a traversal is required over an edge which connects two partitions, intuitively one or more network requests are required; we refer to these as inter-partition traversals (*ipt*). As network latency is orders of magnitude higher and more variable than main-memory, or even disk access [22], the more *ipt* an operation causes, the slower its running time.

In order to scale out graph based systems effectively, sophisticated graph partitioners have been proposed [17, 52, 58, 76, 110, 117, 125], the details of which we discuss in Chapter 3. The fundamental goal of these systems is to arrange the vertices and edges of a graph across distributed partitions in such a way that operations may be executed efficiently, with minimal network latency.

1.2 Motivating problem

Despite the challenging nature of the problem, aforementioned existing partitioners are often able to produce good results: graph partitionings which support the efficient execution of operations whilst distributed. This is especially common when given graphs subject to workloads of offline analytical operations. However, these systems are rarely an effective or appropriate choice when given large graphs used in an online context. This lack of graph partitioning systems which are well optimised for online data-management serves as the ongoing motivation for this work.

Broadly speaking, the difference in effectiveness is due to the fact that existing partitioners make trade offs which are acceptable to an offline system, but incompatible with the requirements of an online one. For instance, most graph partitioners are slow over large graphs, taking on the order of tens of minutes, or even hours [117]. Additionally, most partitioners are not incremental in nature, i.e. if a sub-graph is added to an existing partitioning, the partitioner must be re-executed over the entire graph, not just the new portion.

For offline analytics applications, where the data is often static or updated in infrequent batches, graphs can be partitioned **once** prior to an operation. The execution time of an analytics operation may exceed that of even a complex partitioner by several orders of magnitude [125]. Therefore, if a quality graph partitioning reduces the operation's execution by (e.g.) 30% then the upfront cost of the partitioner is worthwhile.

Meanwhile, in online data-management applications graphs grow continuously, which requires repeated re-execution of expensive non-incremental partitioners. As consistent availability and performance is paramount online, such existing partitioners are rendered impractical [55].

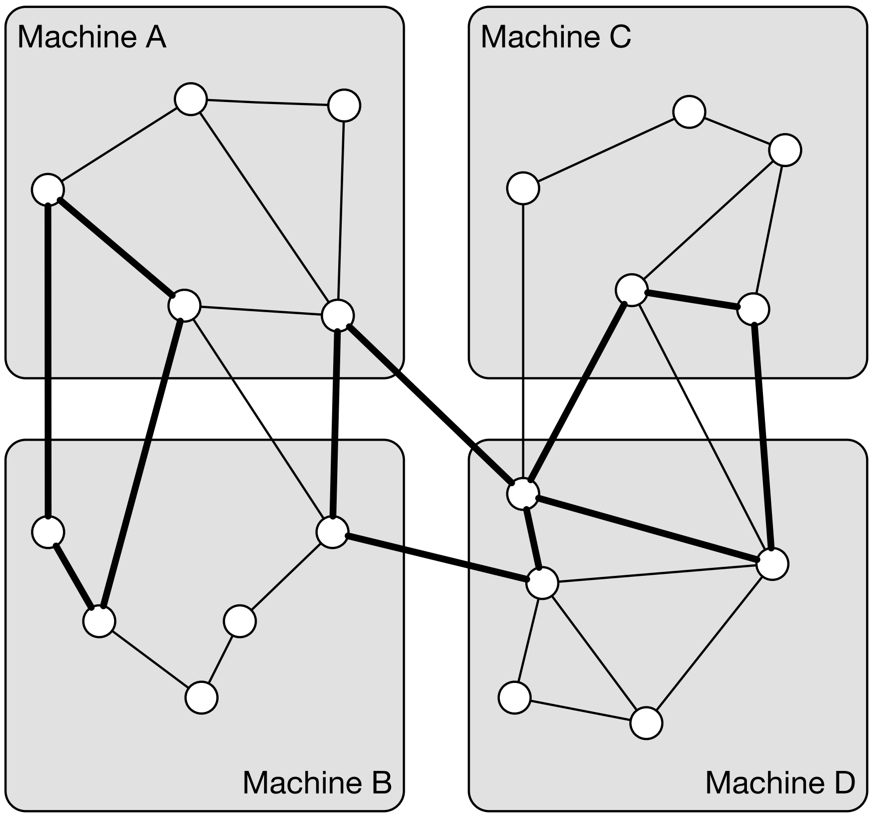


Figure 1.3: Sub-optimal partitioning w.r.t a workload Q

More importantly, even existing incremental [52, 87, 110, 117] partitioners may produce sup-optimal partitionings in an online context because they optimise for an inappropriate goal.

1.2.1 *Workload-agnostic partitioners*

Recall that when partitioning graphs to scale out effectively, our primary aim is to minimise the network latency when executing operations (Sec. 1.1). *ipt* is an ideal approximation for partitioning quality with respect to this goal, because it is essentially **equivalent** to network latency [7], whilst also being a scale-free metric.

On the other hand, **existing partitioners are largely designed to operate independent of any particular workload and therefore cannot optimise directly for *ipt*** or even network latency, which both intuitively require a workload trace to

[7]Assuming, for simplicity, a homogeneous network and a consistent number of network packets per traversal.

measure. Instead, these systems, which we call *workload agnostic*, use some other measure of graph partitioning quality as the objective function for their optimisation. The most common function used in existing partitioners [33, 58, 90, 102, 110, 117] is the number of edges which connect vertices in different partitions (a.k.a *min. edge-cut*).

It follows that the quality of graph partitionings produced by a workload agnostic system depend upon the accuracy with which that system's objective function approximates *ipt*, given a workload.

For example, min. edge-cut approximates *ipt* closely, **only** assuming a constant and uniform likelihood of traversal for each edge throughout workload execution. In other words, if every edge is equally likely to be traversed, then minimising the number of inter-partition (cut) edges minimises the number of inter-partition traversals. Such uniform distributions of edge traversals are actually common to many categories of offline analytical operation [107], including PageRank [75] and Graph Colouring. It is for this reason that workload agnostic partitioners often produce quality graph partitionings with respect to offline analytical workloads.

However, given a workload of the pattern matching queries common to online data-management applications such as GDBMS, the assumption of uniform edge traversal likelihoods is unrealistic: a query workload may traverse a limited subset of edges and edge types, which is specific to its graph patterns and subject to change over time. For example, consider Fig. 1.3. The partitioning A, B, C, D is optimal for the min. edge-cut function, but may not be optimal for the queries in a given workload Q. For example, Q only traverses highlighted edges then every query would increase *ipt* (create network requests). This explains why workload agnostic partitioners often produce poor quality partitionings with respect to online data-management workloads.

1.3 Research Aim and Contributions

In order to address its motivating problem, the high-level aim of this thesis is to:

Design, implement and evaluate techniques for producing partitionings of large graphs which are well optimised for use in an online data-management context.

Implicit in this aim, however, there are a number of more nuanced research questions:

- What properties are desirable for graph partitioning techniques intended for use online?
- How best to capture information about a given query workload?
- How to incrementally and efficiently partition a growing graph in such a way that it exhibits high quality (few *ipt*) with respect to the given workload.
- How to efficiently **re**partition an existing graph partitioning such that it exhibits the same or better quality?

In an attempt to answer these questions and thereby address its aim, this thesis presents the following concrete contributions:

1. A detailed survey on the state-of-the-art in graph partitioning, including recent, relevant or fundamental results from the literature. Existing systems are described, categorised and critically analysed. Finally, we isolate a number of properties of graph partitioners which are beneficial for addressing our motivating problem (e.g. workload-sensitive). These properties constitute a framework which may be used to consistently evaluate systems with respect to the thesis aim. This is presented in Chapter 3.
2. A compact, trie-based [69] datastructure for encoding the paths of edge traversals which occur in a graph when executing **path queries** from a given workload Q, along with their frequencies. We describe how this structure may be efficiently constructed and updated if Q evolves over time. This is presented in Chapter 4.

3. An extended version of the previous trie datastructure, which builds upon frequent sub-graph mining research [115] to encode common patterns of edge traversals over a graph when executing a workload of **pattern matching queries**. We present an efficient algorithm for constructing the structure using a probabilistic method of sub-graph isomorphism. Finally, we demonstrate how these structures may be used as a space-efficient and discriminative indices over graphs. This is presented in Chapter 5.

4. A practical system, called *TAPER*, for improving the quality of an existing graph partitioning with respect to a workload of **path queries**. Specifically, TAPER uses the original trie data structure to calculate which vertices in a partitioning are presently most likely to be the source of inter-partition traversals and iteratively relocates small numbers of them. Given a naive initial partitioning, this reduces future *ipt* by around 80%, comparable or superior to state-of-the-art workload agnostic partitioners [59], while requiring far less network communication. This is presented in Chapter 4.

5. Another practical system, *Loom*, which produces a high quality partitioning of a graph stream[8] with respect to a given workload of general **pattern matching queries** Q. Loom uses the extended trie data structure to efficiently detect sub-graphs which Q will frequently traverse together, as they arrive in the graph stream. The system then attempts to place these sub-graphs entirely within single partitions, reducing *ipt* by up to 40% relative to state-of-the-art streaming graph partitioners [110, 117]. This is presented in Chapter 5.

[8]Equivalent to an online, growing graph.

2

Preliminaries

Contents

In this chapter we provide those concepts and definitions which are depended upon or referred to throughout the remainder of this work.

2.1 Graph datastructures

A **simple graph** G, such as the one seen in figure 2.1, is usually denoted as $G = (V, E)$ where V is a set of vertices $v_1, v_2, \dots v_n$ and E is a set of pairwise relationships between these vertices, called edges $e = (v_i, v_j) \in E$. A graph's size is defined as $|E|$, its order as $|V|$.

Throughout this work we. also discuss several distinct forms of graph data, though all are specific instances of the above simple graphs.

For instance, graphs which also have labels associated with their vertices and/or edges. Such a **labelled graph** is denoted as $G = (V, E, L_v, f_v)$, where L_v is the set of vertex labels, the function f_v is a mapping of vertices to labels and so on. Note that f_v is surjective; i.e. every vertex has a label, which may be shared by several other vertices respectively.

A proper **sub-graph** G_i of G is a graph whose vertices and edges are a subset of G's vertices and edges, $G_i = (V_i, E_i)$, $V_i \in V$, $E_i \in E$.

A **path** of length k is a sub-graph in which k vertices and $k-1$ edges form an alternating sequence $(v_1, e_1, v_2, \dots, v_{k-1}, e_{k-1}, v_k)$ such that each vertex is part of no more than 2 edges.

The **neighbourhood** of a vertex $N_G(v)$ is defined as the set of all vertices in V which are *adjacent* to v. Formally, $N_G(v) = \{u \in V : (u, v) \in E\}$, where N_G is a function from a vertex to some vertex set, $V \to \mathcal{P}(V)$.

A graph **motif** is a typically small graph which occurs, with a frequency of more than some user defined threshold $\mathcal{T}$, as a sub-graph of some larger graph, or a collection of larger graphs.

A **graph stream** is defined simply as a (possibly infinite) sequence of vertices and edges which are being accumulated to a graph G, over time. Note also that when we discuss sliding windows over such graph streams, we consider them to be *fixed width.*

In other words, a sliding window of “time” t is equivalent to the t most recently added elements, rather than those which have arrived within the last time period t.Relatedly, note that an **online graph** [1] may be viewed as an *infinite* graph stream; we use the two terms interchangeably.

Finally, note that there exist three additional forms of graph data which, for simplicity, we do not consider throughout the remainder of this thesis.

Firstly, **directed graphs**, wherein edges are not simple pairs, but have *source* and *target* vertices.

Secondly, **edge-labelled graphs**, whose definition is expanded to include a set of edge labels L_e and additional surjective labelling function f_e, allowing edges to possess labels distinct from their vertices.

Thirdly, given that we do not consider directed or edge-labelled graphs, we also do not consider **multi-graphs**, which allow multiple edges to exist between the same two vertices. The reason for this is that having multiple, undirected, unlabelled edges between two vertices is intuitively *identical* to having just a single edge.

2.1.1 Graph stream orderings

Clearly, when working with graph streams it is important to consider not only the graph elements (data), but also the order in which elements appear in a stream. Throughout this thesis, we consider the following commonplace orderings:

- **Random** ordering is computed by randomly permuting the existing ordering of a graph’s elements.
- **Breadth-first** ordering is computed by performing a bread-first traversal across the connected components of a graph. If a graph contains several connected components then these are selected in random order.
- **Depth-first** ordering is computed by performing a depth-first traversal across the connected components of a graph. Again, if a graph contains several such components, they are selected in random order.

[1]Sometimes referred to as a dynamic or growing graph.

One more important ordering to consider, particularly for a dynamic graph or a static snapshot of a dynamic graph, is the order in which its elements are created. We refer to this ordering as **stochastic**. Unfortunately, the information necessary to derive a stochastic ordering is not available for most publicly accessible datasets. Therefore, throughout this work we consider random ordering to be an imperfect proxy for stochastic as many graphs may be viewed as growing at least pseudo-randomly.

2.2 Graph operations

The various algorithms and other operations which may be executed over a graph can be broadly separated into one of two previously mentioned (Ch. 1) categories: offline analytical operations, and online data-management operations.

Offline analytical operations are often designed in a vertex-centric fashion and executed using *bulk synchronous parallel* (BSP) systems, such as Google's Pregel [75]. In such BSP systems a graph operation is performed in a number of supersteps.

These operations are expressed as functions executed for each vertex, where the vertex contains information about itself and its neighbours. At the start of each system superstep, a vertex will receive messages sent from its neighbours during the previous superstep. The operation's function will then execute, updating the vertex's stored information and sending messages on to its neighbours. Additionally, during a superstep a vertex may *vote to halt*, rendering itself inactive. When all vertices in a graph are inactive the operation terminates.

As messages may be serialised and supersteps between partitions synchronised, these operations can execute over distributed graphs. The scalability and the relative ease of programming has made this vertex-centric pattern popular for graph processing systems [78].

There are many examples of offline analytical operations besides the PageRank algorithm [75] previously mentioned (Ch. 1). These include, e.g, computing minimum spanning trees [41] or graph *matching* algorithms [2], which derive sub-graphs containing only those edges not adjacent to one another and are commonly used in graph partitioners.

Three common features shared by the majority of such operations is that they a) require the entire graph as input; b) take on the order of minutes to hours to complete; and c) are typically executed as one-off events or at large regular intervals (e.g. weeks). This is distinct from **Online data-management operations**, which typically complete on the order of milliseconds to seconds, consider only a very small subset of a graph and are executed continuously in large numbers.

Online operations are typically executed in systems with soft real-time constraints like graph database management systems (GDBMS) such as Neo4j [86]. Pattern matching queries, which we highlight earlier (Ch. 1), are one of the most common examples.

We consider a **pattern matching query** as defined in terms of sub-graph isomorphism. Given a pattern graph $q = (V_q, E_q)$ and a host graph G , a query should return R: a set of sub-graphs of in G. For each returned sub-graph $R_i = (V_{R_i}, E_{R_i})$ there should exist a bijective function f such that: *a)* for every vertex $v \in V_{R_i}$, there exists a corresponding vertex $f(v) \in V_q$; *b)* for every edge $(v_1, v_2) \in E_{R_i}$, there exists a corresponding edge $(f(v_1), f(v_2)) \in E_q$; and *c)* for every vertex $v \in R_i$, the labels match those of the corresponding vertices in q, $l(v) = l(f(v))$. As an example, consider Figure 2.1, which presents a host graph G, along with three queries (pattern graphs) q_1, q_2 andq_3. Given the query q_2, a result R is returned, containing two sub-graphs $\{(1,2),(2,3)\}$ and $\{(6,2),(2,3)\}$ in G.

Although pattern matching queries may be described in terms of sub-graph isomorphism, they are rarely implemented solely in those terms as the problem is known to be NP-Complete [43] and practical verification algorithms [79, 119] are expensive [99]. Instead modern pattern matching query engines adopt what is known as a *filter-verify* approach [32, 49, 86]. In the filter step, a graph index structure [60] to look up *candidate* vertices or sub-graphs, which may form part or all of a pattern match. Subsequently, in the verify step, the candidates and their local neighbourhoods are traversed, edge by edge, to detect any exact matches. Intuitively, the number of traversals which occur when executing a pattern matching query depend upon the number of candidates returned by a filter step and the average degree of the graph around each candidate.

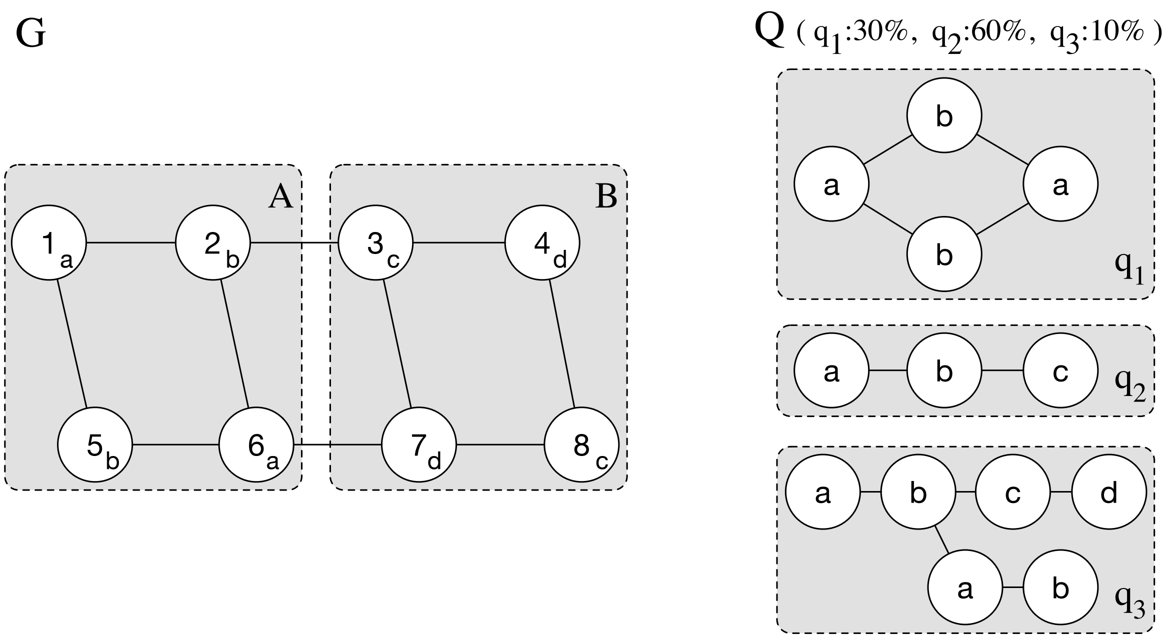


Figure 2.1: Example graph G with query workload $\mathcal{Q}$

Throughout this work we consider partitioning strategies for large graphs which account for particular workloads of the above pattern matching queries. Formally, we consider a **query workload** as a simple set of tuples $Q = \{(q_1, n_1) \ldots (q_h, n_h)\}$, where n_i is the relative frequency of each query q_i in Q.

Note that in this work we do not consider pattern matching queries as defined in terms of graph *homomorphism* [32]. This is primarily because homomorphism does not imply an exact match between a pattern graph q and a host graph G, but also because the same choice (i.e. opting for isomorphism) is taken by many widely used graph query languages [32, 40, 49].

Furthermore, for simplicity, we do not consider queries which perform negative pattern matching, i.e. vertex a must be adjacent to vertex b ($a - b$), but **not** c. However, all techniques presented throughout this thesis naturally apply to queries which include negative matches because the process of executing them is identical to that of executing queries with only positive matches. Consider again q_2 from Fig. 2.1: to verify whether the vertex 2 does, *or does not* have a c labelled neighbour, an execution engine must still traverse all neighbours of vertex 2.

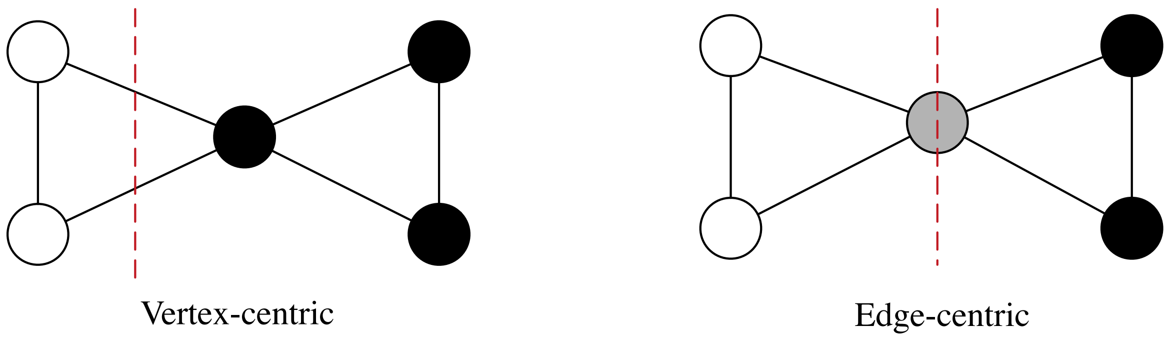


Figure 2.2: Vertex vs Edge centric partitionings

2.3 Graph partitions

A k-way **graph partitioning** $P_k(G)$ may be thought of as a view over a graph G, wherein G is separated into a set of k sub-graphs. These graph partitionings are typically defined in one of two ways: vertex-centric or edge-centric.

A vertex-centric graph partitioning is defined as a disjoint family of sets of vertices $P_k(G) = \{V_1, V_2, \ldots, V_k\}$. Each set V_i, together with its edges E_i (where $e_i \in E_i$, $e_i = (v_i, v_j)$, and $v_i, v_j \subseteq V_i$), is referred to as a *partition* S_i. A partition forms a proper sub-graph of G such that $S_i = (V_i, E_i)$, $V_i \subseteq V$ and $E_i \subseteq E$.

An edge-centric partitioning is similar, though defined as a disjoint family of sets of *edges*. Note that, in a vertex-centric partitioning[2], vertices are unique to single partitions whilst edges may be shared between two. Meanwhile, in an edge-centric partitioning, edges are unique whilst vertices may be shared between two or more partitions. Figure 2.2 provides a simple example of the difference between these two definitions.

The focus of this thesis is largely upon producing and improving those partitions which are vertex-centric.

[2]Assuming no replication

2.4 Partitioning Quality and Objective Functions

In order to improve a graph partitioning, some consistent notion of partitioning quality is needed. Existing graph partitioners usually define this quality as one of several objective functions, i.e. some measure, calculated over an entire graph, which must be maximised or minimised.

The most common such measure is the previously mentioned *min. edge-cut* (Sec. 1.2). Edge-cut is the number of edges which connect vertices in different partitions, formally: $|E_{cut}|$ where $e \in E_{cut}$, $e = (v_i, v_j)$, $v_i \in V_A$, $v_j \in V_B$ and $A \neq B$. By minimising the number of these inter-partition edges, systems **somewhat** reduce the network communication cost for a broad range of analyses, including many BSP operations and sub-graph pattern matching. Besides min. edge-cut, there are other metrics which may be used as objective functions for graph partitioners, including *communication volume* [76] and *partition stability* [24].

The communication volume of a vertex v refers to the number of distinct partitions adjacent to v, i.e. the number of partitions which contain neighbours of v but not v itself. Communication volume partitioners minimise this metric for all the vertices $v \in V$. This is similar to, but distinct from min. edge-cut partitioners: communication volume does not account for multiple edges between a vertex v and neighbours in a single partition. Communication volume partitioners have become increasingly popular for application to graphs over which min. edge-cut systems do not achieve good results, such as power-law graphs [76].

Partiton stability, first introduced by Delvenne et al. [24], is another measure of partitioning quality, defined in terms of network flow and random walks over a graph. The full definition of stability is left “in situ” in chapter 4, where it is required for context.

Note that, by default, the measures above are not able capture the quality of a graph partitioning *with respect to a specific query workload*. As a result they are unsuitable, both as objective functions for the techniques we present in this work and as a means by which to evaluate their impact.

Some measures may be modified, making them *workload-sensitive*. Indeed, we propose a modified version of stability for this purpose (Sec. 4.2.2). Technically, if graph edges are given weights corresponding to the frequency with which they are traversed by a query workload, then even min. edge-cut [3] partitionings are high quality with respect to that workload. However, in an online system such modified metrics are expensive to calculate and impractical to update [55].

Regardless of the objective functions they employ, when evaluating the graph partitioning quality achieved by different algorithms, throughout this thesis we primarily consider the number of inter-partition traversals (*ipt*) which occur when executing a given workload Q over $P_k(G)$. The reasons for this are twofold. Firstly, as we have argued (Sec. 1.2), low *ipt* is equivalent to the true goal of graph partitioners in the context of data processing systems: reduced network latency [4]. Secondly, unlike network latency, *ipt* is a scale-free measure, independent of complex implementation details.

2.5 Partitioning Hardness

Note that, in addition to the metrics above, quality graph partitionings must be approximately **balanced** [5]. In the absence of a balance requirement, optimising for quality metrics leads to work being distributed unevenly between partitions (and physical hardware), which is inefficient. As a pathological example, consider that a single partition containing all vertices and edges (i.e. an unpartitioned graph) is guaranteed not to cut any edges an is therefore optimal w.r.t imbalanced min. edge-cut. **k-way balanced graph partitioning**, as it is formally called, is known to be NP-Hard.

Andreev and Racke [3] demonstrate that, if perfect partition balance is required, there exists no constant-time approximation for partitioning general graphs. They also present an algorithm which is able to offer an improved approximation of $O(\log^2 n)$ in the event that the balance constraint is relaxed, e.g. one partition is permitted to be 30% larger than another. However, this approximation, along with others like it [13] is too slow and expensive to be used for large graphs.

[3]Now *min. edge-weight-cut*

[4]Assuming, for simplicity, a homogeneous network and a consistent number of network packets per traversal.

[5]Where no one partition is **significantly** bigger or smaller than any other.

3

Related work

Contents

Summary

This chapter seeks to provide context for the contributions presented in our thesis by examining both the relevant background material and more recent related works.

k-way balanced graph partitioning is clearly of practical importance to any application with large amounts of graph structured data. As a result, despite the fact that the problem is known to be NP-Hard [3] and available approximation algorithms too expensive [13], various practical solutions have been proposed using heuristics [18, 52, 58, 76, 90, 110, 117, 125].

In the sections that follow we survey these partitioning solutions and assign them to one of five potentially overlapping categories: Local (Sec. 3.1), Global (Sec. 3.2), Distributed (Sec. 3.3), Streaming (Sec. 3.4) and Workload-sensitive (Sec. 3.5). Graph partitioning has been the subject of significant research over many years, which we do not attempt to comprehensively review. Instead, in this chapter we highlight recent, relevant or fundamental results. Particularly close attention is paid to solutions which are either well suited to online use[1], or workload-sensitive, as these may partially address the motivations for this work (Ch. 1). We refer the reader to [5, 12, 13, 106] for further general material.

Finally, in section (Sec. 3.6), we identify eight key properties for graph partitioners, particularly those which are workload-sensitive. These properties are used as a framework for comparing the graph partitioners presented throughout the chapter, highlighting deficiencies in existing research and providing context for our own efforts. This framework will also be used to evaluate techniques presented in later chapters (Ch. 4 and 5).

3.1 Local graph re-partitioners

The category of local graph partitioners includes any partitioner which makes use of *local search*, which is also known as iterative vertex swapping or local refinement. Simply put, local search aims to improve an existing graph partitioning by swapping

[1]As opposed to offline, analytical use.

vertices between partitions in order to minimise some objection function, usually min. edge cut. Local graph partitioners vary in how they select which vertices to swap and which partitions they will consider sending vertices between (e.g. adjacent partitions, or highly imbalanced partitions).

In their work to reduce the number of object references between memory pages during program execution, Kernighan and Lin [61] propose the classic example of a local search algorithm. Indeed perhaps the first example of a graph partitioner in general. Their key intuition was as follows: given two partitions forming halves of a balanced graph bisection $P_2(G) = \{V_1, V_2\}$, there exist subsets of vertices $A \subset V_1, B \subset V_2$ which may be swapped between partitions to produce an arrangement that is globally optimal for some objective function (usually min. edge-cut).

The Kernighan-Lin algorithm (*KL*) operates in iterations, selecting vertex sets to swap which will result in the greatest improvement (which we call objective function *gain*, or simply *gain*). Within each iteration, the algorithm considers every vertex v_i from a partition (say V_1), then calculates the potential gain when swapping v_i with each vertex v_j from partition V_2. The pair (v_i, v_j) with the highest gain is marked for swapping (i.e. v_i, v_j are added to A, B respectively) and the next vertex in $V_1 \setminus v_1$ is considered with each vertex in $V_2 \setminus v_2$. Note that when considering swapping the neighbours of vertices already marked in this iteration, the potential objective gain for those neighbours is calculated as if marked vertices *have already been swapped*. When every vertex has been considered, the sets A, B are swapped between partitions and the next iteration may begin. Iterations continue until the total gain for all suggested swaps is ≤ 0.

There are two main issues with the *KL* algorithm as a graph partitioner. The first is that it is limited to improving the partitioning quality of graph bisections, rather than k-way partitionings. The second is that it is highly expensive, with a single iteration of the algorithm having the complexity $O(n^2 \log n)$, remembering that $n = |V|$.

In order to address such issues with the *KL* algorithm, several improvements have been proposed. Perhaps most significantly, Fiduccia and Mattheyses [33] present a modified algorithm (*KL/FM*) which is significantly less expensive. The iteration complexity for *KL/FM* is $O(m)$, nothing that $m = |E|$ and that the upper bound [2]for m is $(n-1)^2$.

There are two major differences between the *KL/FM* and *KL* algorithms. Firstly, the vertex swapping between bisections is asymmetric, i.e. vertices are marked for transfer individually, rather than as a pair with a vertex from the other partition. This means that for each vertex considered for swapping, it is not necessary to consider every vertex from the other partition. Secondly, Fiduccia and Mattheyses use a datastructure called a bucket queue [80] for efficiently updating neighbour objective function gain after marking vertices for swapping.

Despite the improvement that the *KL/FM* algorithm represents, it shares *KL*'s original limitation of being application only to graph bisections. However there are extensions of local search techniques which generalise to improving k-way partitionings [58, 98, 132]. Notably, Karypis and Kumar [58]propose an algorithm they call Greedy Refinement (*GR*) as part of their work on the well know global partitioner METIS, which we discuss shortly.

For efficiency, *GR* moves only boundary vertices [3] between partitions, considering all such vertices in random order during each iteration. For each vertex v, *GR* orders the partitions to which v is adjacent by the potential gain of moving v to them, subject to some balance constraints. If a move does not satisfy chosen balance constraints, then progressively less beneficial destination partitions are considered. Once a destination partition has been chosen, the move is immediately performed and the next random vertex considered. An iteration of *GR* terminates when more than some threshold number of vertex moves have been performed without any positive objective function gain as a result.

Unfortunately, despite some ability to climb out of local optimisation minima, the quality of partitionings produced by the local search algorithms above is strongly dependent on the quality of the initial partitionings they receive. They remain important to consider however, for two reasons. Firstly, they are often integrated as part of a

[2]In the rare case of a strongly connected simple graph with an edge from each vertex to every other vertex.

[3]Vertices with neighbours in ≥ 1 external partitions.

global graph partitioner [51, 58, 101] such as with Greedy Refinement and the aforementioned METIS [58]. Secondly, as the name local search implies, the information required for each vertex migration is local to that vertex. This allows asymmetric methods to perform very little or no global coordination between partitions. As a result, such methods are effectively applied in distributed settings [98, 120, 131], where inter-partition coordination is costly.

3.2 Global graph partitioners

In general, global graph partitioners [9, 26, 28, 29, 50, 51, 58, 64, 82, 90, 102] refers to those partitioners which, unlike their local counterparts, take an entire unpartitioned graph as input. By default, such partitioners are also executed within the confines of a single machine.

They are the most commonly used family of techniques, likely due to their effectiveness: global graph partitioners produce some of the highest quality partitionings of any techniques we consider. Indeed Karypis and Kumar's METIS [58] is considered the de-facto gold standard for partitioning quality [76]. However, it is also likely that the popularity of such techniques is at least partially due to their being relatively simple to use and available in a number of robust software packages [4] [5] [6].

As we alluded to previously in Chapter 1, however, simple (undistributed) global graph partitioners are not without their drawbacks, particularly in an online setting. Firstly, they are highly resource intensive [120] and are typically performed ahead of offline analytical workloads. Additionally, they require an entire graph to be available *a priori* as input, and may therefore require periodic re-execution, i.e. given a dynamic graph following a series of graph updates, which is impractical online [55].

Secondly, like local graph partitioners, the vast majority of global partitioning techniques [33, 58, 90, 102, 110, 117] optimise for the min. edge cut objective function. This renders them workload-agnostic: assuming uniform and constant usage of a graph by a workload. As a result the partitionings they produce, whilst effective at reducing

[4] The METIS [58] family of graph partitioning software: `http://bit.ly/1tqUcSQ`
[5] The Scotch [90] graph partitioning software: `http://bit.ly/2r2mbfI`
[6] The KaHiP [102] graph partitioning software: `http://bit.ly/2raQSj2`

the runtime of distributed analytical jobs (e.g. Pagerank), are sub-optimal for other types of workload, such as sub-graph pattern matching queries, which are common in an online graph data-management setting.

Note that although global graph partitioners are broadly similar in terms of advantages, disadvantages and behaviour, they may differ significantly in the method of their implementation. For example, some systems [26, 82, 90] use the natural representation of a graph as a network, and derive partitions in terms of breadth-first traversals and diffusion. Other systems [9, 28, 29, 51], known as *spectral* partitioners, treat graphs as matrices and derive partitions using linear algebra.

Finally, some global partitioning systems [50, 58, 64, 90, 102] apply an existing technique (spectral, diffusion-based or otherwise), but over compressed versions of a graph. These systems, which are called *multilevel*, are arguably the most effective global graph partitioners in terms of performance, scalability and partitioning quality.

3.2.1 Spectral techniques

Spectral graph partitioning was first proposed by Donath and Hoffman [29] for computing bisections of a graph G with respect to min. edge-cut. Firstly the Laplacian matrix L_G of G is computed by subtracting G's adjacency matrix A_G from its degree matrix D_G, $L_G = D_G - A_G$. Secondly, the eigenvector associated with the second smallest eigenvalue of L_G is computed. The algorithm relies upon the intuition that this eigenvector, called the Fiedler vector [34], contains an integer value for each vertex, which corresponds to its connectedness in the graph. Using this value as an ordering, the algorithm then divides the vertices of G around the median, into two sets of equal size. These sets represent a bisection which is good with respect to min. edge-cut. Note that this algorithm generalises to producing k-way partitionings of graphs through recursively bisecting generated partitions[7]. All other spectral partitioning techniques [9, 28, 51] extend this core algorithm.

These extensions usually aim to improve performance as, in practice, the Fielder vector is approximated using Lanczos algorithm [67] which is highly computationally expensive for large graphs. For instance, Hendrickson and Leland [51] propose a method

[7]Provided k is a power of 2.

for computing graph partitionings where $k > 2$ *without* recursive application , thereby avoiding computing the Fiedler vector more than once. Furthermore, Barnard and Simon [9] propose a multilevel spectral method. Whilst the algorithm is structurally similar to those we discuss in section 3.2.3, Barnard and Simon do not compute and refine a graph partitioning over compressed versions of a graph. Instead, they compute and refine an approximation of a Fiedler vector over compressed versions of a given graph, thereby substantially reducing its computational cost. This Fiedler vector is subsequently used to compute a bisection over the uncompressed graph.

3.2.2 Diffusions techniques

Besides spectral techniques, there are a number of global graph partitioners which employ random walks or breadth-first traversals in order to derive partitions [26, 82, 90]. These systems are broadly referred to as *Diffusion*-based and implement some variation on the following simple algorithm:

Firstly, for a k-way partitioning, k seed nodes are selected, evenly distributed throughout a graph's structure. Secondly, random walks or traversals are performed throughout the graph, originating from these seeds. Each vertex may only be traversed once. Once all vertices have been traversed, each vertex belongs to the same partition as its traversal's seed. This procedure is often employed iteratively, with new seeds being selected each round [26, 82]. Diekmann et al. [26] were the first to propose this modification, referring to is as the *Bubble framework*. Meyerhenke et al. [82] extend the Bubble framework, generalising it to graphs with variable edge weights and improving its performance by introducing a random walk mechanism which only operates over small (local) areas of a graph.

Whilst Diffusion and traversal based partitioning techniques can yield high quality results, they are also somewhat computationally intensive. Variants of the Bubble framework have a worst case complexity of $O(km)$, where m is equal to the number of vertices in a graph and k the desired number of partitions. Additionally, naive implementations of the Bubble framework can lead to highly imbalanced partitions [103], though this limitation is usually addressed through the use of additional heuristics, these may add to the overall computational complexity of a scheme.

Pellegrini et al. also employ diffusion-based techniques in their popular graph partitioning tool *Scotch* [90]. However, *Scotch* is an example of the multilevel partitioners we discuss in the following section. In other words, in order to ameliorate the complexity of diffusion-based partitioning, they only apply their technique to compressed versions of a graph and even then, only to vertices near predicted partition boundaries.

3.2.3 Multilevel application

As mentioned at the start of this section, some of the most effective global graph partitioning systems are known as multilevel [50, 58, 64, 90, 102]. Proposed in its current form by Hendrickson and Leland [50], multilevel partitioning works in three stages: coarsening, partitioning and uncoarsening.

The Coarsening stage: a succession of recursively compressed graphs is computed, tracking exactly how the graph was compressed at each step.

The Partitioning stage: the coarsening stage continues until the most compressed form of the graph is small enough that an initial partitioning may be trivially produced using an existing technique (e.g. spectral partitioning).

The Uncoarsening stage: using the knowledge of how each compressed graph was produced from the previous one, the initial partitioning is then "projected" back onto the original graph, using a local technique (e.g. KL/FM [33]) to improve the partitioning after each step. Figure 3.1 presents an example of multilevel partitioning over a small graph.

The various multilevel techniques which exist differ in how they implement the above three stages. Consider the canonical example of a multilevel partitioner: the aforementioned METIS [58].

In the coarsening stage METIS compresses a graph G by computing *maximal egde matchings* [50][8]. An edge matching is defined as a set of edges E_M from G, such that no two edges in E_M are incident upon the same vertex. Given such a matching, a

[8]Also referred to as a maximal independent edge set.

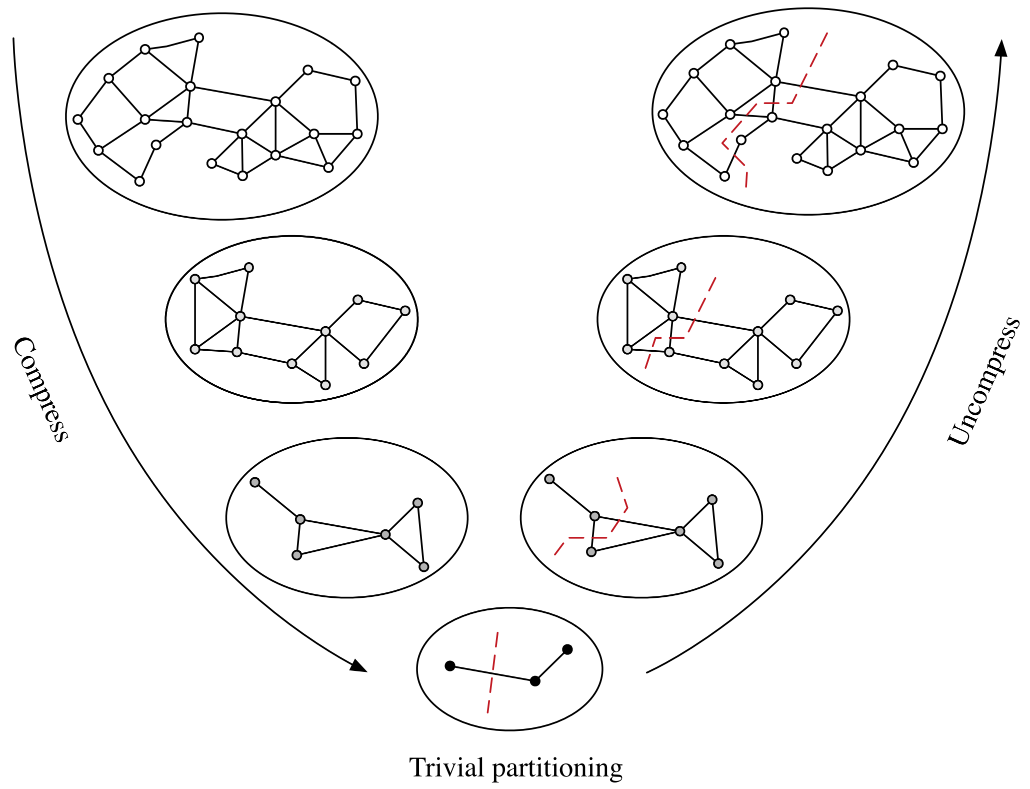


Figure 3.1: The partitioning pattern common to multilevel apparaoches. Inspired by similar figures in [58, 102]

single level of compressed graph is computed by combining vertices connected by an edge $e \in E_M$, treating each pair as a single "multi" vertex. Compression is performed using these matchings rather than, for example, arbitrarily combining vertices as it prevents any one "multi" vertex from containing many more elements than another. As a result, a balanced partitioning of the compressed graph will correspond to a balanced partitioning of the original graph.

Next, when computing an initial partitioning for the compressed graph G_M, METIS uses a technique based upon spectral recursive bisection. Finally, in the uncoarsening stage of partitioning, METIS uses the Greedy Refinement local algorithm (Sec. 3.1) to move "multi" vertices between partitions, improving the partitioning after each step of match/combine compression has been reversed.

On the other hand, consider alternative multilevel partitioners [64, 102]. Korosec and Silc propose MACA [64], which uses an ant-colony optimisation technique for both initially partitioning the most compressed graph, and for improving the partitioning after each step of the uncoarsening stage. This is interesting because ant-colony optimisation techniques are highly parallelisable; a fact Tashkova et al. exploit with their work in Distributed MACA [112] (DMACA). Meanwhile, Sanders and Schulz [102] propose an edge rating function which prioritises edges which are close *w.r.t algebraic distance* [15][9] when computing edge matchings in the coarsening phase. As a result of this edge rating function, good partitionings of compressed graphs correspond to good partitionings of their uncompressed counterparts, even more closely than with other matching techniques.

In general, the three key advantages to a multilevel approach to global graph partitioning may be summarised as follows: Firstly, as matchings and other forms of graph compression are relatively inexpensive, it is possible to compress large graphs before applying an initial partitioning technique which is highly effective, but would be be impractically expensive at the graph's original scale. Secondly, the local partitioning techniques applied during the uncoarsening stage will perform well, given that the movement of a vertex in the compressed graph corresponds to the movement of several vertices in the original graph and that each compressed graph . Finally, due to the initial partitioning and repeated improvements, the input to each step of uncoarsening will be of high quality; as mentioned (Sec. 3.1) this significantly increases the effectiveness of local partitioning techniques performing the improving.

As a result of these advantages, multilevel partitioners are the most performant and effective of all global graph partitioners. Despite this, they still share many of the disadvantages, such as optimising for an objective function and requiring an entire graph as input. This means they too are not suited for graphs which may grow, or be subject to a changing workload; i.e. they are unsuitable for application to an online graph data-management setting. Most importantly, however, the scalability of such undistributed multilevel partitioners remains fundamentally limited by the resources of their host hardware and so struggle to partition graphs with a more than a few tens of millions of vertices and edges [117].

[9]Edges which are part of strongly connected clusters in a graph will have low algebraic distance (are close to one another).

3.3 Distributed graph partitioners

In order to scale to graphs which do not fit within the memory of a single machine, global graph partitioners (i.e. those which require the entire graph *a priori*) must be modified to operate when distributed: executed by multiple machines communicating via a network.

As mentioned in Chapter 1, in recent years there have been an increasing number of applications making use of such large amounts of graph data, such as social networking [47, 122], search engines [11] and genome analysis [10, 121]. As a result, several distributed graph partitioning algorithms have been proposed, often as extensions to existing systems, e.g. PT-Scotch [18], ParMETIS [59] and ParHIP [83] and DMACA [112].

These distributed extensions [18, 59] are implemented along broadly similar lines to their single-machine counterparts. In other words, the first stage of the algorithm is to recursively coarsen/compress a graph, this time by computing a distributed maximal edge matching. Intuitively, computing whether a given inter-partition edge forms part of a matching requires coordination between the machines to which the edge's two vertices belong. Typically, this coordination happens iteratively, with each partition sending messages to it's neighbours, proposing and rejecting potential matched edges in rounds. To reduce the communication overhead caused by these messages, progressively smaller graphs are gathered (or "folded") onto smaller numbers of machines until, at the end of the coarsening stage, the smallest graph resides on a single machine.

Once, the smallest graph is contained on a single machine, the initial partitioning stage takes place as normal. Subsequently, during the uncoarsening stage, the graph partitioning is refined using a local refinement technique to move vertices between (distributed) partitions. In addition, after each step of uncoarsening, the graph partitioning is "unfolded" back onto a larger number of machines. Fig. 3.2 presents an example of this distributed multilevel partitioning process.

Meyerhenke et al's recent ParHIP [83] follows a similar pattern, though employs a novel parallel label-propagation based graph clustering technique to coarsen the input graph, instead of computing an edge matching. This label-propagation technique is

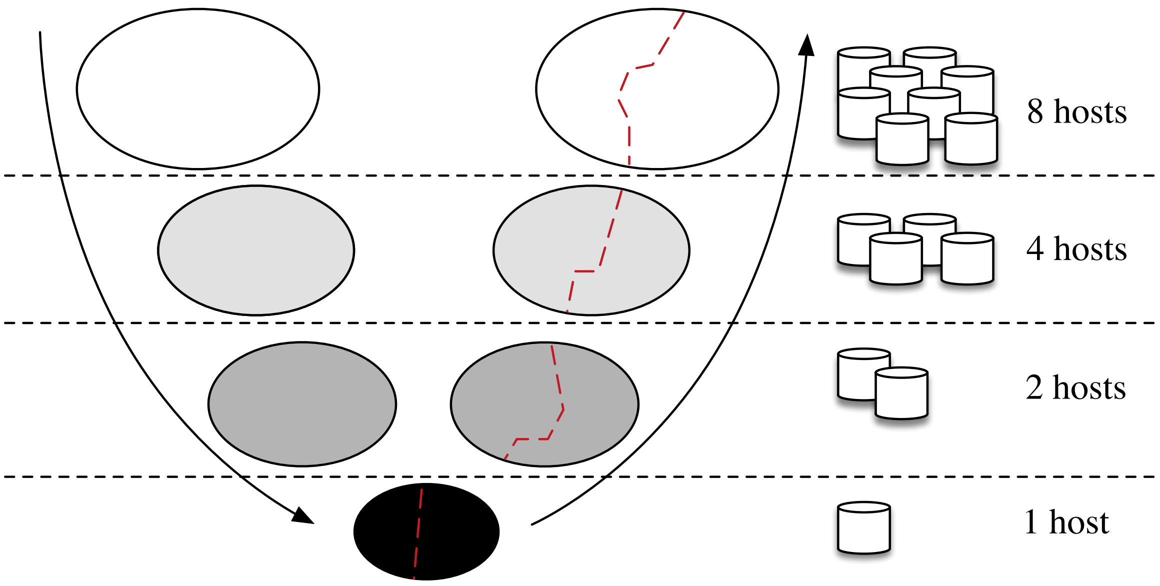


Figure 3.2: Example of “folding” in distributed mulilevel graph partitioners.

highly effective, producing significantly smaller graphs at the coarsest level than other edge matching techniques, and allowing ParHIP to partition larger graphs than is possible with, e.g. ParMETIS.

Another interesting example of a distributed graph partitioner which uses label-propagation is JA-BE-JA [98]. JA-BE-JA is actually an example of a *local-search* technique (Sec. 3.1), using vertex labels to denote partition assignments, and iteratively swapping labels between neighbouring vertices. Additionally, JA-BE-JA uses simulated annealing to escape local optimisation minima better than other local search techniques.

Finally, Margo and Seltzer’s work on Sheep [76] is worthy of consideration. The Sheep algorithm efficiently creates an elimination tree [92] from a distributed graph using a map-reduce procedure, then partitions the tree and subsequently translates it into a partitioning of the original graph. Unlike the other distributed partitioners we consider, which optimise for min. edge-cut, Sheep optimises for the min. communication volume objective function. In other words, it minimises the number of different partitions in which a given vertex v has neighbours. This metric has been shown to be more effective than min. edge-cut for producing partitionings of certain graphs, such as graphs whose degrees are distributed according to a power-law.

In general distributed partitioners exhibit significantly improved scalability, successfully partitioning graphs with billions of edges [83]. Furthermore, because such algorithms require input graphs to be spread between host machines, they are theoretically applicable to **re**-partitioning a graph in situ, unlike their single-machine counterparts.

Despite addressing the scalability concerns of global graph partitioners, the distributed systems above are not without their drawbacks, however. Principle among these is communication cost: all of the above systems incur significant communication overhead, both when computing a new partitioning[10] and when migrating vertices and edges between partitions.

Whilst some techniques do exist to minimise the number of inter-partition vertex swaps which occur during a repartitioning [104], these inevitably trade off against the quality of the final partitioning. Additionally, such techniques have been shown to have only a limited effect [132].

As a result, despite distributed partitioners' theoretical applicability to the task, graph analytical frameworks often employ them only as initial, highly scalable, partitioning step, rather than for repeated repartitioning [62].

Finally, note that distributed partitioners share the other drawbacks of normal global partitioners, besides scalability. In particular, they remain unsuitable for dynamically growing graphs and agnostic to changes in the workloads being applied to them.

3.4 Streaming graph partitioning

Like the distributed systems above, streaming graph partitioners [52, 87, 110, 117] have been proposed to address the scalability and performance issues of global partitioners. The strict streaming model considers each element of a graph stream[11] as soon as it arrives, efficiently assigning it to a partition. Additionally, streaming partitioners do not perform any refinement of portions of the graph already considered and partitioned. In other words, they will not employ local refinement techniques to move vertices to other partitions, nor perform any sort of global introspection such as in spectral partitioning. This model has two key advantages. Firstly, the memory usage of streaming partitioners is both low and independent of the size of the graph being

[10]In spite of improvements due to "folding" heuristics [18, 59].
[11]This may be either vertices or edges.

partitioned, allowing them to scale to to very large graphs (e.g. billions of elements). Secondly, although graph streams are often created by reading static data serially from disk, streaming partitioners may trivially be applied to dynamically growing graphs by treating each new edge or update as an element in the stream. This is in stark contrast to the mentioned difficulties with applying distributed partitioners to such graphs.

The canonical examples of streaming graph partitioners are Linear Deterministic Greedy (LDG) [110] and Fennel [117], which both make partition assignment decisions on the basis of inexpensive heuristics considering the local neighbourhood of each vertex [12] at the time it arrives. LDG was proposed by Stanton and Kliot in a survey of various streaming graph partitioning heuristics, where it was the most effective technique considered. It assigns vertices to the partitions where they have the most existing neighbours, but penalises that number of neighbours for each partition by how full it is, maintaining balance. Later, Tsourakakis et al. propose Fennel, which interpolates between the LDG and another heuristic [95], which amounts to assigning vertices to the partitions where they have the *fewest non-neighbours*. Fennel produces partitionings of higher quality than LDG *w.r.t.* min. edge-cut, in similar runtime, though at the cost of slightly worse partitioning balance.

Despite offering unlimited scalability and up to an order of magnitude speedup vs global and distributed techniques [117], streaming partitioners have their drawbacks as well. By relying upon the available local neighbourhood information for a vertex v *at the time* v *is added* to the graph stream, such techniques render themselves sensitive to the graph stream's ordering. For example, consider one of the common stream orderings outlined in chapter 2.1.1, *breadth-first* (BFS). Given a BFS ordering, a graph stream maintains a high degree of locality: vertices which are connected will appear close together within the stream. As a result, for each new vertex v which arrives in a BFS stream, the partitioner has likely already placed many of its neighbours and can make an effective decision about where to place v such that it is in the same partitions as the largest number of them.

[12]All neighbouring vertices and adjacent edges

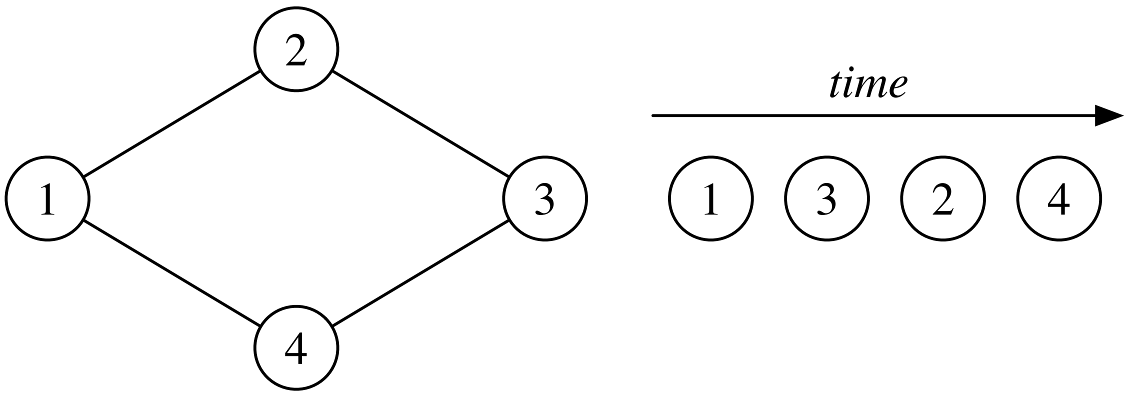


Figure 3.3: Trivial example of an adversarial graph stream ordering.

One the other hand it is simple to come up with an ordering for a graph stream which deprives a streaming partitioner of almost all neighbourhood information for each arriving vertex. Consider the 2-way partitioning of a diamond graph in Fig. 3.3, streamed in the order $(1, 3, 2, 4)$. Given no neighbours for the first half of vertices received, a naive partitioner might greedily place them in a single partition which, intuitively, causes a final balanced partitioning with the maximum edge cut: $|E|$. This is referred to as an *adversarial* ordering. Whilst truly adversarial graph stream orders are uncommon, *random* graph stream orderings have been shown to exhibit somewhat poor locality [110] and occur frequently in practice. Indeed we consider the order in which dynamic graphs grow to be **approximately** random.

As a result of this sensitivity, streaming algorithms generally produce partitionings of lower quality than their non-streaming counterparts but with much improved performance. Note an important caveat to this performance improvement: some streaming algorithms (e.g. Fennel) are difficult to parallelise [87] and therefore must read a graph purely sequentially. Thus, if a large static graph is available in advance, the loading of which dominates partitioner runtime, distributed techniques may exhibit better performance [76].

3.4.1 Re-streaming

Some systems [52, 87] attempt to strike a balance between the performance of a streaming graph partitioner and the quality of a non-streaming one by dropping the "one-pass" requirement of the strict streaming model.

For example, Huang et al's Leopard [52] repeatedly considers placed vertices for reassignment to other partitions later. Specifically, Leopard notes that when a new vertex v' arrives in a stream and is assigned to partition V_i, an adjacent vertex v previously assigned to partition V_j may no longer be in the optimal partition to cut as few edges as possible. Therefore, Leopard *re-streams* vertices such as v after their neighbourhood has changed, reapplying the original partitioning heuristic. This improves partitioning quality over time, especially for dynamic graphs, but at the cost of performance.

To understand the performance impact of this technique, consider a pathological case where each new vertex v' in the stream causes a cascade of re-streaming operations amongst its neighbours, then their neighbours, and so on. Huang et al. control the trade off between quality and performance by only actually re-streaming a subset of the neighbours of a new vertex v'. This subset is selected using a heuristic which is biased against neighbours with high degree. Therefore, if a neighbour v has many other neighbours already, it will be re-streamed less frequently.

Interestingly, Leopard can also consider replicating vertices as they arrive in a graph stream. Replication may be used to ensure data availability and to further optimise for the min. edge-cut objective function, as discussed below (Sec. 3.5.3). Throughout this review, however, we consider Leopard without replication as that is how it is evaluated [52].

Rather than focus on partitioning dynamically growing graphs, Nishimura et al. [87] focus on a common use-case whereby an entire graph is partitioned using a streaming heuristic, repeatedly and at regular intervals. For example, a social network may wish to partition its data each night before performing some routine analytical computation.

They propose partitioning in a number of iterations, processing the entire graph stream in each iteration using a modified streaming heuristic which accounts for the location of each vertex's neighbours after previous iterations, even if they have not yet arrived in the current stream. This provides significant partitioning quality improvements over strictly "one-pass" streaming systems, whilst preserving their constant memory usage and therefore scalability. Furthermore, Nishimura et al's evaluation demonstrates that their approach achieves partitioning quality similar to that of global partitioners such as METIS, in around 10 iterations. Whilst impressive, this approach intuitvely has one major drawback: performance. It takes n times longer than, e.g., LDG to partition of a graph from scratch, where n is the desired number of restreaming iterations.

Finally, note that besides their other advantages and disadvantages, all the (re)streaming partitioners above[13] are workload agnostic, optimising for the min. edge-cut objective function. As a result, they share the implied disadvantages with their non-streaming counterparts.

3.5 Workload sensitive partitioning

Thus far in this chapter we have discussed only those systems which optimise for various objective functions (Sec. 2.4), most commonly min. edge-cut. As mentioned, optimising for this function renders the output of such partitioners agnostic to the traversal patterns of specific workloads, cutting each edge of a graph with equal likelihood. A workload which focuses on small and shifting subsets of graph edges, e.g. a stream of sub-graph pattern matching queries, risks causing many expensive inter-partition traversals.

It is technically true that many min. edge-cut partitioners [33, 58, 90, 102, 110, 117] may account for edges with varied weights, essentially optimising for min. edge-**weight**-cut. If these weights are set to reflect the frequency with which a workload traverses each particular edge, then a min. edge-weight-cut partitioning may be somewhat workload-sensitive. However, for many classes of graph workload it is not possible, or at least highly expensive to track the number of traversals of each edge in order to derive their weights [129]. Additionally, partitionings optimised for this modified function are still agnostic to workloads which may change over time. A min. edge-weight-cut system would have to continuously update edge weights **and** repeatedly repartition a graph in order to be workload sensitive.

Recently, several specialised systems [20, 89, 91, 97, 107, 125] attempt to account for specific workloads more efficiently when partitioning graphs. However, these often focus on offline, analytical workloads [107, 125] or are replication [84, 91, 96, 120, 129] schemes, rather than graph partitioners. There are some systems [20, 89, 97] which

[13]Both streaming and restreaming.

are focused on online, transactional workloads, though these are largely confined to partitioning traditional relational databases, rather than large graphs.

Note that workload-sensitive systems may be local, global, distributed or streaming, but are discussed separately here as they most closely relate to the motivating problem for our thesis (Sec. 1.2).

3.5.1 Offline workloads

Some partitioners, such as LogGP [125], CatchW [107] and Mizan [62], are focused on improving graph analytical workloads designed for the *bulk synchronous parallel* (BSP) model of computation (Sec. 2.2).

Remember that in the BSP model a graph processing job is performed in a number of supersteps, synchronised between partitions. CatchW examines several common categories of graph analytical workload. It proposes techniques for predicting the set of edges likely to be traversed in the next superstep, given the set of edges traversed in the previous ones, along with the category of workload being executed. Subsequently, between supersteps, CatchW moves a small number of these predicted edges to other partitions, minimising future inter-partition communication.

LogGP uses a similar log of activity from previous supersteps of an analytical job to construct a hypergraph[14] where vertices which are frequently accessed together are connected. LogGP then uses an existing global technique to partition this hypergraph, using the result to suggest placement of graph vertices and reducing the job's execution time in future.

Finally, Mizan also collects a log of inter-partition traversals (messages) from each superstep, along with their response times. Rather than construct and partition a hypergraph, however, Mizan uses a local partitioning technique. It identifies a small number of vertices which may be causing workload imbalance and iteratively swaps them with other partitions before the next superstep.

Though not strictly workload-sensitive, another collection of partitioning techniques optimised for BSP workloads are those which are *environment-sensitive* [126, 131,

[14] A hypergraph is a graph in which single edges may connect more than 2 vertices.

132]. These systems contend that, whilst reducing the number of inter-partition edges traversed by an analytical job effectively reduces its communication time, it may not be the most effective way to minimise total job execution time. This is due to the fact that a cluster of machines may be heterogeneous[15], i.e. machines may have different levels of computing power, or be virtualised across various physical hosts, affording different levels of inter-machine communication.

For example, Xu et al. 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.

3.5.2 Online workloads

Relational database management systems (RDBMS), whilst not strictly storing graphs, are most often subject to online, transactional workloads. Some works propose partitioning schemes for relational data which are sensitive to such workloads [20, 89, 97]. Interestingly, this often involves representing a relational query workload as a graph [20, 97].

For instance, *Schism* [20] captures a query workload sample ahead of time, modelling it as a hypergraph where edges correspond to sets of records which are involved in the same transaction. This graph is then partitioned using existing global techniques (specifically, METIS) to achieve a min. edge-cut. When 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.

Whilst this technique does reduce inter-partition communication and therefore improve query performance, it does have some limitations. Firstly, a hypergraph representation of the relational query workload can be very large and therefore expensive or impractical to partition with METIS, impacting system scalability [97]. Secondly, because the workload-graph partitioning is "one-off", Schism is only as workload-sensitive as

[15]Not to be confused the heterogeneity of labelled graphs, which we define in chapter 4.

min. edge-weight-cut graph partitioners. In other words, it assumes that a workload is available *a priori* and does not change over time.

In *SWORD*, Quamar et al. [97] build upon the ideas presented in Schism while addressing some limitations. For instance, they use a compressed representation of the workload graph and perform incremental re-partitioning to improve the partitioning's scalability to workload changes.

Despite these improvements, SWORD does not quite address the motivating problem of this thesis (Sec. 1.2). Fundamentally, the system is focused on optimising workloads executed over relational data, which 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 sub-graph 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 pattern matching.

Other RDBMS partitioners, such as *Horticulture* [89], rely upon a function to estimate the cost of executing a sample workload over a database and subsequently explore a large space of possible candidate partitionings. We do not consider such partitioners in this work for two reasons. Firstly, these cost functions, whilst effective, are expensive and so executed offline and ahead of time. Partitioning systems which employ them therefore suffer from the same drawbacks as Schism: assuming a consistent workload. Secondly, many cost functions, such as the "Large-Neighbourhood Search" employed by Horticulture, depend heavily upon the relational data model and are not naturally applicable to graph databases.

In the domain of RDF stores, Peng et al. [91] use frequent sub-graph mining ahead of time to select a set of patterns common to a provided SPARQL query workload. They then propose partitioning strategies which ensure that any data matching one of these frequent patterns is allocated wholly within a single partition, thus reducing average query response time. They also keep track of what frequent pattern matches have been assigned to which partitions, using this index to produce distributed query plans during workload execution.

Note that, because sub-graphs may form part of multiple frequent pattern matches, they are replicated across many partitions. Whilst replication is an important approach to improving the effective quality of a graph partitioning, it is not without its drawbacks, increased storage and data synchronisation costs. We discuss these costs further in the following section.

3.5.3 Replication systems

Replication based partitioners broadly refers to those systems which produce graph partitionings where vertices, edges or larger sub-graphs exist in several partitions simultaneously. The most common application of replication in OLTP focused databases is data redundancy, i.e. ensuring that data is still accessible in the event that one or more partitions become unavailable (due to an error, maintenance etc...). However, replication may also be used to improve the quality of a graph partitioning with respect to a workload.

Consider two adjacent vertices a, b residing on separate partitions S_1, S_2. If a query engine must traverse the connecting edge in order to verify a potential match, then a costly inter-partition traversal will occur. If instead, a and b are replicated to S_2 and S_1 respectively, then no such traversal will occur; all of the information is available to the query engine within a single partition.

As mentioned, replication has two distinct costs: storage space and write performance. If you have a replication factor of three[16] then your graph will take three times as much space. Additionally, if a vertex or edge which has been replicated is subsequently updated then all the replicas must also be updated, incurring additional network communication overhead. As a result of these costs, graph partitioning systems must be selective about which elements they replicate and how many replicas they produce. This is most often achieved by considering characteristics of the workload due to be executed over a graph [84, 91, 96, 129]. In other words, replication based systems are often **workload-sensitive**.

Peng et al. [91], mentioned previously, consider micro-level properties of a workload (namely the sub-graphs common to many pattern queries in a workload) when deciding

[16]Maintain 2 replicas of each vertex, on average.

what to replicate. However, other replication systems [84, 91, 129] focus on macro-level, statistical workload properties.

For instance, Pujol et al. [96] present a "social partitioning and replication middleware" (SPAR). In social networking, and other applications which produce very large scale graph data, the *de facto* approach is to use a random, hash-based partitioning. Whilst this approach minimises the complexity of generating graph partitionings, it also maximises the *edges cut*, and subsequently network traffic. In order to combat this issue, Pujol et al. propose ensuring that all single-hop neighbours of a vertex are replicated locally to that vertex's parent partition; They call this ensuring *local semantics*. For most classes of query common to social networking applications (e.g. *list all followers of user* x), the relevant information is a single traversal away from x. In the presence of these broad, shallow traversals, local semantics may provide a reduction in network communication overhead of up to 200%. However, this performance increase is dependent upon the domain specific knowledge that the majority of traversals are of length 1, and would be substantially less effective in the case of other, more general workloads. Furthermore, despite Pujol et al's assertion that data is likely already replicated due to a requirement for data redundancy, the scheme incurs significant space complexity costs, with replication factors as high as 8 [84].

In a later work, Mondal and Deshpande [84] propose an extension to the local semantics concept which significantly reduces the replication overhead incurred. They introduce a new measure, called the *fairness criterion*, which dictates that for each vertex v, at least a fraction τ of its neighbours must be replicated locally (to v's partition), where τ is a threshold value less than or equal to 1. Thus, the local semantics of Pujol et al. can be seen as a special case of the fairness criterion: where $\tau = 1$.

In addition, Mondald and Deshpande's system monitors the read/write frequencies of vertices and use these to decide **which** fraction τ of a vertex's neighbours it is most beneficial to replicate at any given time.

Mondal and Deshpande also propose novel techniques for reducing the cost of keeping replicated nodes upto date throughout a distributed system. Typically, systems seeking to synchronise distributed data must choose between *push-on-change* (active) and *pull-on-demand* (passive) semantics. The former is more expensive in terms of overall network communication, but the latter incurs more query-time latency. In their system, Mondal and Deshpande adopt a mixed approach, selecting active or passive updates based on the same vertex read/write frequencies they collect to inform replication decisions.

Overall, the replication techniques presented by Mondal and Deshpande are effective, reducing the number of messages required to keep replicas upto date by 30% whilst only increasing the average read query latency by 2-3 *ms*. However, as mentioned, the system stores histograms of read/write frequencies for **each vertex**. Not only is there significant overhead associated with storing this metadata; in some database management systems (DBMS) it may be non-trivial to collect.

Finally, Yang et al. propose *Sedge* [129], a system which analyses online query workloads as they're being executed in order to detect "hotspots": cross partition clusters of vertices that are being frequently traversed. Sedge dynamically replicates these hotspots, thereby reducing *ipt* and network communication whilst the hotspot is active. Whilst highly effective at dealing with unbalanced query workloads over time, the system focuses solely upon the replication of vertices and edges using temporary secondary partitions. It does not improve upon the initial partitioning of the graph, which is not guaranteed to be workload sensitive. This can result in Sedge's replication mechanism doing more work than is necessary over time, adversely affecting overall system performance.

3.6 Comparing partitioner systems

In preceding sections we have surveyed various existing approaches to the problem of balanced graph partitioning. The survey identifies a number of flaws common to each category of approach, then discusses to what extent these flaws are minimised or eliminated by the state of the art.

In this section we identify eight distinct properties of state of the art graph partitioners which allow them to overcome the flaws of their predecessors, with respect to our motivating problem (Sec. 1.2).

Recall that this problem is: optimising large online graph partitionings for workloads of transactional queries executed online, e.g. against a graph database management system (GBMS). The properties identified below are those which we believe are beneficial in addressing this said problem. They are not intended as a comprehensive list of the useful properties of graph partitioners. Indeed, note that each property naturally implies a dual, such as *workload sensitive* vs. *workload agnostic* partitioners. A partitioner which supports a given property essentially **cannot** support its potentially useful dual. For example, *streaming* is identified as a property; whilst normally beneficial for performance, as noted (Sec. 3.4), streaming partitioners should be avoided when partitioning quality is of the highest importance. Instead a partitioner with the dual property *global* should be preferred.

3.6.1 System properties

Streaming (S) partitioners treat graphs as ordered sequences of vertices and edges, immediately assigning each to a partition using cheap heuristics, thus trading off partitioning quality against performance and scalability (Sec. 3.4).

Distributed (DT) partitioners partition graphs which have already been partitioned, albeit potentially naively. The cluster of machines which hosts the graph communicate to compute reassignments of elements amongst partitions, optimising for some objective function. The ability to add machines to the cluster means that these partitioners typically have excellent scalability, though at the cost of some network overhead and therefore performance (Sec. 3.3).

Scalable (SC) partitioners are those partitioners which may take advantage of "infinite" resources (i.e. are distributed) or whose resource usage scales independently to the size of graph being partitioned (i.e. streaming). These partitioners may be applied to graphs with billions of vertices and edges.

Dynamic (DY) partitioners are those systems which are capable of partitioning graphs which grow over time. Streaming partitioners are naturally applicable to dynamic graphs, but other systems may also introduce schemes to handle the updates to a greater or lesser extent. These schemes usually come at the expense of some computational overhead.

Replication Free (RF) partitioners are those which maintain the disjoint property for produced partitionings, i.e. each element of the graph exists on a single partition. Replication may be used in partitioners to effectively optimise for min. edge-cut (or some other objective function) by ensuring that vertices and edges are stored locally to the various partitions where they may be needed, reducing the number of expensive inter-partition traversals (*ipt*). *Replication free* partitioners evidently lose these benefits, however they also avoid the storage overhead incurred in storing replicas as well as the network overhead associated with keeping replicas consistent in the presence of updates.

Workload Sensitive (WS) graph partitioners are those which produce partitionings optimised for a specific workload.

Note that there are markedly different approaches to graph partitioning which may be said to share the *workload sensitive* property. As a result, the subsequent two properties may be used to distinguish between workload-sensitive systems.

Online Workload (OW) capable partitioners are those which are able to continuously optimise a graph partitioning for a given workload, the contents or relative frequencies of which may change over time.

Transactional Workload (TX) capable graph partitioners are those which optimise for a workload typically consisting of many queries traversing highly localised areas of a graph and either performing updates or returning small result sets. In GDBMS these queries correspond to sub-graph pattern matching operations.

3.6.2 Suitability of existing systems to online workload-sensitive partitioning.

Table 3.1 shows which of the graph partitioners we have surveyed support which of the properties listed above. Each cell contains a **Y** to indicate support, a **P** to indicate

Table 3.1: Properties of Graph partitioners

System	**S**	**D**	**SC**	**DY**	**RF**	**WS**	**OW**	**TX**
KL/FM [33]		Y		P	Y			
Spectral [9, 28, 51]					Y			
DMACA [112]		Y			Y			
PT-Scotch [18]		Y	Y		Y			
ParMETIS [59]		Y	Y		Y			
ParHiP [83]		Y	Y		Y			
Sheep [76]		Y	Y		Y			
LDG [110]	Y		Y	Y	Y			
Fennel [117]	Y		Y	Y	Y			
Nishimura et al. [87]	Y		P	P	Y			
Leopard [52]	Y		Y	Y	Y			
Peng et al. [91]			P			Y		Y
CatchW [107]		Y	Y			Y	P	
Mizan [62]		Y	Y			Y	P	
LogGP [125]		Y	Y			Y	P	
Sedge [129]		Y	Y			Y	Y	Y
Mondal & Deshpande [84]		Y				Y	P	Y
Schism [20]						Y		Y
Sword [97]			P	P		Y	P	Y

partial support, or is blank to indicate no support. In order to improve readability, some systems in Table 3.1 are grouped together, e.g. Spectral graph partitioners [9, 28, 51] sharing the same properties, or elided altogether, e.g. the non-parallel versions of global partitioners (KaHiP [102], METIS [58], Scotch [90] and MACA [64]).

Note that Table 3.1 indicates partial support (**P**) of the online workload (**OW**) property by several systems. This is because workload-sensitive (**WS**) partitioners may be more or less capable of handing a dynamic workload, even without explicitly tracking workload changes over time. For example, the **WS** partitioners *least* suited to handling a dynamic workload are those like Schism [20], which account for specific vertices and edges traversed, *a priori*, instead of general traversal patterns [91, 97] or broad statistics [84].

Table 3.1 would seem to indicate that systems such as Fennel [117], Leopard [52] or Sedge [129] are the most appropriate choice for partitioning online graphs. However, all these systems have flaws highlighted previously in this survey, leaving significant room for innovation.

As a replication system, Sedge is unable to produce initial partitionings. Instead, Yang et al. employ a workload-sensitive method to create temporary partitions and replicate graph elements currently involved in large numbers of queries, improving read performance. This means the amount of replication Sedge performs, and therefore the overhead it incurs, is sensitive to the quality of the initial partitioning with respect to the workload (Sec. 3.5.3). As Table 3.1 demonstrates, there are currently no scalable **and** workload-sensitive partitioners with which to produce such an initial partitioning.

As streaming partitioners, both Fennel and Leopard are able to scale to partitioning large dynamic graphs. However, as discussed (Sec. 3.4), the "one-pass" approach of Fennel produces partitionings of around $40-60\%$ lower quality than those produced by Leopard. Meanwhile, the "re-streaming" approach of Leopard is up to **44** times slower than Fennel, unless it sacrifices some partitioning quality . Additionally, both systems optimise for the min. edge-cut objective function (i.e. are not workload-sensitive) and therefore may not produce truly high-quality partitionings with respect to certain query workloads (Sec. 3.5).

In general, Table 3.1, along with the above issues, demonstrates a clear need for graph partitioning techniques which are scalable, applicable to online graphs and sensitive to online workloads. It is such techniques which we propose and evaluate throughout subsequent chapters.

4

Query-aware partition-enhancement with TAPER

Contents

Summary

This chapter presents *TAPER*: our graph repartitioning algorithm which is both workload-sensitive and suitable for use in online settings. *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 workload of path queries. 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 8 iterations.

4.1 Introduction

Recall that one of the advantages of representing application data as a labelled graph is the many useful types of operation which are rendered efficient and/or simple to express (Ch. 1). One such operation type is the path query, employed specifically for learning analytics [93], querying scientific workflow provenance [1] and fuzzy search of semantic web documents [94].

In this chapter we address the problem of efficiently, incrementally and **continuously improving the performance of these path queries over k-way partitionings** of large, heterogeneous, labelled graphs.

Note that we consider the heterogeneity of a graph as the absolute size of its **vertex** label set, or $|L_v|$ given the definition of a labelled graph $G = (V, E, L_v, f_v)$ from section 2.1. For example, a social graph with $L_v = \{Person, Post\}$ is more heterogeneous than a web graph with $L_v = \{Url\}$.

In the Introduction chapter (Ch. 1), we present how balanced k-way graph partitioning may be used to effectively horizontally scale graph based applications (Sec. 1.1).

However, when subsequently outlining the motivating problem for this thesis (Sec. 1.2), we argued that existing systems for producing such graph partitionings [17, 20, 33, 51, 52, 58, 59, 76, 83, 87, 90, 97, 102, 107, 110, 117, 125] are unsuitable for graphs subject to online, data-management workloads. This argument is further supported by Yang et al.[129], who experimentally demonstrate that online graph queries (including path queries) cause large numbers of expensive inter-partitioned traversals (*ipt*) over graphs partitioned by traditional methods.

The literature survey in Chapter 3 outlined eight distinct properties whose lack may render graph partitioners unsuitable for addressing our motivating problem (Sec. 3.6). However, in this chapter, we focus on three in particular.

Firstly, many partitioners [51, 52, 58, 90, 102, 110, 117] cannot operate whilst distributed across multiple machines (i.e. no support for property **D**), instead requiring that an entire graph is collected to, or streamed through a single point. Additionally, all graph partitioning algorithms will require re-execution in an online data-management setting, i.e. after a series of graph updates or workload changes. As graphs are likely only partitioned at all if they are very large, this implies a huge network communication overhead, which is certainly impractical online [55].

Secondly, most existing partitioners [17, 33, 51, 52, 58, 59, 59, 76, 83, 87, 102, 110, 117] are workload-agnostic, as they optimise for the minimum number of inter-partition edges (*min. edge-cut*) or some other workload-agnostic objective function. In other words, they lack the workload-sensitive (**WS**) property. As previously mentioned, these workload-agnostic partitioners essentially assume a uniform likelihood of traversal for each edge throughout workload execution, which, for workloads of online operations such as path queries, is unrealistic (Sec. 1.2). Even applicable existing workload-sensitive graph partitioners [20, 91, 97, 107, 125] can only account for a constant likelihood of each edge's traversal and therefore lack the **OW** property: they are not sensitive to any workload changes which may occur over time.

To appreciate the importance of query-sensitive partitioning, consider the graph of Fig. 4.1. The partitioning $\{A, B\}$ is optimal following a balanced min edge-cut approach [58], but it may not be optimal when query patterns are taken into account.

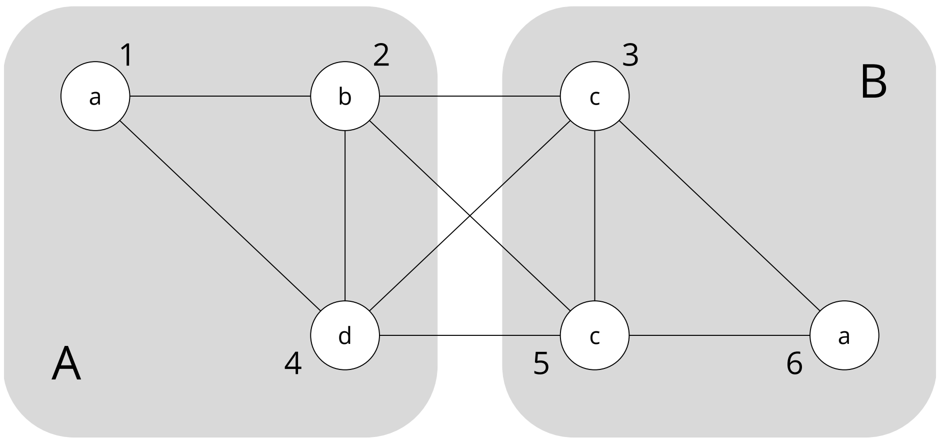


Figure 4.1: Illustrative example graph

Following common practice, we express queries using a Regular Path Queries [8, 81] (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. 4.1.

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 $(3, 2), (5, 4)$ require traversing a partition boundary, although this new partitioning is not optimal with respect to min edge-cut. As concluded in the previous chapter, mature research on the creation and maintenance of online workload-sensitive partitionings is currently confined to relational DBMS and replication systems [20, 97, 129].

4.1.1 The TAPER re-partitioner

This chapter presents *TAPER*: a new graph repartitioning system which is sensitive to evolving query workloads (**WS** and **OW**), designed to work on distributed graphs (**D**) and not reliant on replication (**RF**). Let $\mathcal{Q} = \{(Q_1, n_1) \ldots (Q_h, n_h)\}$ 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 [58], LDG [110] or spectral recursive octasection [51].

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.,

$$P_k(G, Q) \xrightarrow{\mathcal{Q}'} P'_k(G, Q') \tag{4.1}$$

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:

$$P_k^0(G) \xrightarrow{\mathcal{Q}_1} P_k^1(G, \mathcal{Q}_1) \xrightarrow{\mathcal{Q}_2} P_k^2(G, \mathcal{Q}_2) \ldots \tag{4.2}$$

These partitionings have the property that each $P_k^i(G, \mathcal{Q}_i)$ exhibits lower extroversion than the previous $P_k^{i_1}(G, \mathcal{Q}_{i-1})$ **given** $\mathcal{Q}_i$: it is approximately optimised for that new 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 4.1, 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 (Def. 4.1) we update our data structures for $\mathcal{Q}'$ and G', recompute vertex extroversion and begin a new round of iterations. Note that as a technique which relies upon iterative swapping of vertices, *TAPER* is technically a "local graph re-partitioner", similar to others discussed in the literature survey (Sec. 3.1).

4.1.2 Contributions

The specific contributions contained within this chapter are as follows:

- Firstly, from the notion of *stability* of a partition[24] we derive a workload-sensitive operational metric of partitioning quality, expressed in terms of extroversion for each vertex. This acts as the objective function for *TAPER*'s repartitioning.
- Secondly, we describe a space-efficient encoding of the traversal probabilities for each edge in G, given $\mathcal{Q}$. We also show how these traversal probabilities can be updated following the evolution of $\mathcal{Q}$.
- Thirdly, we demonstrate how *TAPER* makes use of these structures to iteratively achieve a re-partitioning step (Def. 4.1).

We present an extensive evaluation of the *TAPER* system using both real and synthetic graphs of varying sizes, comparing its efficiency and output quality against one-off workload-agnostic partitionings obtained using the popular METIS approach[58], 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 78% quality improvement when a hash-based $P_k^0(G)$ starting point is used, and 31% 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.

4.1.3 Related Work

Although Chapter 3 presents a thorough survey of graph partitioning techniques which are related to those in this thesis, this section briefly highlights those which are similar or relevant to *TAPER* in particular.

TAPER's goal of repeatedly refining an existing partitioning is technically supported by ParMETIS[59], the parallel implementation of METIS[58], as well as by other distributed versions of global graph partitioners[18, 83] highlighted previously (Sec. 3.3). However, this process is computation and communication intensive and as such is often only used as a "one-off" step, rather than for repeated repartitioning of a graph[120].

Additionally recall that, like their undistributed global counterparts[58, 90, 102], these partitioners are workload-agnostic. Although some, including ParMETIS, may be provided with edge-weights which correspond to the traversal frequencies of individual edges by a query workload, such weights are much too expensive to collect and maintain over time (Sec. 3.5).

The work of Vaquero et al. [120] 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 in a data-management context (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. Other modern iterative swapping systems, whilst still similar to TAPER either focus on analytical workoads [62] or adapting to heterogeneity in the physical network layer of a distributed system [126, 131, 132].

4.2 Definitions

Given the previous definition of a labelled graph $G = (V, E, L_v, f_v)$, where the function $f_v : V \to L_v$ surjectively associates a label $f_v(v)$ from the given set L_v to each vertex $v \in V$, a **path-query** q over G is equivalent to a regular expression over the symbols in L_v. We use a type of Regular Path Queries (RPQ) [81], defined by the following expression language over L_v:

$$E ::= \tau \mid (E \cdot E) \mid (E + E) \mid (E \mid E) \mid E* \tag{4.3}$$

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 paths $G_i = (V_i, E_i, L_v, l)$, 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 path queries are a special case of more general sub-graph pattern matching queries, able to express only simple sequential paths of connected vertices, rather than more complex topologies such as those including branches and cycles. Whilst RPQs may be used to express sub-graph pattern matching queries, this requires either conjunctions between expressions or other extensions, such as those proposed by Barcelo et al.[8]. For simplicity, these extensions are not covered by the RPQ fragment defined by expression language (Def. 4.3), and are considered out of scope for the work presented in this chapter.

4.2.1 Stability of a graph partitioning

The broad goal of *TAPER* is to increase the quality of a k-way graph partitioning (Sec. 4.1.1). Recall that throughout this thesis, we consider the *quality* of a partitioning $P_k(G)$ to be defined as the number of inter-partition traversals (*ipt*) which occur when executing a given query workload Q, however, do not employ *ipt* directly as an objective function in partitioning algorithms (Sec. 2.4).

In the following sections we define the workload-sensitive proxy quality measure which does serve as the objective function for *TAPER*'s optimisation. For this, we extend the notion of partition *stability* [24], first introduced by Delvenne et al. in the context of multi-resolution community detection in graphs [23]; stability defines partition quality 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 (by ergodicity[1]):

$$p(V_i, t_0, t_0 + t) - p(V_i, t_0, \infty)$$

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.

4.2.2 Workload-sensitive stability

We extend stability, creating a new measure of partition quality which we will refer to as *workload-sensitive stability*. Our extensions are driven by two main requirements. Firstly, as mentioned, *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** $\mathcal{Q}$ (Def. 4.1 in Sec. 4.1.1). 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.

[1]According to the ergodic hypothesis [42] $P(S, t_0, \infty)$, after an infinite time a walker holds no memory of its initial position.

4.2.3 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_m to v_n only depends on the prior probability of being in v_m:

$$Pr(v_m \rightarrow v_n | v_i \rightarrow \ldots \rightarrow v_m) = Pr(v_m \rightarrow v_n | v_m)$$

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

$$M[m,n] = Pr(v_m \rightarrow v_n | v_m)$$

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

$$Pr(v_m \rightarrow v_n | p \rightarrow v_m) \neq Pr(v_m \rightarrow v_n | p' \rightarrow v_n)$$

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

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 $v_m \rightarrow v_n$ is now explicitly conditioned on the paths, of length up to t, which lead to v_m.

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

$$VM(t) \equiv \{ VM^{(1)}, \ldots, VM^{(t)} \} \tag{4.4}$$

where t denotes the longest query matching pattern in $\mathcal{Q}$, and $VM^{(n)}$ has dimension $1 \leq n \leq t$. We use the term **Visitor Matrix** (VM) to refer to (4.4).

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 $VM^{(2)}$.

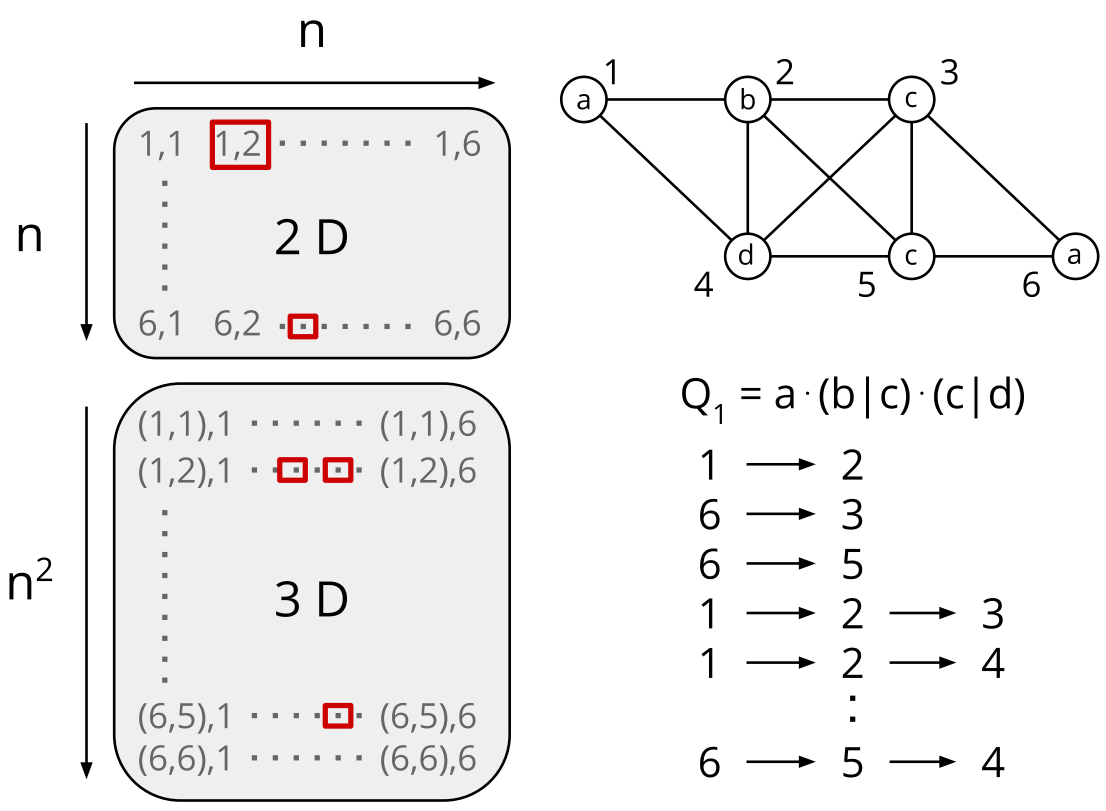


Figure 4.2: Visitor Matrix structure

Formally:

$$VM^{(1)}[i] = Pr(v_i)$$

$$VM^{(2)}[i_1, i_2] = Pr(v_{i_1} \to v_{i_2} | v_{i_1}) = M[i_1, i_2]$$

$$VM^{(n)}[i_1, .., i_n] = Pr(v_{i_{n-1}} \to v_{i_n} | v_{i_1} \to \ldots \to v_{i_{n-1}}) \tag{4.5}$$

for $2 < n \leq t$. Fig. 4.2 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. 4.1), relative to query expression Q_1. For example, path $1 \to 2 \to 3$ 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 to compute the elements of $VM(t)$ for a given query workload $\mathcal{Q}$ in Section 4.4.2.

In practice, we assume $VM(t)$ is partitioned into k sub-matrices $VM_i(t)$, one for each of k 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 Sec. 4.5.2 we present heuristics that are designed to reduce both space complexity, as well as to avoid computing some of the cells in the VM.

4.3 Enhancing a Partitioning

We exploit the VM structure, to compute a new partitioning $P(G, \mathcal{Q})$ from a partitioning $P(G)$, as in Def. 4.1. 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 $\mathcal{Q}$ 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-sensitive 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 Def. 4.2).

4.3.1 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 $\mathcal{Q}$. *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 [61] and later improved upon by Fiduccia and Mattheyes [33]. They present techniques that attempt to find sets of vertices and edges which, when moved between two halves of a graph bisection, produce an

[2]We never explicitly calculate stability, as it is an expensive global measure ($O(|V|\cdot|E|+|V|^2)$ [77]) , unsuitable for use as a cost function.

arrangement that is globally optimal for some criteria (usually min edge-cut). Karypis and Kumar [58] 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 vertex[3] 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 Section 4.5.5. Using traversal probabilities, precomputed in the visitor matrix, we identify a vertex v's *family*: those vertices likely to be the source of traversals to v. This is the clique of vertices which should accompany a swapping candidate to a new partition.

4.3.2 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 $VM_i^{(n+2)}[v_{i_1}, \ldots, v_{i_n}, v, v']$ denotes the probability of a transition from v to v', given a

[3]A vertex with neighbours in ≥ 1 external partitions.

path $p = v_{i_1} \rightarrow \ldots \rightarrow v_{i_n} \rightarrow v$ that matches a query pattern. Let $\mathit{paths}(v, V_i)$ 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 $p \in \mathit{paths}(v, V_i)$ and every destination vertex $v' \in V_i$. Formally:

$$\mathit{introversion}(v) = \frac{1}{Pr(v)} \sum_{p} (\; Pr(p) \cdot \sum_{v' \in V_i} VM_i(t)[p, v'])$$
$$\text{for all } p \in \mathit{paths}(v, V_i) \tag{4.6}$$

where for path $p = v_{i_1} \rightarrow \ldots \rightarrow v_{i_n} \rightarrow v$ of length $n + 1$, we have:

$$\begin{aligned} Pr(p) = \; & Pr(v | v_{i_1} \rightarrow \ldots \rightarrow v_{i_n}) \cdot \\ & Pr(v_{i_n} | v_{i_1} \rightarrow \ldots \rightarrow v_{i_{n-1}}) \cdot \ldots \cdot Pr(v_{i_2} | v_{i_1}) \cdot Pr(v_{i_1}) = \\ & VM_i^{(n+1)}[v_{i_1}, \ldots, v] \cdot \\ & VM_i^{(n)}[v_{i_1}, \ldots, v_{i_n}] \cdot \ldots \cdot VM_i^{(2)}[v_{i_1}, v_{i_2}] \cdot VM_i^{(1)}[v_{i_1}] \end{aligned}$$

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

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

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:

$$\mathit{extroversion}(v) = \frac{1}{Pr(v)} \sum_{p} (\; Pr(p) \cdot (1 - \sum_{v' \in V_i} VM_i(t)[p, v']))$$
$$\text{for all } p \in \mathit{paths}(v, V_i) \tag{4.7}$$

4.4 Prefix Trie encoding of query expressions

We use a prefix trie [69], 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 $p_1, \ldots, p_n$ is *connected*, such that $(p_n, p_{n-1}) \in E$, and its corresponding sequence of labels $l(p_1) \ldots l(p_n)$ 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. [70] 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 [4] in this, smaller, trie of labels. In practice, each path in the trie is an intensional 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-to-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.

[4]Note that throughout this thesis, we refer to elements of data and query graphs as vertices, elements of metadata structures like The TPSTry as nodes.

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[5]. The labels for each query in $\mathcal{Q}$ are hashes of the expressions themselves, as these are guaranteed to be unique[6]. 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):

$$str(l) = \{l\} \text{ for each } l \in L_V$$

$$str(e_1 \mid e_2) = str(e_1) \cup str(e_2)$$

$$str(e_1 \cdot e_2) = \{append(x, y) | x \in str(e_1), y \in str(e_2)\}$$

$$str(e^{\{min,max\}}) = str(e \cdot e \ldots e) \quad // \text{ between } min \text{ and } max \text{ times}$$

Example. Consider again the graph in Fig. 4.1 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. 4.3(a). The two trees are then further combined into the single prefix tree in Fig. 4.3(b), with each node labelled with the set of queries it pertains to.

Finally, consider these additional notes about the construction of the *TPSTry* data structure:

Firstly, recall that the RPQ fragment we consider in this chapter includes the Kleene star operator (Sec. 4.2), allowing for zero or more repetitions of an expression. Intuitively, expressions which include this operator map to an infinite set of arbitrarily large strings, presently impractical to encode in the *TPSTry*. Instead, we consider the repetition of an expression to have a minimum length min and a maximum length max. We contend that this is not an important limitation however, as most practical implementations of graph query languages recommend or enforce similar limits for

[5] *TPSTry* nodes may be labelled with multiple queries

[6] We use Q_i labels in examples for readability

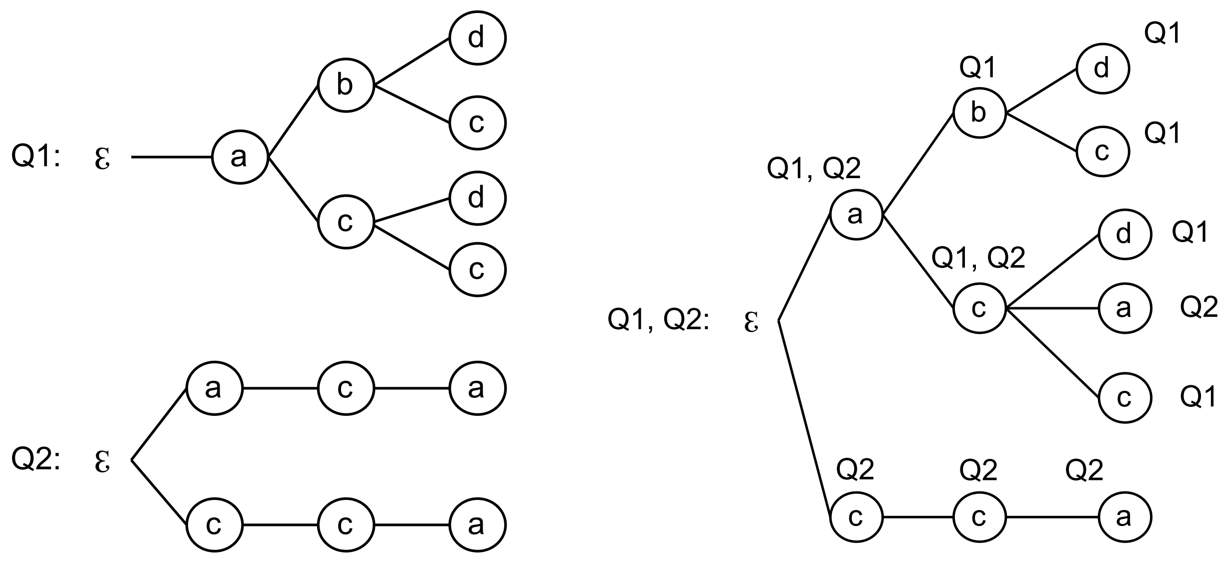


Figure 4.3: Summary trie construction from queries.

performance reasons [7, 40]. Thus, the *TPSTry* is still able to encode realistic query workloads.

Secondly, although we do not consider directed or labelled edges in this work (Sec. 2.1), the *TPSTry* is equally applicable to path queries which **do** include them. Provided that the edge directions and labels are captured as distinct symbols in query expressions they may be translated to strings using the same $str(Q)$ procedure, simply producing a *TPSTry* of greater depth.

4.4.1 Associating probabilities to trie nodes

Given a trie, such as in Fig. 4.3(b), 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. 4.3, 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 $Pr(\mathcal{E} \rightarrow a \rightarrow c)$:

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

where

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

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 η_n in the trie by the sequence of n steps $(\eta_1, \eta_2, \ldots, \eta_n)$ required to reach it from the root node, $\mathcal{E}$. A *probability label* $Pr(\eta_n)$ is then associated with each node, its value computed as follows:

$$\begin{aligned} Pr(\eta_n) = &\; Pr(\mathcal{E} \rightarrow \ldots \rightarrow \eta_n) = \\ &\sum_{Q_i \in \mathcal{Q}} Pr(\mathcal{E} \rightarrow \ldots \rightarrow \eta_n|Q_i) \cdot Pr(Q_i) \end{aligned}$$

The individual terms of the sum are conditional probabilities over the path in the trie to node η_n. As we have seen in the example, these conditional probabilities over the

$$
\begin{array}{c|cccccc}
 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\
\hline
v_1 & 0.67 & 0.33 & 0 & 0 & 0 & 0 \\
v_2 & 0 & 0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
v_6 & 0 & 0 & 0.33 & 0 & 0.33 & 0.33 \\
v_1, v_1 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_1, v_2 & 0 & 0 & 0.25 & 0.5 & 0.25 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
v_6, v_3 & 0 & 0 & 0 & 0.25 & 0.25 & 0.5 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
v_6, v_6 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

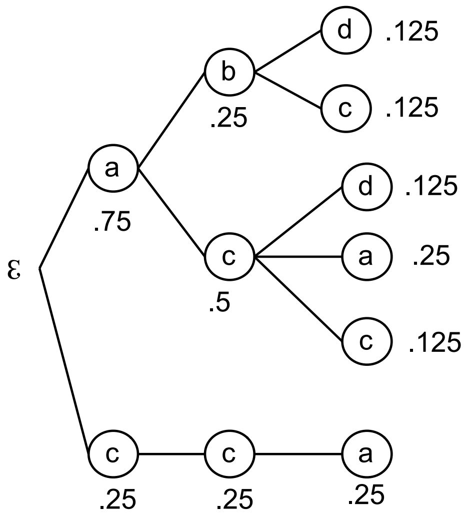


Figure 4.4: Visitor Matrix (left), TPSTry probabilities (right)

paths are computed recursively on the length n:

$$
\begin{aligned}
&Pr(\mathcal{E} \rightarrow \ldots \rightarrow \eta_{n-1} \rightarrow \eta_n | Q_i) = \\
&\qquad Pr(\eta_{n-1} \rightarrow \eta_n | \mathcal{E} \rightarrow \ldots \rightarrow \eta_{n-1}, Q_i) \cdot \\
&\qquad P(\mathcal{E} \rightarrow \ldots \rightarrow \eta_{n-1} | Q_i)
\end{aligned}
$$

4.4.2 *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 (Sec. 4.4.1). 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 $p = p_1, p_2, \ldots, p_n$, 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 $\eta_i \in \mathcal{N}$ which represent legal labels for the next vertex to be traversed, along with each label's associated probability $Pr(\eta_i)$. Subsequently, the traversal probabilities for each label are uniformly distributed amongst those neighbours of p_n which share that label. This produces a vector of traversal probabilities, one for each neighbour of the p_n. 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 of no subsequent traversal from a vertex as probability to traverse to the same vertex[7], as this is equivalent to intra-partition traversal probability.

In Section 4.2.3 we described a VM cell as containing the probability of traversing to a vertex v given some preceding sequence of traversals $p_1 \to p_2 \to \ldots \to p_{t-1}$. Formally, we compute the value of a cell $VM^{(t)}[p_1, \ldots, p_{t-1}, v]$ as

$$\begin{aligned} &Pr(p_{t-1} \to v | p_1 \to \ldots \to p_{t-1}) = \\ &\qquad Pr(l(p_{t-1}) \to l(v) | \mathcal{E} \to l(p1) \to \ldots \to l(p_{t-1})) \cdot \\ &\qquad Pr(p_t = v | l(p_t) = l(v), p_t \in N_G(p_{t-1})) \end{aligned}$$

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

Example. Given the graph in Figure 4.1, 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 $a \to b$: c and d. From the query pattern summary trie in Figure 4.4, we know that the relative frequency of c from $a \to b$ is 0.5 .

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

The relative frequency of d from $a \to b$ 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 of traversing to vertex 3 is the probability of traversing to a c labelled vertex, divided by

[7]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.

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 Figure 4.4(left), we have $VM^{(3)}[1,2,*] = (0,0,0.25,0.5,0.25,0)$.

In the previous section (Sec. 4.4.1) 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 (Sec. 4.3.1) 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.

4.5 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 summary 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.

4.5.1 Architecture

We have implemented a system prototype on top of the Tinkerpop graph processing framework [113], which allows us to use any of several popular GDBMS to store G. Though our prototype, built using the Akka framework [72], 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

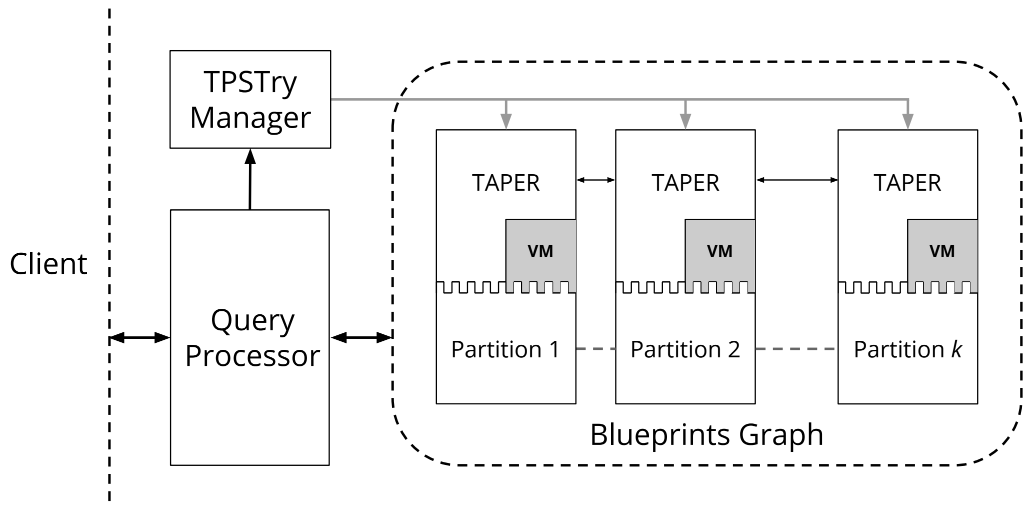


Figure 4.5: Architecture

that multiple edge-disjoint sub-graphs are treated as a single, global, graph and queried using the Gremlin query language[8]. An inter-partition traversal is detected when a Gremlin query retrieves the external neighbours of a cut vertex. Our test architecture is shown in Figure 4.5. 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.

4.5.2 Reducing the cost of the Visitor matrix

As noted in Section 4.2.3, 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.

4.5.2.1 Space complexity

Firstly, we note that large graphs are typically sparse [44] : i.e. $|E| << |V|^2$. 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

[8]The Gremlin query language: `http://bit.ly/1tqUpWk`

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 equations 4.6 & 4.7 (Sec. 4.3.2), 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 Figure 4.1. Accounting for the *TPSTry* of Figure 4.3, 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 Figure 4.4, completing this process for paths $p \in paths(v_3, V_B)$ 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.

4.5.2.2 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 though 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 (Sec. 4.4.2). 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.

4.5.3 TPSTry Implementation

TPSTry (Sec. 4.4) 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 [6] which samples the occurrences of each query within a sliding window of time t.

4.5.4 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 (Sec. 4.3.2).

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 Section 4.5.2. A simplified version of the procedure is expressed in Algoritm 1. Consider again our earlier example of vertex 3 in partition B (Fig. 4.1),

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)
```

along with the pattern summary trie in Figure 4.4 (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.

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. 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* $0.06 \cdot \frac{1}{0.5} = 0.12$.

4.5.5 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 [58] (Sec. 4.3.1). This is a two-step, symmetric process, shown in Figure 4.6: 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

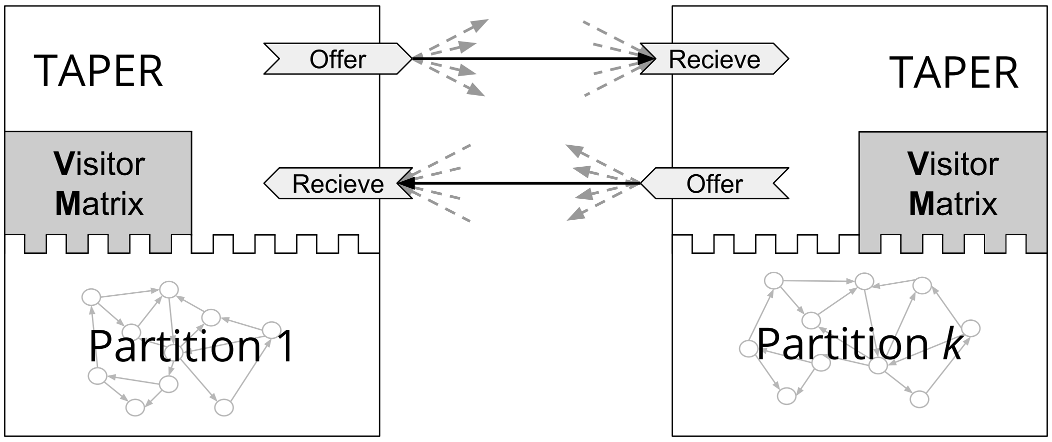


Figure 4.6: Offer/Receive algorithm in each *TAPER* instance

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-sensitive stability*. In order to avoid this we adopt another technique from (Par)METIS [59]. 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$\rightarrow$lower, lower$\rightarrow$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 sub-graph 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-sensitive 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.

4.6 Evaluation

Our evaluation aims to show how *TAPER* achieves and maintains high partitioning quality, measured as low *inter*-partition traversals (*ipt*). We present three main results. The first two on the effect of a single *TAPER* invocation given a workload snapshot $\mathcal{Q}$ (i.e. Def. 4.1, from Sec. 4.1.1): **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 [58] 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. Def. 4.2 from Sec. 4.1.1): **3)** given a workload stream $\mathcal{Q}_1, \mathcal{Q}_2, \ldots$, 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 [58] 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 [76, 117]. 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 additional, implementation dependent, metrics because, as a prototype, *TAPER* is unlikely to exhibit realistic performance. For instance, without true distributed query processing (Sec. 4.5.1) across a network, query response times are meaningless as a measure of partitioning quality.

4.6.1 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, with *TAPER* we optimise a graph partitioning for higher workload-sensitive stability (Sec. 4.2.2), corresponding to lower probability of inter-partition traversals occuring when executing a workload Q. We measure the effectiveness of this experimentally by executing snapshots of query workloads over partitioned graphs and counting (Sec. 4.5) the number of inter-partition traversals *ipt*. All algorithms, data structures and dataset pre-processing steps, including *calcVMRows* and the *TPSTry*, are publicly available[9]. All our experiments are performed on a machine with a 3.1Ghz Intel Core i7 CPU and 16GB of RAM.

4.6.1.1 Test datasets

TAPER is designed to perform best on heterogeneous graphs (Sec. 4.4). 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.

[9]The *TAPER* repository: `http://bit.ly/1W3f0eH`

The first, **MusicBrainz** [111][10], 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 [35] and compliant with the PROV data model [85]. 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 [85], PROV graphs naturally have three labels, representing the three main elements of provenance: *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.

4.6.1.2 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 pattern[11] 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.

[10]The MusicBrainz database: `http://bit.ly/1JOwlNR`
[11]Details of the workload stream are elided for space.

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.

$\mathbf{MQ_1}$ $Area \cdot Artist \cdot (Artist|Label) \cdot Area$: searches for two distinct patterns which would indicate an artist has moved away from their country of origin.

$\mathbf{MQ_2}$ $Artist \cdot Credit \cdot (Track|Recording) \cdot Credit \cdot Artist$: might be used to detect collaboration between 2 or more artists on a single track.

$\mathbf{MQ_3}$ $Artist \cdot Credit \cdot Track \cdot Medium$: 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 [25, 57]. Using these categories, we propose four query patterns typical of provenance analysis.

$\mathbf{PQ_1}$ $Entity \cdot (Entity)^{\{1,5\}} \cdot Entity$: computes the transitive closure over a data derivation relationship.

$\mathbf{PQ_2}$ $Agent \cdot Activity \cdot Entity \cdot Entity \cdot Activity \cdot Agent$: identifies pairs of agents who have collaborated as data producer/consumers pairs.

$\mathbf{PQ_3}$ $(Entity)^{\{1,5\}} \cdot Activity \cdot Entity$: returns all entities and all activities involved in the creation of a given entity.

$\mathbf{PQ_4}$ $Entity \cdot Activity \cdot (Agent)^{\{1,5\}}$: returns agents responsible for the creation of a given an entity.

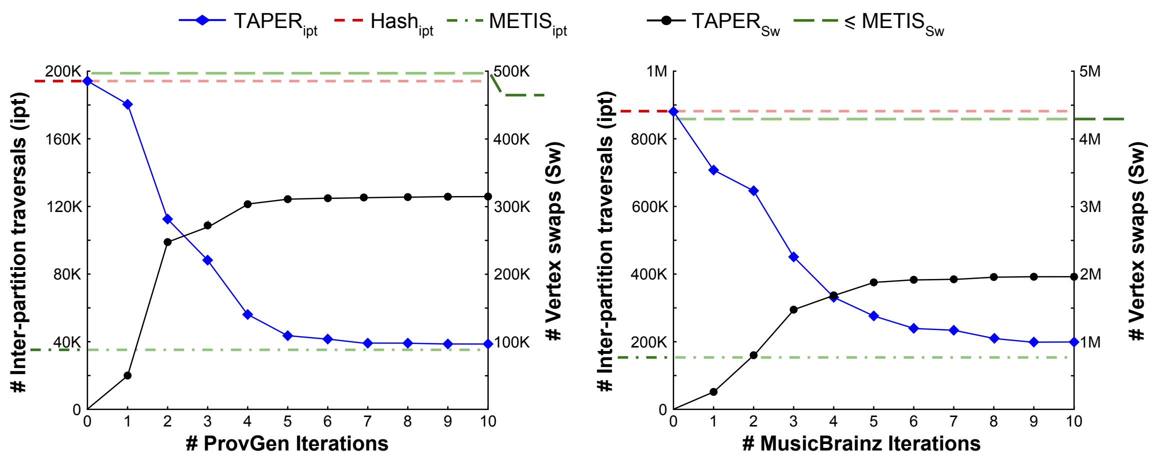


Figure 4.7: *ipt* per *TAPER* internal iteration

4.6.2 Results

4.6.2.1 Improvement over an initial hash partitioning

Figure 4.7 shows the improvement in partitioning quality which a single *TAPER* invocation achieves for each dataset, given static workload snapshots $\mathcal{Q}$ and initial hash-partitionings $P_8^0(G)$. The top dotted line bisecting the left y-axis is our baseline: the *ipt* required to execute $\mathcal{Q}$ over $P_8^0(G)$. The bottom dotted line indicates the *ipt* required to execute $\mathcal{Q}$ 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 4.7 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. 4.7(a)) require just 300k vertex swaps to produce $\sim 80\%$ enhancement.

We use a lower bound in our comparisons because, as mentioned (Sec. 4.1.3), there are multiple different implementations of METIS [58, 59] 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 [105]. Meanwhile, the number of vertex swaps caused by ParMETIS' diffusion based repartitioning approach is ~ 1.5X the edge-cut reduction produced [105].

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 (Sec. 4.3.1), we are correctly prioritising those swaps that are more effective at reducing *ipt*, given a workload snapshot $\mathcal{Q}$. 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.

4.6.2.2 Improving over other initial partitionings

Figure 4.8 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

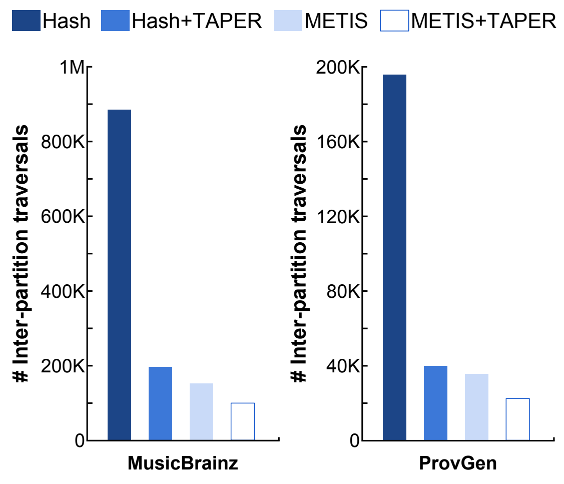


Figure 4.8: *ipt* per approach

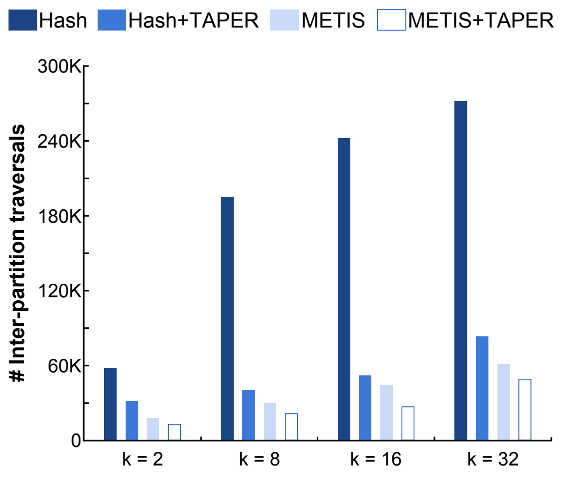


Figure 4.9: *ipt* vs *k* partitions (ProvGen)

family cliques (Sec. 4.5.5), 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 improvement. Note that improvement is not necessarily possible when METIS is given an input graph with edge-weights corresponding to traversal likelihood given $\mathcal{Q}$. 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 [129]. 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.

4.6.2.3 The effect of differing numbers of partitions

Figure 4.9 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 $\mathcal{Q}$ over several distinct partitionings of the ProvGen graph: four for each of the partitioning approaches we consider, with k values

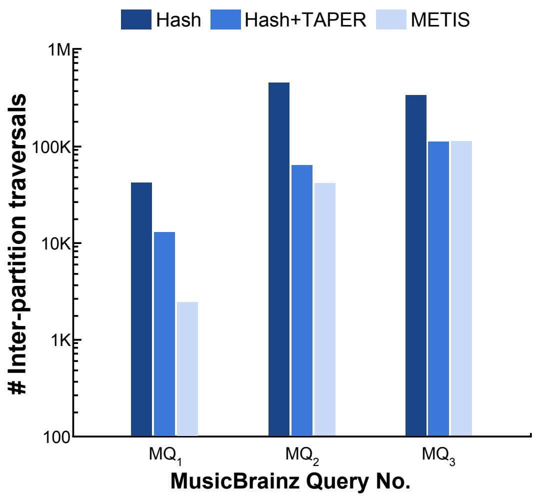


Figure 4.10: *ipt* per query

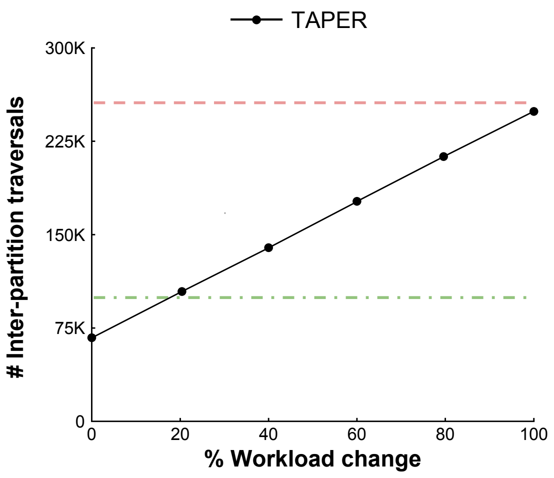


Figure 4.11: *ipt* vs Workload change

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 [12], this results in an increase of *absolute ipt* when executing $\mathcal{Q}$ 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 $k = 2$, *TAPER* achieves only around 50% reduction in *ipt* relative to Hash; for $k = 16$ this reduction is over 80%.

4.6.2.4 Optimising for frequent queries

Figure 4.10 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 $\mathbf{MQ_1}$, $\mathbf{MQ_2}$, and $\mathbf{MQ_3}$ at 10%, 20% and 70%, respectively. Relative to the METIS partitioning, a *TAPER* invocation achieves its worst quality for $\mathbf{MQ_1}$, improving with $\mathbf{MQ_2}$ and surpassing the other system for $\mathbf{MQ_3}$. This is because

[12]Which prevent *TAPER* from amassing extroverted vertices to a single partition, reducing *ipt* regardless of k

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.

4.6.2.5 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 (Sec. 4.2.2). However, within a larger workload stream, query frequencies are likely to change continuously. Figure 4.11 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: $\mathbf{Q_a}$ *Entity* · *Entity* and $\mathbf{Q_b}$ *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 $\mathbf{Q_a}$ is 100%; throughout the stream the frequency of $\mathbf{Q_a}$ tends linearly to 0%, $\mathbf{Q_b}$ to 100%. The initial partitioning has been pre-improved with *TAPER*, assuming a workload of 100% $\mathbf{Q_a}$ queries. As the frequency of $\mathbf{Q_b}$ queries increases, so does the *ipt*. For comparison, the top dotted line in Figure 4.11 shows the *ipt* required to execute solely $\mathbf{Q_b}$ queries over a hash-partitioning of the graph; the bottom line shows those required over a partitioning improved by *TAPER* **correctly assuming** $\mathbf{Q_b}$ = 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 (Sec. 4.4) and our experiments depicted in Figure 4.7 demonstrate that *TAPER* invocations are inexpensive compared to a full re-partitioning operation. Therefore, by

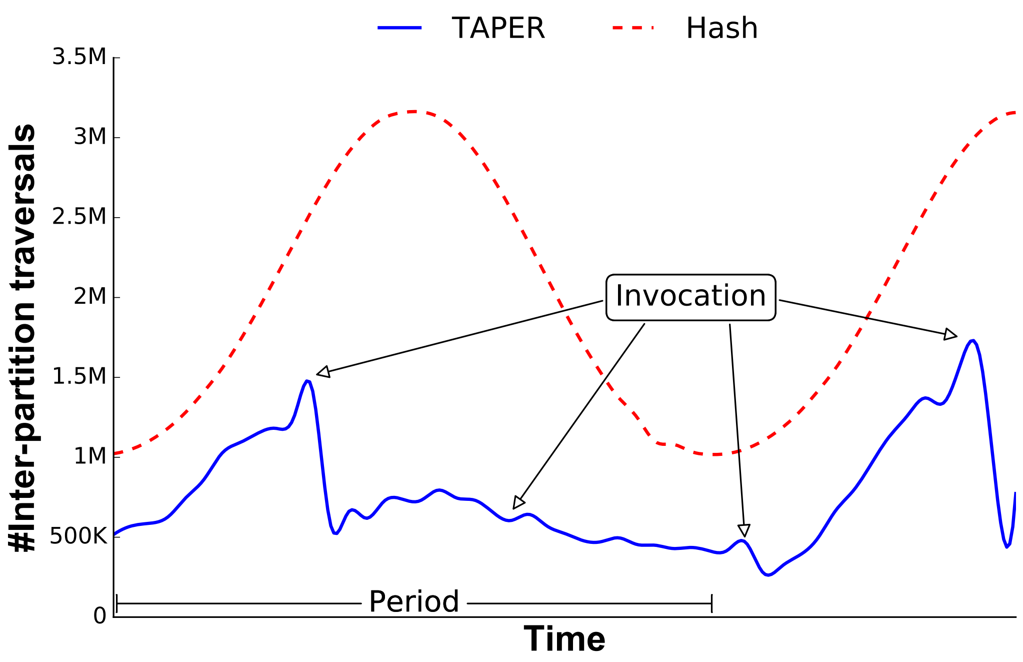


Figure 4.12: *ipt* over time w. *TAPER* invocations

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 4.12 presents the *ipt* which occur when executing a full streaming query workload, generated as described (Sec. 4.6.1.2), 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. 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, Figure 4.12 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.

4.7 Conclusion

In this chapter we have presented *TAPER*: a practical system for improving path query performance over graph partitionings. 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. Using vertex labels in the TPSTry (Sec. 4.4) as an intensional representation of these traversal patterns, along with several other heuristics and datastructures such as the Visitor Matrix, 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-sensitive, *TAPER* may even improve the quality of input partitionings already good *w.r.t.* some workload agnostic objective function, such as min edge-cut.

The final experiment shows that, with the TPSTry acting as a continuous workload summary, and the incremental nature of *TAPER* invocations, we are able to maintain high *workload-sensitive stability* and therefore low *ipt*, in the presence of a change stream of queries.

Recall that in terms of our comparison framework for partitioning techniques (Sec. 3.6) this means that *TAPER* has both the **WS** and **OW** properties: it is sensitive to online, dynamic query workloads. Table 4.1 presents an overview of all the framework properties *TAPER* possesses, along with Table 4.2 which acts as an index for property keys.

Table 4.1: Properties of the *TAPER* re-partitioner

System	**S**	**D**	**SC**	**DY**	**RF**	**WS**	**OW**	**TX**
TAPER		Y	P		Y	Y	Y	P

Table 4.2: Comparison framework properties overview

Key	**Description**
S	Capable of partitioning graph streams
D	Capable of partitioning distributed graphs
SC	Capable of partitioning very large graphs
DY	Capable of partitioning dynamic, growing graphs
RF	Partitionings do not rely upon replication
WS	Partitionings are sensitive to given workloads
OW	Partitionings are sensitive to changing workloads
TX	Partitionings are sensitive to changing data-management workloads

Though we contend that the work presented in this chapter is a valuable contribution in isolation, Table 4.1 also serves to highlight some properties which are either partially or wholly unaddressed.

Firstly, there is no support for the partitioning of dynamic, growing graphs (**DY**) as *TAPER* is a strict repartitioner and cannot produce initial partitionings of graphs at all, whether static or dynamic. Instead, *TAPER* may be used in conjunction with another partitioner which **does** have the **DY** property, such as Hash or streaming graph partitioners[110, 117].

Secondly, we indicate only partial support for data-management workload sensitivity (**TX**). As mentioned, the TPSTry is only capable of encoding path queries: single regular expressions over the set of vertex labels (Sec. 4.2). As a result, *TAPER*'s partitioning is only sensitive to workloads of path queries, a subset of the more general sub-graph pattern matching query workloads required in an online, data-management context.

Thus, there exist several useful and interesting avenues for further research, building on the contributions of this chapter. Indeed, in the subsequent chapter, we present a workload-sensitive graph stream partitioning technique which, while unable to improve existing partitionings, does support dynamic, growing graphs (i.e. possesses the **DY** property). This technique also makes use of a modified TPSTry datastructure, able to encode arbitrary graph patterns, and is therefore sensitive to the more general class of workloads (i.e. fully supports the **TX** property). Finally, we still plan to explore more sophisticated, predictive, trigger conditions for *TAPER* invocations when given a changing workload stream, as the current regular intervals are ineffective.

5

Loom: Query-aware Partitioning of Online Graphs

Contents

Summary

In the previous chapter we presented *TAPER*, an efficient graph repartitioning technique which is sensitive to workloads of path queries.

In this chapter we present a further technique, *Loom* which creates graph partitionings from scratch. Loom consumes the stream of graph updates which constitute a dynamic, growing graph and continuously allocates new vertices and edges to partitions. These allocations account for a given workload of sub-graph pattern matching queries, along with their relative frequencies.

First we capture the most common patterns of edge traversals which occur when executing queries. We then compare sub-graphs, which present themselves incrementally in the graph update stream, against these common patterns. Finally we attempt to allocate each match to single partitions, reducing the number of inter-partition edges within frequently traversed sub-graphs and improving average query performance.

Loom is extensively evaluated over several large test graphs with realistic query workloads and various orderings of the graph updates. We demonstrate that, given a workload, our prototype produces partitionings of significantly better quality than existing streaming graph partitioning algorithms Fennel & LDG.

5.1 Introduction

The overall aim of this thesis is to propose graph partitioning techniques which are suitable for online, data-management contexts (Sec. 1.3).

The sub-graph pattern matching query is a class of operation fundamental to most applications which use the graph data model online. In addition to the application domains highlighted in the Introduction chapter of this thesis, these queries are used for “real-time” anomaly detection in computer network logs [19] and payment records [116], as well as a means for interacting with web-based APIs [31, 48].

Efficiently and continuously partitioning large, growing graphs to optimise for given workloads of such queries is the primary goal of this chapter.

Informally [1], recall that a sub-graph pattern matching query usually involves exploring the sub-graphs of a large, labelled graph G then finding those which match a small labelled graph q. Figure 5.1 [2] presents an example graph G and a set of queries Q, which we will refer to throughout. Note that, as mentioned, sub-graph pattern matching queries are a super-category containing the path queries considered in the previous chapter (Sec. 4.2). Indeed, q_2 in Figure 5.1 *is* equivalent to a path.

Just as existing partitioners are unsuitable for graphs subject to dynamic workloads of path queries, they are unsuitable for dynamic graphs subject to workloads of pattern matching queries. The reasons for this have been stated several times throughout this thesis (Ch. 1, 3 & 4), and crystallised in a comparison framework of eight properties which indicate how suitable a given partitioner is to addressing our thesis aim.

Namely, however, workload-agnostic partitioners [17, 33, 51, 52, 58, 59, 59, 76, 83, 87, 102, 110, 117] employ objective functions which poorly approximate partitioning quality with respect to a given workload of sub-graph pattern matching queries (Sec. 1.2.1). As a result, partitionings will suffer many expensive inter-partition traversals (*ipt*) during workload-execution.

Even if a graph partitioning is initially high quality with respect to a given workload, recall that most existing partitioners are not designed for incremental execution and are too computationally intesnsive to re-execute regularly (Sec. 3.6.2). As a result, partitioning quality will deteriorate over time given dynamic graphs and workloads, both common in the online data-management context we focus on here.

The *TAPER* repartitiner was purposefully designed to address some of these shortcomings with existing partitioners. For instance, its partitionings are both of high quality with respect to a given workload and are cheaply adapted in the event of workload change (i.e. possess the **WS** and **OW** properties from our partitioning framework). However, as concluded in the previous chapter (Sec. 4.7), there are still some properties beneficial to the aim of this thesis which *TAPER* does not possess.

Firstly, as a re-partitioner it is not capable of providing an initial partitioning for dynamic graphs which, as mentioned, are common in our context. Secondly, *TAPER*

[1]For a formal definition refer to section 2.2 in the Preliminaries chapter.
[2]Repeating Figure 2.1 from Section 2.2 for ease of reference.

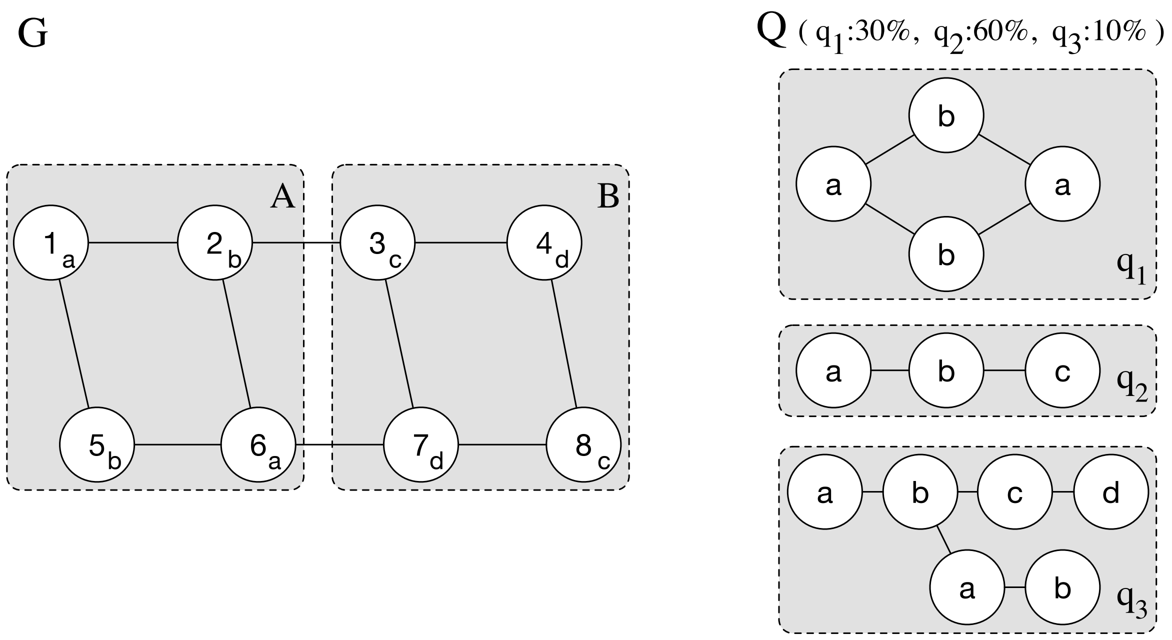


Figure 5.1: Example graph G with query workload Q

considers workloads of path queries and so is sensitive to only a subset of all possible pattern matching queries, whose distributed performance we aim to improve throughout this thesis.

In this chapter we focus on designing a workload-senstive graph partitioner which fully supports these two properties, amongst others.

To appreciate the importance of a workload-sensitive partitioning, consider the graph of Figure 5.1. The partitioning $\{A, B\}$ is optimal for the balanced min. edge-cut goal, but may not be optimal for the query graphs in Q. For example, the query graph q_2 matches the sub-graphs $\{(1,2),(2,3)\}$ and $\{(6,2),(2,3)\}$ in G. Given a workload which consisted entirely of q_2 queries, every one would require a potentially expensive inter-partition traversal (*ipt*). It is easy to see that the alternative partitioning $A' = \{1,2,3,6\}$, $B' = \{4,5,7,8\}$ offers an improvement (0 *ipt*) given such a workload, whilst being strictly worse *w.r.t* min. edge-cut.

5.1.1 The Loom partitioner

Given the above motivation, we present Loom: a system for partitioning an online, dynamic graph G into k parts (**DY**), optimising for a given workload of sub-graph pattern matching queries Q (**WS** and **TX**). The resulting partitioning $P_k(G, Q)$ reduces the *probability* of *ipt*, when executing a random $q \in Q$.

This is equivalent to reducing the total extroversion of the graph partitioning (Sec. 4.3.2), though Loom does not rely upon this definition, nor calculate explicit traversal probabilities of any sort. Instead, Loom's partitioning is made up of three distinct components. Firstly, we discover patterns of edge traversals which are common when executing queries from a given workload Q (Sec. 5.2). Secondly, we efficiently detect instances of these patterns in the ongoing stream of graph updates which constitutes an online, dynamic graph (Sec. 5.3). Thirdly, we attempt to assign these pattern matches wholly within individual partitions, avoiding the need for *ipt* when corresponding queries traverse them later (Sec. 5.4).

As mentioned (Sec. 2.1), online, dynamic graphs may be viewed as graph streams. Thus, Loom is actually a graph-stream partitioner, similar to those discussed [52, 87, 110, 117] in our literature survey (Sec. 3.4). It executes the latter two steps above continuously and is able to partition each new vertex and edge which is added as a graph changes over time $G \rightarrow G'$.

This is distinct from the *TAPER* re-partitioner (Sec. 4.1.1), which only operates over existing graph partitionings but can continuously improve the quality of a partitioning with respect to a query workload which changes over time. An effective way to summarise this key difference between the two systems is to compare their abstract definitions below:

$$\begin{aligned} \textbf{Loom} &: P_k^0(G, Q) \xrightarrow{G_1} P_k^1(G_1, Q) \xrightarrow{G_2} P_k^2(G_2, Q) \ldots \\ \textbf{TAPER} &: P_k^0(G) \xrightarrow{Q_1} P_k^1(G, Q_1) \xrightarrow{Q_2} P_k^2(G, Q_2) \ldots \end{aligned}$$

5.1.2 Contributions

Underpinning Loom's components, this chapter presents the following spcecific contributions:

- A compact [3] trie-like encoding of the most frequent traversal patterns over edges in G. This encoding is conceptually similar to the TPSTry from the previous chapter (Sec. 4.4), though its implementation and use differs significantly. We show how it may be constructed and updated given a workload of sub-graph pattern matching queries Q.
- A method of sub-graph isomorphism checking, extending a recent probabilistic technique [109]. We show how this measure may be efficiently computed and demonstrate both the low probability of false positives and the impossibility of false negatives.
- A method for efficiently and incrementally computing matches for frequent traversal patterns in a graph stream, using our trie-like encoding and isomorphism method.
- Finally, a novel heuristic for assigning matching sub-graphs from the graph stream, splitting as few as possible across inter-partition borders whilst still preserving partition balance.

As mentioned, Loom is a graph stream partitioner, therefore we present an extensive evaluation comparing Loom to popular single-pass streaming graph partitioners Fennel [117], and LDG [110]. We partition real and synthetic graph streams of various sizes and with three distinct stream orderings: breadth-first, depth-first and random order. Subsequently, we execute query workloads over each graph, counting the number of expensive *ipt* which occur. Our results indicate that Loom achieves a significant improvement over both systems, with between 15 and 40% fewer *ipt* when executing a given workload.

[3]Grows with the size of query graph patterns, which are typically small

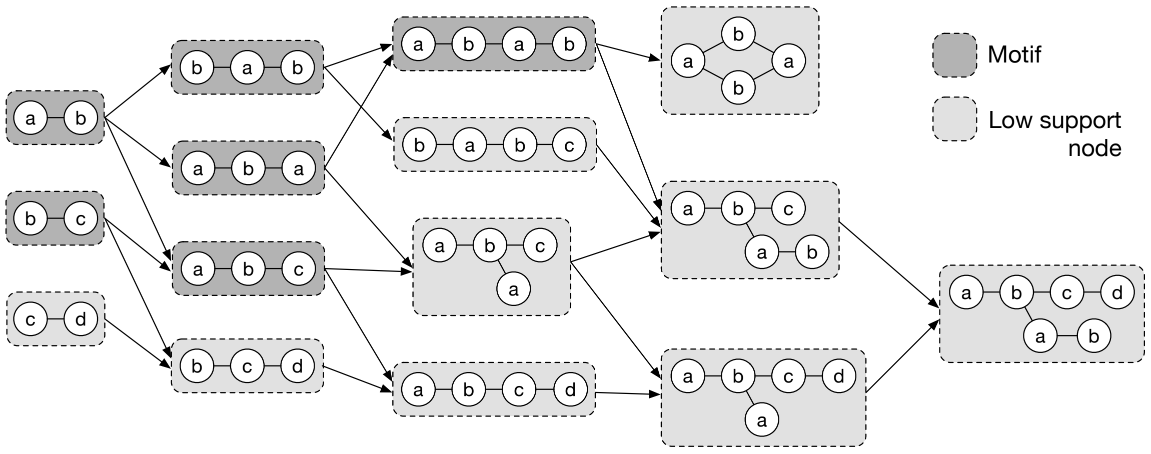


Figure 5.2: TPSTry++ for $\mathcal{Q}$ in fig. 2.1

5.2 Identifying Motifs

We now describe the first of Loom's three distinct components (Sec. 5.1.1), namely the encoding of all query graphs found in our pattern matching query workload Q. For this we use a trie-like datastructure, conceptually extending the *Traversal Pattern Summary Trie* (TPStry) from the previous chapter (Sec. 4.4); we refer to this new datastructure as a TPSTry++. For this, we use a trie-like datastructure which we have called the *Traversal Pattern Summary Trie* (TPSTry++). A TPSTry++ forms a directed acycling graph (DAG) in which every node [4] represents a graph, while every parent node represents a sub-graph which is common to the graphs represented by its children. As an illustration, the complete TPSTry++ for the workload Q in Fig. 2.1 is shown in Fig. 5.2.

This structure not only encodes all sub-graphs found in each $q \in Q$, it also associates a support value p with each of its nodes. This support value tracks the relative frequency of occurences of each sub-graph in our query graphs, revealing query *motifs*.

Recall that, given a threshold $\mathcal{T}$ for the frequency of occurrences, a *motif* is defined as a sub-graph which occurs at least $\mathcal{T}$ times within some larger graph or collection of graphs (Sec. 2.1); in this instance, the query graphs in Q. As an example, for $\mathcal{T} = 40\%$, Q's motifs are the shaded nodes in Fig. 5.2.

[4]Note again that throughout this thesis, we refer to elements of data and query graphs as vertices, elements of metadata structures like The TPSTry++ as nodes.

Intuitively, a sub-graph of G which is frequently traversed by a query workload should be assigned to a single partition in order to reduce the number of inter-partition traversals which occur during query processing. We can idenfity these sub-graphs as they form within the stream of graph updates, by matching them against the motifs in the TPSTry++. Details of the motif matching process are provided in Sec. 5.3. In the rest of this section we explain how a TPSTry++ is constructed, given a workload Q. First however, to appreciate the difference between the TPSTry++ and its predecessor, consider the following.

The TPSTry is a tree datastructure which encodes a space of possible traversal **paths** as a conventional prefix trie of strings (Sec. 4.4). Each path is a string of vertex labels and possible paths are described by a stream of regular path queries [81].

In contrast, the TPSTry++ employs methods from frequent sub-graph mining [54, 115] to compactly encode **general labelled graphs**. The resulting structure is formally a sub-graph lattice [115], practically a Directed Acyclic Graph (DAG), using multiple routes to each node to reflect the multiple ways in which a particular query pattern may extend smaller ones. For example in Fig. 5.2 the graph in node *a*-*b*-*a*-*b* can be produced in two ways, by adding a single *a*-*b* edge to either of the sub-graphs *b*-*a*-*b*, and *a*-*b*-*a*.

Note that the TPSTry++ is similar to, though more general than, Ribiero et al's G-Trie [99] and Choudhury et al's SJ-Tree [19], which use trees (not DAGs) to encode unlabelled graphs and labelled paths respectively.

Finally, note that the TPSTry is a relatively compact structure, as it grows with $|L_V|^t$, where t is the number of edges in the largest query graph in Q and L_V is typically small.

5.2.1 Sub-graph signatures

We build the trie for Q by progressively building and merging smaller tries for each $q \in Q$, as shown in Fig. 5.3. This process relies on detecting graph isomorphisms, as any two trie nodes from different queries that contain identical graphs should be merged. Failing to detect isomorphism would result, for instance, in two separate trie

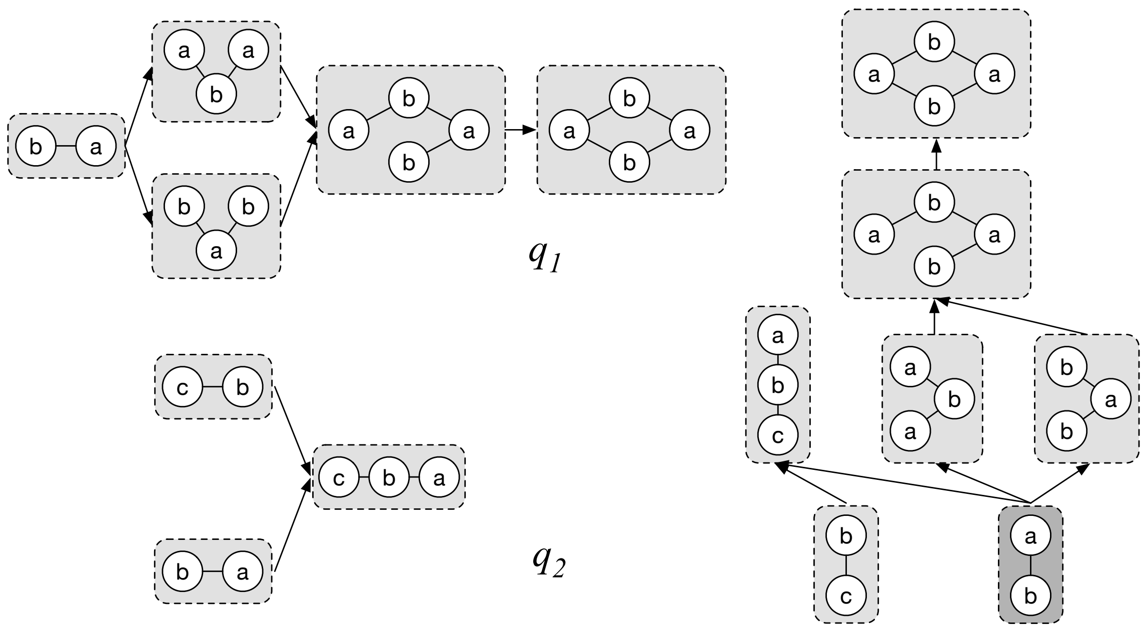


Figure 5.3: Combining tries for query graphs q_1, q_2

nodes being created for the simple graphs *a*-*b*-*c* and *c*-*b*-*a*, rather than a single node with a support of 2, as intended. One way of detecting isomorphism, often employed in frequent sub-graph mining, involves computing the lexicographical canonical form for each graph [79, 127], whereby two graphs are isomorphic if and only if they have the same canonical representation.

Computing a graph's canonical form provides strong guarantees, but can be expensive [99]. Instead, we propose a probabilistic, but computationally more efficient approach based on *number theoretic signatures*, which extends recent work by Song et al. [109]. In this approach we compute the signature of a graph as a large, pseudo-unique integer hash that encodes key information such as its vertices, labels, and nodes degree. Graphs with matching signatures are likely to be isomorphic to one another, but there is a small probability of collision, i.e., of having two different graphs with the same signature.

Given a query graph $G_q = \{V_q, E_q\}$ we compute its signature as follows. Initially we assign a random value $r(l) = [1, p)$, between 1 and some user specified prime p, to each possible label $l \in L_{V_i}$ from our data graph G; recall that the function f_l maps vertices in G to these labels. We then perform three steps:

1. Calculate a factor for each edge $e = (v_i, v_j) \in E_q$, according to the formula:

$$edgeFac(e) = (\ |r(f_l(v_i)) - r(f_l(v_j))|\ )\ mod\ p$$

2. Calculate the factors that encode the degree of each vertex. If a vertex v has a degree n, its degree factor is defined as:

$$degFac(v) = ((r(f_l(v)) + 1)\ mod\ p)\cdot$$
$$((r(f_l(v)) + 2)\ mod\ p) \cdot \ldots \cdot ((r(f_l(v)) + n)\ mod\ p)$$

3. Finally, we compute the signature of $G_q = (V_q, E_q)$ as:

$$(\prod_{e \in E_i} edgeFac(e)) \cdot (\prod_{v \in V_i} degFac(v))$$

To illustrate, consider query q_1 from Fig. 2.1. Given a p of 11 and random values $r(a) = 3$, $r(b) = 10$ we first calculate the edge factor for an a-b edge: $edgeFac((a, b)) = (|3 - 10|)\ mod\ 11 = 7$. As q_1 consists of four a-b edges, its total edge factor is $7^4 = 2401$. Then we calculate the degree factors [5], starting with a b labelled vertex with degree 2: $degFac(b) = ((10+1)\ mod\ 11) \cdot ((10+2)\ mod\ 11) = 11$, followed by an a labelled vertex also with degree 2: $degFac(a) = 20$. As there are two of each vertex, with the same degree, the total degree factor is $11^2 \cdot 20^2 = 48400$. The signature of $q_1 = 2401 \cdot 48400 = 116208400$.

Note that an edge's factor is calculated using the *absolute* difference between the two random values corresponding to its vertices, e.g. $|3 - 10| = 7$. This is due to the fact that we consider only undirected edges throughout this thesis (Sec 2.1). However, factors may be extended to account for directed edges, by simply subtracting the random value for a destination vertex from that of a source vertex. For example, $edgeFac'((a, b)) = (3 - 10)\ mod\ 11 = 4$, while $edgeFac'((b, a)) = (10 - 3)\ mod\ 11 = 7$.

This signature based approach to graph isomorphism is appealing for two reasons. Firstly, since the factors in the signature may be multiplied in any order, a signature for G can be calculated incrementally if the signature of any of its sub-graphs G_i is known, as this is the combined factor due to the additional edges and degree in $G \setminus G_i$.

[5]Note we don't consider 0 a valid factor, and replace it with p (e.g. $11\ mod\ 11 = 11$)

Secondly, the choice of p determines a trade-off between the probability of collisions and the performance of computing signatures. Specifically, note that signatures can be **very** large numbers (thousands of bits) even for small graphs, rendering operations such as remainder costly and slow. A small choice of p reduces signature size, because all the factors are mapped to a finite field [71] (*factor mod p*) between 1 and p, but it increases the likelihood of collision, i.e., the probability of two unrelated factors being equal. We discuss how to improve the performance and accuracy of signatures in Section 5.2.3.

5.2.2 Constructing the TPSTry++

Our approach to constructing the TPSTry++ is to incrementally compute signatures

Algorithm 2 Recursively add a query graph G_q to a TPSTry++

```
1: $factors(e, g)$ ← degree/edge factors to multiply a graph $g$'s signature when adding edge $e$
2: $support(g)$ ← a map of TPSTry++ nodes (graphs) to $p$-values
3: $tpstry$ ← the TPSTry++ for workload $\mathcal{Q}$
4: $parent$ ← a TPSTry++ node, initially root (an empty graph)
5: $G_q$ ← the query graph defined by a query $q$
6: $g$ some sub-graph of $G_q$

7: for $e$ in edges from $G_q$ do
8:     g ← new empty graph
9:     corecurse(parent, e, tpstry, g)
10:        $sig \leftarrow factor(e, g) \cdot parent.signature$
11:        if $tpstry.signatures$ contains $sig$ then
12:            $n$ ← node from $tpstry$ with signature $sig$
13:            $n.support \leftarrow n.support + 1$
14:        else
15:            $n$ ← new node with graph $g + e$, signature $sig$ and support 1
16:            $tpstry \leftarrow tpstry + n$
17:        if not $parent.children$ contains $n$ then
18:            $parent.children \leftarrow parent.children + n$
19:        $newEdges$ ← edges incident to $g + e$ & not in $g + e$
20:        for $e'$ in $newEdges$
21:            corecurse($\mathbf{n}, \mathbf{e'}, \mathbf{tpstry}, \mathbf{g + e}$)
22: return $tpstry$
```

for sub-graphs of each query graph q in a trie, merging trie nodes with equal signatures to produce a DAG which encodes the sub-graphs of all $q \in Q$. Alg. 2 formalises this approach.

Essentially, we recursively "rebuild" the graph $G_q \mid Eq \mid$ times, starting from each edge $e \in E_q$ in turn. For an edge e we calculate its edge and degree factors, initially assuming a degree of 1 for each vertex. If the resulting signature is not associated with a child of the TPSTry++'s root, then we add a node n representing e. Subsequently, we "add" those edges e' which are incident to $e \in G_q$, calculating the additional edge and degree factors, and add corresponding trie nodes as children of n. Then we recurse on the edges incident $e + e'$.

Consider again our earlier example of the query graph q_1: as it arrives in the workload stream Q, we break it down to its constituent edges $\{a\text{-}b,\ a\text{-}b,\ a\text{-}b,\ a\text{-}b\}$. Choosing an edge at random we calculate its combined factor. We know that the edge factor of an *a-b* edge is 7. When considering this single edge, both a and b vertices have a degree of 1, therefore the signature for *a-b* is $7 \cdot ((3+1)\ mod\ 11) \cdot ((10+1)\ mod\ 11) = 308$. Subsquently, we do the same for all other edges and, finding that they have the same signature, leave the trie unmodified. Next, for each edge, we add each incident edge from q_1 and compute the new combined signature. Assume we add another *a-b* edge adjacent to b to produce the sub-graph *a-b-a*. This produces three new factors: the new edge factor 7, the new a vertex degree factor $((3+1)\ mod\ 11)$ and an additional degree factor for the existing b vertex $((10+2)\ mod\ 11)$. The combined signature for *a-b-a* is therefore $308 \cdot 7 \cdot 4 \cdot 1 = 8624$; if a node with this signature does not exist in the trie as a child of the *a-b* node, we add it. This continues recursively, considering larger sub-graphs of q_1 until there are no edges left in q_1 which are not in the sub-graph, at which point, q_1 has been added to the TPSTry++.

5.2.3 Avoiding signature collisions

As mentioned, number theoretic signatures are a probabilistic method of ismorphism checking, prone to collisions. There are several scenarios in which two non-isomorphic graphs may have the same signature: *a)* two factors representing different graph features, such as different edges or vertex degrees, are equal; *b)* two distinct sets of factors have the same product; and *c)* two different graphs have identical sets of edges, vertices and vertex degrees.

The original approach to graph isomorphic checking [109] makes use of an expensive authoritative pattern matching method [73] to verify identified matches. Given a query graph, it calculates its signature in advance, then incrementally computes signatures for sub-graphs which form within a window over a graph stream. If a sub-graph's signature is ever divisible by that of the query graph, then that sub-graph should contain a query match.

There are some key differences in how we compute and use signatures with Loom, which allow us to rely solely upon signatures as an efficient means for mining and matching motifs. Firstly, remember our overall aim is to heuristically lower the probability that sub-graphs in a graph G which match our discovered motifs straddle a partition boundary. As a result we can tolerate some small probability of false positive results, whilst the manner in which signatures are executed (Sec. 5.2.1) precludes false negatives; i.e. two graphs which **are** isomorphic are guaranteed to have the same signature. Secondly, we can exploit the structure of the TPSTry++ to avoid ever explicitly computing graph signatures. From Fig. 5.2 and Alg. 2, we can see that all possible sub-graphs of a query graph G_q will exist in the TPSTry++ by construction. We calculate the edge and degree factors which would multiply the signature of a sub-graph with the addition of each edge, then associate these factors to the relevant trie branches. This allows us to represent signatures as sets of their constituent factors, which eliminates a source of collisions, e.g. we can now distinguish between graphs with factors $\{6, 2\}$, $\{4, 3\}$ and $\{12\}$. Thirdly, we never attempt to discover whether some sub-graph **contains** a match for query q, only whether it **is** a match for q. In other words, the largest graph for which we calculate a signature is the size of the largest query graph $|G_q|$ for all $q \in Q$, which is typically small[6]. This allows us to choose a larger prime p than Song et al. might, as we are less concerned with signature size, reducing the probability of factor collision, another source of false positive signature matches.

Concretely, we wish to select a value of p which minimises the probability that more than some acceptable percentage $\mathcal{C}\%$ of a signature's factors are collisions. From Section 5.2.1 there are three scenarios in which a factor collision may occur: *a)* two

[6]Of the order of 10 edges.

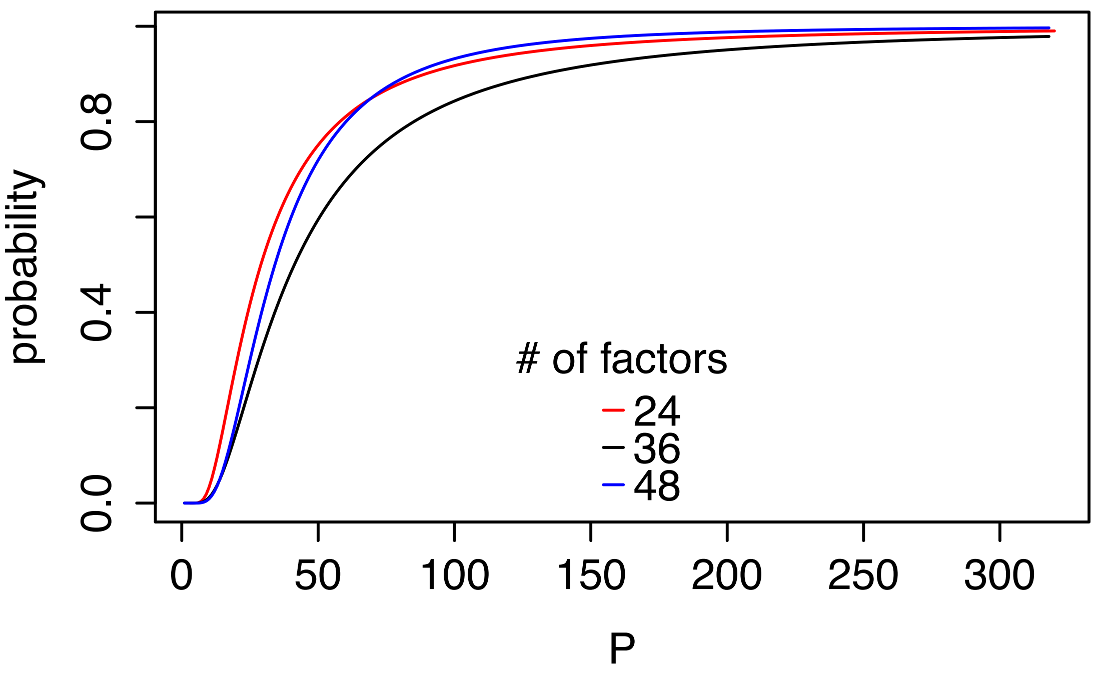


Figure 5.4: Probability of $< 5\%$ factor collisions for various p

edge factors are equal despite different vertices with different random values from our range $[1, p)$; *b)* an edge factor is equal to a degree factor; and *c)* two degree factors are equal, again despite different vertices. Song et al. show that all factors are uniform random variables from $[1, p)$, therefore each scenario occurs with probability $\frac{1}{p}$.

For either edge or degree factors, from the above it is clear that there are two scenarios in which a collision may occur, giving a collision probability for any given factor of $\frac{2}{p}$. The Handshaking lemma [27] tells us that the total degree of a graph must equal $2|E|$, which means that a graph must have $3|E|$ factors in its signature: one per edge plus one per degree. Combined with the binary measure of "success" (collision / no collision), this suggests a binomial distribution of factor collision probabilities, specifically $Binomial(3|E|, \frac{2}{p})$. Binomial distributions tell us the probability of exactly x "successes" occuring, however we want the probability that no more than $\mathcal{C}_{max} = \mathcal{C}\% \cdot 3|E|$ factors collide and so must sum over all acceptable outcomes $x \in \mathcal{C}_{max}$:

$$\sum_{x=0}^{\mathcal{C}_{max}} Pr(X = x) \text{ where } X \sim Binomial(3|E|, \frac{2}{p})$$

Figure 5.4 shows the probabilities of having fewer than 5% factor collisions given query graphs of 8, 12 or 16 edges and p choices between 2 and 317. In Loom, when identifying and matching motifs, we use a p value of 251, which as you can see gives a neglible probability of significant factor collisions.

5.3 Matching Motifs

We have seen how motifs that occur in Q are identified. By construction, motifs represent graph patterns that are frequently traversed during executions of queries in Q. Thus, the sub-graphs of G that match those motifs are expected to be frequently visited together and are therefore best placed within the same partition. In this section we clarify how we discover pattern matches between sub-graphs and motifs, whilst in the next Section we describe the allocation of those sub-graphs to partitions.

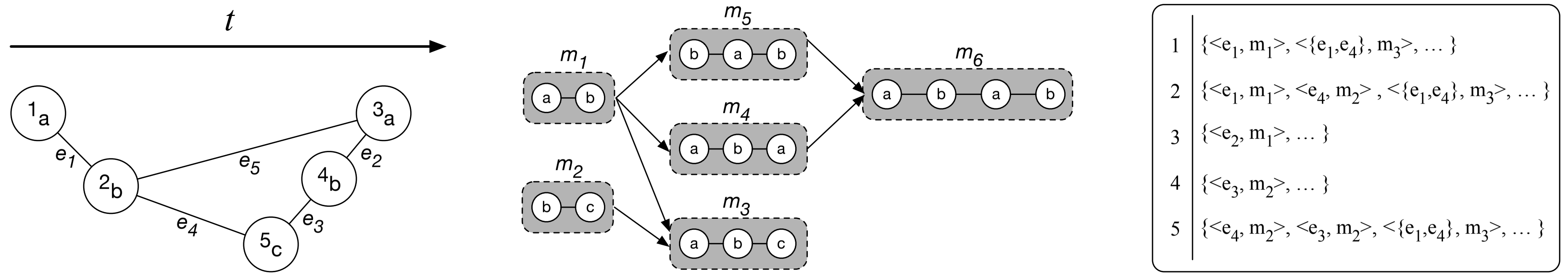
Loom operates on a sliding window of configurable size over the stream of edges that make up the growing graph G. The system monitors the connected sub-graphs that form in the stream within the space of the window, efficiently checking for isomorphisms with any known motif each time a sub-graph grows. Upon leaving the window, sub-graphs that match a motif are immediately assigned to a partition, subject to partition balance constraints as explained in Section 5.4.

Note that this technique introduces a delay, corresponding to the size of the window, between the time edges are submitted to the system and the time they are assigned and made available. In order to allow queries to access the new parts of graph G, Loom views the sliding window *itself* as an extra partition, which we denote P_{temp}. In practice, vertices and edges in the window are accessible in this temporary partition prior to being permanently allocated to their own partition.

To help understand how the matching occurs, note that in the TPSTry++, by construction, *all* anscestors of any node n *must* represent strict sub-graphs of the graph represented by n itself. Also, note that the support of a node n is the relative frequency with which n's sub-graph G_n occurs in Q. As, by definition, each time G_n occurs in Q so do all of *its* sub-graphs, a trie node n must have a support lower than any of its anscestors. This means that if any of the nodes in the trie, including those representing single edges, are not motifs, then none of their descendants can be motifs either. Thus, when a new edge $e = (v_1, v_2)$ arrives in the graph stream, we compute its signature (Sec. 5.2.1) and check if e matches a single-edge motif at the root of the TPSTry++. If there is no match, we can be certain that e will never form part of any sub-graph that matches a motif. We therefore immediately assign e to a partition and do not add it to our stream window P_{temp}. If, on the other hand, e does match a single-edge motif then we record the match into a map, *matchList*, and add e to the window. The *matchList* maps vertices v to the set of motif matching sub-graphs in P_{temp} which contain v; i.e. having determined that $e = (v_1, v_2)$ is a motif match, we treat e as a sub-graph of a single edge, then add it to the *matchList* entries for both v_1 and v_2. Additionally, alongside every sub-graph in *matchList*, we store a reference to the TPSTry++ node which represents the matching motif. Therefore, entries in *matchList* take the form $v \to \{\langle E_i, m_i \rangle, \langle E_j, m_j \rangle, \ldots\}$, where E_i is a set of edges in P_{temp} that form a sub-graph g_i with the same signature as the motif m_i.

Given the above, any edge e which is added to P_{temp} must at least match a single edge motif. However, if e is incident to other edges already in P_{temp}, then its addition may also form larger motif matching sub-graphs which we must also detect and add to *matchList*. Thus, having added $e = (v_1, v_2)$ to *matchList*, we check the map for existing matches which are connected to e; i.e we look for matches which contain one of v_1 or v_2. If any exist, we use the procedure in Alg. 3, along with the TPSTry++, to determine whether the addition of edge e to these sub-graphs creates another motif match.

Essentially, for each sub-graph g_i from *matchList* to which e is connected, we calculate the set of edge and degree factors $fac(e, g_i)$ which would multiply the signature of g_i upon the addition e, as in Sec. 5.2. Recall, also from Sec. 5.2, that a TPSTry++ node contains a signature for the graph it represents, and that these signatures are stored as sets of factors, rather than their large integer products. As each sub-graph in *matchList* is paired with its associated motif n from the trie, we can efficiently check if n has a child c where *a)* c is a motif; and *b)* the difference between n's factor set and c's factor set corresponds to factors for the addition of e to g_i, i.e., $fac(e, g_i) = c.signatures \setminus n.signatures$. If such a child exists in the trie then adding e to a graph which matches motif n (g_i) will likely create a graph which matches motif c: the addition of e to P_{temp} has formed the new motif matching sub-graph $g_i + e$.



1	{<e_1, m_1>, <{e_1,e_4}, m_3>, … }
2	{<e_1, m_1>, <e_4, m_2> , <{e_1,e_4}, m_3>, … }
3	{<e_2, m_1>, … }
4	{<e_3, m_2>, … }
5	{<e_4, m_2>, <e_3, m_2>, <{e_1,e_4}, m_3>, … }

Figure 5.5: t-length window over G (left), Motifs from TPSTry++ (center) and motif *matchList* for window (right)

Algorithm 3 Mine motif matches from each new edge $e \in G$

```
1: $fac(e, g) \leftarrow$ degree/edge factors to multiply a graph $g$'s signature when adding edge $e$
2: $tpstry \leftarrow$ the filtered TPSTry++ of motifs for workload $Q$

3: for each new edge $e(v_1, v_2)$ do
4:    $matches \leftarrow matchList(v_1) \cup matchList(v_2)$
5:    for each sub-graph $m$ in $matches$ do
6:       $n \leftarrow$ the $tpstry$ node for $m$
7:       if $n$ has child $c$ w. $factor = fac(e, m)$ then
8:          add $\langle m + e, c \rangle$ to $matchList$ for $v_1 \& v_2$ //Match found!
9:    $ms_1 \leftarrow matchList(v_1)$
10:   $ms_2 \leftarrow matchList(v_2)$
11:   for all possible pairs $(m_1, m_2)$ from $(ms_1, ms_2)$ do
12:      $n_1 \leftarrow$ the $tpstry$ node for $m_1$
13:      recurse$(\mathbf{tpstry}, \mathbf{m_2}, \mathbf{m_1}, \mathbf{n_1})$
14:         for each edge $e_2$ in $m_2$ do
15:            if $n_1$ has child $c_1$ w. $factor = fac(e_2, m_1)$ then
16:               recurse$(\mathbf{tpstry}, \mathbf{m_2} - \mathbf{e_2}, \mathbf{m_1} + \mathbf{e_2}, \mathbf{c_1})$
17:         if $m_2$ is empty then //Match found!
18:            add $\langle m_1 + m_2, n_1 \rangle$ to $matchList$ for $v_1 \& v_2$
```

We also detect if the joining of two existing multi edge motif matches ($\langle E_1, m_1 \rangle$, $\langle E_2, m_2 \rangle$) forms yet another motif match, in roughly the same manner. First we consider each edge from the smaller motif match (e.g. $e \in E_2$ from $\langle E_2, m_2 \rangle$), checking if the addition of any of these edges to E_1 constitutes yet another match [7]; if it does then we add the edge to E_1 and recursively repeat the process until E_2 is empty. If this process **does** exhaust E_2 then $E_1 \cup E_2$ constitute a motif matching sub-graph. Once this process is complete, *matchList* will contain entries for **all** of the motif matching sub-graphs currently in P_{temp}. Note that as more edges are added to P_{temp}, *matchList* may contain multiple entries for a given vertex where one match is a sub-graph of another, i.e. new motif matches don't replace existing ones.

As an example of the motif matching process, consider the portion of a graph stream (left), motifs (center) and *matchList* (right) depicted in Fig. 5.5. Our window over the graph stream G is initially empty, with the depicted edges being added in label order (i.e. $e_1, e_2, \ldots$). As the edge e_1 is added, we first compute its signature and verify whether e_1 matches a single-edge motif in the TPSTry++. We can see that, as an *a*-*b* labelled edge, the signature for e_1 must match that of motif m_1, therefore we

[7]Treating E_1 as a sub-graph.

add e_1 to P_{temp}, and add the entry $\langle e_1, m_1\rangle$ to *matchList* for both e_1's vertices 1,2. As e_1 is not yet connected to any other edges in P_{temp}, we do not need to check for the formation of additional motif matches. Subsequently, we perform the exact same process for edge e_2. When e_3 is added, again we verify that, as a *b*-*c* edge, e_3 is a match for the single-edge motif m_3 and so update P_{temp} and *matchList* accordingly. However, e_3 is connected to existing motif matching sub-graphs in P_{temp} therefore the union of *matchList* entries for e_3's vertices 4,5 (line 4 Alg. 3) returns $\{\langle e_2, m_1\rangle\}$. As a result, we calculate the factors to multiply e_2's signature by, when adding e_3. Remember that when computing signatures, each edge has a factor, as well as each degree. Thus, when adding e_3 to e_2 our *new* factors are an edge factor for a *b*-*c* labelled edge, a first degree factor for the vertex labelled *c* (5) and a second degree factor for the vertex labelled *b*[8] (4) (Sec. 5.2.1). Subsquently we must check whether the motif for e_2, m_1, has any child nodes with additional factors consistent with the addition of a *b*-*c* edge, which it does: m_3. This means we have found a new sub-graph in P_{temp} which matches the motif m_3, and must add $\langle\{e_2, e_3\}, m_3\rangle$ to the *matchList* entries for vertices 3, 4 and 5. Similarly, the addition of *b*-*c* labelled edge e_4 to our graph stream produces the new motif matches $\langle e_4, m_2\rangle$ and $\langle\{e_1, e_4\}, m_3\rangle$, as can be seen in our example *matchList*.

Finally, the addition of our last edge, e_5, creates several new motif matches (e.g. $\langle\{e_1, e_5\}, m_4\rangle, \langle\{e_2, e_5\}, m_5\rangle$ etc...). In particular, notice that the addition of e_5 creates a match for the motif m_6, combining the new motif match $\langle\{e_1, e_5\}, m_4\rangle$ with an existing one $\langle e_2, m_1\rangle$. To understand how we discover these slightly more complex motif matches, consider Alg. 3 from line 11 onwards. First we retrieve the updated *matchList* entries for vertices 2 and 3, including the new motif matches gained by simply adding the single edge e_5 to connected existing motif matches, as above. Next we iterate through all possible pairs of motif matches for both vertices. Given the pair of matches $(\langle\{e_1, e_5\}, m_4\rangle, \langle e_2, m_1\rangle)$, we discover that the addition of any edge from the smaller match (i.e. e_2) to the larger produces factors which correspond to a child of m_4 in the TPSTry++: m_6. As e_2 is the only edge in the smaller match, we simply add the match $\langle\{e_1, e_2, e_5\}, m_6\rangle$ to the *matchList* entries for 1, 2, 3 and 4. In the general case however, we would not add this new match but instead recursively "grow"

[8]As, with the addition of e_3, vertex 4 has degree 2.

it with new edges from the smaller match, updating *matchList only* if all edges from the smaller match have been successfully added.

5.3.1 Building a graph index

As mentioned in Chapter 2, sub-graph pattern matching queries are defined in terms of sub-graph isomorphism checking, an NP-Complete problem [43] with computationally expensive practical implementations [79, 119]. As a result, in production systems where the graphs may be very large, a graph index [60] is used to produce a number of candidate sub-graphs which may then be verified as matching or not using an expensive algorithm. This approach improves query performance by reducing the number of isomorphism checks which need to be performed, and is known informally, as *filter-verify*.

Traditional graph indices such as *gIndex* [128] are conceptually similar to a Hash MultiMap datastructure, where the key (hash) for each entry is a canonical labelling [79] for a given pattern graph q and the value is a list of the sub-graphs in G which contain a match for q. However, some highly effective graph indices [123, 130] employ a sub-graph lattice structure rather than a HashMap of canonical labels. For example, *Lindex* [130] maintains a lattice, where each node represents a pattern graph [9] and has an associated list of matching sub-graphs in G.

Now recall that the TPSTry++ **is** a sub-graph lattice (Sec. 5.2), and observe from Figure 5.5 that if the *matchList* were grouped by motif instead of by vertex, our motif matching procedure would produce the exact equivalent of the datstructures underpinning Lindex: a sub-graph lattice containing pattern graphs (motifs), where each node is associated with a list of matching sub-graphs in G (motif matches).

Despite this equivalence between Lindex and the TPSTry++, there are some additional considerations before employing our datastructure in this manner. For example, a *matchList* which never removed entries would quickly grow quite large, needing some

[9]Or other graph feature/motif being indexed.

existing index pruning procedure [123] to remain managable. Such extensions, along with a thorough evaluation, are left as an important area for future work.

5.4 Allocating Motifs

Following graph stream pattern matching, we are left with a collection of sub-graphs, consisting solely of the most recent t edges in G, which match motifs from Q. As new edges arrive in the graph stream, our window P_{temp} grows to size t and then "slides", i.e. each new edge added to a full window causes the oldest $(t + 1^{th})$ edge e to be dropped. Our strategy with Loom is to then assign this old edge e to a permanent partition, along with the other edges in the window which form motif matching sub-graphs with e. The sole exception to this is when an edge arrives that may not form part of any motif match and is assigned to a partition immediately (Sec. 5.3). This exception does not pose a problem however, because Loom behaves as if the edge was never added to the window and therefore does not cause displacement of older edges.

Recall again that with Loom we are attempting to assign motif matching sub-graphs wholly within individual partitions with the aim of reducing *ipt* when executing our query workload Q. One naive approach to achieving this goal is as follows: When assigning an edge $e = (v_1, v_2)$, retrieve the motif matches associated with v_1 and v_2 from P_{temp} using our *matchList* map, then select the subset M_e **that contains** e, where $M_e = \{\langle E_1, m_1\rangle, \ldots \langle E_n, m_n\rangle\}$, $e \in E_i$ and E_i is a match for m_i. Finally, treating these matches as a single sub-graph, assign them to the partition with which they share the most incident edges. This approach would greedily ensure that no edges belonging to motif matching sub-graphs in G ever cross a partition boundary. However, it would likely also have the effect of creating highly unbalanced partition sizes, potentially straining the resources of a single machine, which prompted partitioning in the first place.

Instead, we rely upon two distinct heuristics for edge assignment, both of which are aware of partition balance. Firstly, for the case of non-motif-matching edges that are assigned immediately, we use the existing *Linear Deterministic Greedy* (LDG) heuristic [110]. Similar to our naive solution above, LDG seeks to assign edges[10] to the partition where they have the most incident edges. However, LDG also favours partitions with higher residual capacity when assigning edges in order to maintain a balanced number of vertices and edges between each. Specifically, LDG defines the residual capacity r of a partition S_i in terms of the number of vertices currently in S_i, given as $\mathcal{V}(S_i)$, and a partition capacity constraint C: $r(S_i) = 1 - \frac{|\mathcal{V}(S_i)|}{C}$. When assigning an edge e, LDG counts the number of e's incident edges in each partition, given as $N(S_i, e)$, and weights these counts by S_i's residual capacity; e is assigned to the partition with the highest weighted count. The full formula for LDG's assignment is:

$$\max_{S_i \in P_k(G)} N(S_i, e) \cdot (1 - \frac{|\mathcal{V}(S_i)|}{C})$$

Secondly, for the general case where edges form part of motif matching sub-graphs, we propose a novel heuristic, *equal opportunism.* Equal opportunism extends ideas present in LDG but, when assigning clusters of motif matching sub-graphs to a single partition as we do in Loom, it has some key advantages. By construction, given an edge e to be assigned along with its motif matches $M_e = \{\langle E_1, m_1\rangle \dots \langle E_n, m_n\rangle\}$, the sub-graphs E_i E_j in M_e have significant overlap (e.g. they all contain e). Thus, individually assigning each motif match to potentially different partitions would create many inter-partition edges. Instead, equal opportunism greedily assigns the match cluster to the single partition with which it shares the most vertices, weighted by each partition's residual capacity. However, as these vertices and their new motif matching edges may not be traversed with equal likelihood given a workload Q, equal opportunism also prioritises the shared vertices which are part of motif matches with higher support in the TPSTry++.

Formally, given the motif matches M_e we compute a score for each partition S_i and motif match $\langle E_k, m_k\rangle \in M_e$, which we call a *bid.* Let $\mathcal{N}(S_i, E_k) = |\mathcal{V}(S_i) \cap \mathcal{V}(E_k)|$ denote the number of vertices in the edge set E_k (which is itself a graph) that are already assigned to S_i[11]. Additionally, let $supp(m_k)$ refer to the support of motif m_k in the TPSTry++ and recall that C is a capacity constraint defined for each partition.

[10]LDG may partition either vertex or edge streams.
[11]Note that $\mathcal{N}$ is a generalisation of LDG's function N

We define the bid for partition S_i and motif match $\langle E_k, m_k \rangle$ as:

$$bid(S_i, \langle E_k, m_k \rangle) = \mathcal{N}(S_i, E_k) \cdot (1 - \frac{|\mathcal{V}(S_i)|}{C}) \cdot supp(m_k) \tag{5.1}$$

We could simply assign the cluster of motif matching sub-graphs (i.e. $E_1 \cup \ldots \cup E_n$) to the single partition S_i with the highest bid for all motif matches in M_e. However, equal opportunism further improves upon the balance and quality of partitionings produced with this new weighted approach, limiting its greediness using a rationing function we call l. $l(S_i)$ is a number between 0 and 1 for each partition, the size of which is inversely correlated with S_i's size relative to the smallest partition $S_{min} = \min_{S \in P_k(G)} |\mathcal{V}(S)|$, i.e. if S_i is as small as S_{min} then $l(S_i) = 1$. Equal opportunism sorts motif matches in M_e in descending order of support, then uses $l(S_i)$ to control both the number of matches used to calculate partition S_i's total bid, and the number of matches assigned to S_i should its total bid be the highest. This strategy helps create a balanced partitioning by *a)* allowing smaller partitions to compute larger total bids over more motif matches; and *b)* preventing the assignment of large clusters of motif matches to an already large partition. Formally we calculate $l(S_i)$ as follows:

$$l(S_i) = \frac{|\mathcal{V}(S_i)|}{S_{min}} \cdot \alpha \text{ , where } \alpha = \begin{cases} 1, & |\mathcal{V}(S_i)| = |\mathcal{V}(S_{min})| \\ 0, & |\mathcal{V}(S_i)| > |\mathcal{V}(S_{min})| \cdot b \\ \alpha, & \text{otherwise} \end{cases} \tag{5.2}$$

where α is a user specified number $0 < \alpha \leq 1$ which controls the aggression with which l penalises larger partitions and b limits the maximum imbalance. Throughout this chapter we use an empirically chosen default of $\alpha = \frac{2}{3}$ and set the maximum imbalance to $b = 1.1$, emulating Fennel [117].

Given definitions (5.1) and (5.2), we can now simply state the output of equal oppurtinism for the sorted set of motif matches M_e, as:

$$\max_{S_i \in P_k(G)} \sum_{k=0}^{l(S_i) \cdot |M_e|} bid(S_i, \langle E_k, m_k \rangle) \tag{5.3}$$

Note that motif matches in M_e which are not bid on by the winning partition are dropped from the *matchList* map, as some of their constituent edges (e.g. e, which all matches in M_e share) have been assigned to partitions and removed from the sliding window P_{temp}.

To understand how to the rationing function l improves the quality of equal opportunism's partitioning, not just its balance, consider the following: Just because an edge e' falls within the motif match set M_e of our assignee e, does not necessarily imply that placing them within the same partition is optimal. e' could be a member of many other motif matches in P_{temp} besides those in M_e, perhaps with higher support in the TPSTry++ (i.e. higher likelihood of being traversed when executing a workload Q). By ordering matches by support and prioritising the assignment of the smaller, higher support motif matches, we often leave e' to be assigned later along with matches to which it is more "important".

As an example, consider again the graph and TPSTry++ fragment in Fig. 5.5. If assigning the edge e_1 to a partition at the time $t + 1$, its **support ordered** set of motif matches M_{e_1} would be $\langle e_1, m_1\rangle$, $\langle\{e_1, e_4\}, m_3\rangle$, $\langle\{e_1, e_5\}, m_4\rangle$ and $\langle\{e_1, e_2, e_5\}, m_6\rangle$. Assume two partitions S_1 and S_2, where S_1 is 33.3% larger than S_2 and vertex 2 already belongs to partition S_1, whilst all other vertices in the window are as yet unassigned (i.e. this is the first time edges containing them have entered the sliding window). In this scenario, S_1 is guaranteed to win all bids, as S_2 contains no vertices from M_{e_1} and therefore $\mathcal{N}(S_2, \_)$ will always equal 0. However, rather than greedily assign all matches to the already large S_1, we calculate the ration l for S_1 as $\frac{1}{1.33} \cdot \frac{1}{1.5} = \frac{1}{2}$, given $\alpha = 1.5$. In other words, we only assign edges from the first half of M_{e_1} ($\langle e_1, m_1\rangle$, $\langle\{e_1, e_4\}, m_3\rangle$) to S_1; edges such as e_5 and e_2 remain in the window P_{temp}. Assume an edge $e_6 = (4, 6)$ subsequently arrives in the graph stream G, where vertex 6 already belongs to partition S_2 and e_6 matches the motif m_2 (i.e. has labels *b*-*c*). If we had already assigned e_5 to partition S_1 then this would lead to an inter-partition edge which is more likely to be traversed together with e_5 than are other edges in S_1, given our workload Q. Instead, we compute a match in P_{temp} between $\{e_5, e_6\}$ and the motif m_3, and will likely later assign e_5 to partition S_2. Within reason, the longer an edge remains in the sliding window, the more of its neighbourhood information we are likely to have access to, the better partitioning decisions we can make for it.

5.5 Evaluation

Our evaluation aims to demonstrate that Loom achieves high quality partitionings of several large graphs in a single-pass, streaming manner. Recall that we measure graph partitioning quality using the number of inter-partition traversals (*ipt*) when executing a realistic workloads of pattern matching queries over each graph (Sec. 2.4).

Loom consistently produces partitionings of around 20% superior quality when compared to those produced by state of the art alternatives: LDG [110] and Fennel [117] Furthermore, Loom partitionings' quality improvement is robust across different numbers of partitions (i.e. a 2-way or a 32-way partitioning). Finally we show that, like other streaming partitioners, Loom is sensitive to the arrival order of a graph stream, but performs well given a pseudo-adversarial random ordering.

5.5.1 *Experimental setup*

For each of our experiments, we start by streaming a graph from disk in one of three predefined orders:

- **Breadth-first:** computed by performing a breadth-first search across all the connected components of a graph.
- **Random:** computed by randomly permuting the existing order of a graph's elements.
- **Depth-first:** computed by performing a depth-first search across the connected components of a graph.

We choose these stream orderings as they are common to the evaluations of other graph stream partitioners [52, 87, 110, 117], **including** LDG and Fennel.

Subsequently, we produce 4 separate k-way partitionings of this ordered graph stream, using each of the following partitioning approaches for comparison:

- **Hash:** a naive partitioner which assigns vertices and edges to partitions on the basis of a hash function. This is the default partitioner used by many existing distributed graph databases and graph data management systems, including TitanDB [114], Microsoft's Trinity [108] and Facebook's TAO [122]. As such we use it as a baseline for our comparisons.

- **LDG:** a simple graph stream partitioner with good performance which we extend with our work on Loom.
- **Fennel:** a state-of-the-art graph stream partitioner and our primary point of comparison. As suggested by Tsourakakis et al [117], we use the Fennel parameter value $\gamma = 1.5$ throughout our evaluation.
- **Loom:** our own partitioner which, unless otherwise stated, we invoke with a window size of 10k edges and a motif support threshold of 40%.

Finally, when each graph is finished being partitioned, we execute the appropriate query workload over it and count the number of inter-partition traversals (*ipt*) which occur.

Note that, as with the evaluation of *TAPER* (Sec. 4.6), we avoid implementation dependent measures of partitioning quality because, as an isolated prototype, Loom is unlikely to exhibit realistic performance. For instance, lacking a distributed query processing engine, query workloads are executed over logical partitions during the evaluation. In the absence of network latency, query response times are meaningless as a measure of partitioning quality.

All algorithms, data structures, datasets and query workloads are publicly available[12]. All our experiments are performed on a commodity machine with a 3.1Ghz Intel i7 CPU and 16GB of RAM.

5.5.1.1 Graph datasets

Remember that the workload-agnostic partitioners which we aim to supersede with Loom are liable to exhibit poor workload performance when queries focus on traversing a limited subset of edge types (Sec. 5.1). Intuitively, such skewed workloads are more likely over heterogeneous graphs, where there exist a larger number of possible

[12]The Loom repository: `http://bit.ly/2eJxQcp`

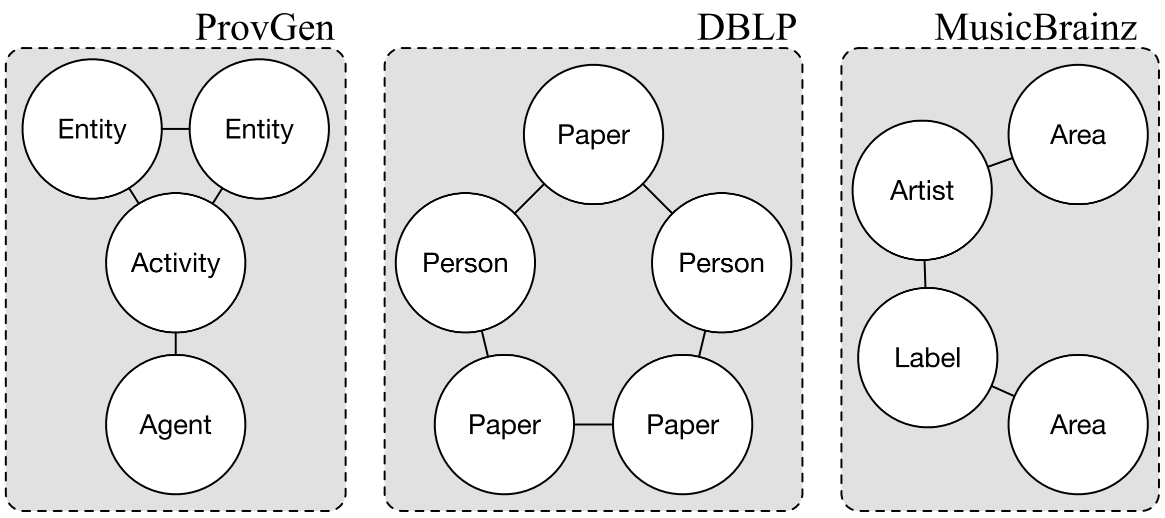


Figure 5.6: Examples of q for MusicBrainz, DBLP & ProvGen

edge types for queries to discern between, e.g. *a-a*, *a-b*, *a-c*. . . vs just *a-a*. Thus, we have chosen to test the Loom partitioner over five datasets with a range of different heterogeneities and sizes; three of these datasets are synthetic and two are real-world. Table 5.1 presents information about each of our chosen datasets, including their size and how heterogeneous they are ($|L_V|$). We use the DBLP [63], and LUBM [46] datasets, which are well known. MusicBrainz [111][13] is a freely available database of curated music metadata, with vertex labels such as *Artist*, *Country*, *Album* and *Label*. ProvGen [35] is a synthetic generator for PROV metadata [85], which records detailed provenance for digital artifacts.

5.5.1.2 Query workloads

For each dataset we must propose a representative query workload to execute so that we may measure partitioning quality in terms of *ipt*. Remember that a query workload consists of a set of distinct query patterns along with a frequency for each (Sec. 2.2). The LUBM dataset provides a set of query patterns which we make use of. For every other dataset, however, we define a small set of common-sense queries which focus on discovering implicit relationships in the graph, such as potential collaboration between authors or artists [14]. The full details of these query patterns are available[12], Fig. 5.6

[13]The MusicBrainz database: `http://bit.ly/1J0wlNR`

[14]If possible, workloads are drawn from the literature, e.g. common PROV queries [25, 57] and DBLP author-nets [116]

Table 5.1: Graph datasets, incl. size & heterogeneity

Dataset	$\sim \lvert V \rvert$	$\sim \lvert E \rvert$	$\lvert L_V \rvert$	Real	Description
DBLP [63]	1.2M	2.5M	8	Y	Publications & citations
ProvGen [35]	0.5M	0.9M	3	N	Wiki page provenance
MusicBrainz [111]	31M	100M	12	Y	Music records metadata
LUBM-100 [46]	2.6M	11M	15	N	University records
LUBM-4000 [46]	131M	534M	15	N	University records

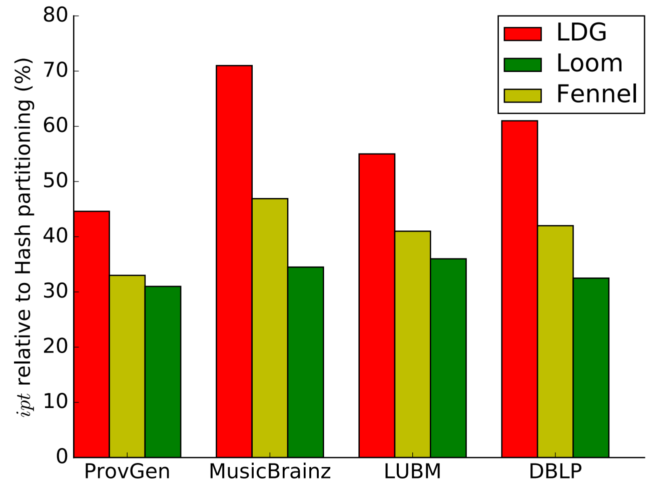
presents some examples. Note that whilst the TPSTry++ may be trivially updated to account for change in the frequencies of workload queries (Sec. 5.2), our evaluation of Loom assumes that said frequencies are fixed and known *a priori*. Recall that, for online databases, we argue this is a realistic assumption (Sec. 5.1). However, more complete tests with changing workloads are an important area for future work.

5.5.2 Comparison of systems

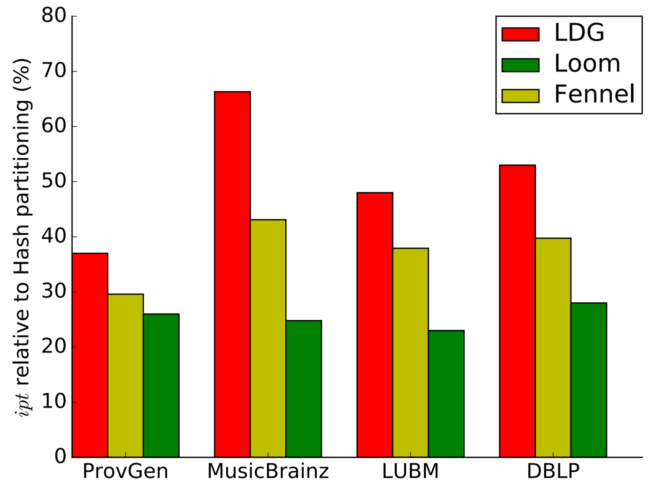
Figures 5.7 and 5.8 present the improvement in partitioning quality achieved by Loom and each of the comparable systems we desribe above. Initially, consider the experiment depicted in Fig. 5.7. We partition ordered streams of each of our first 4 graph datasets[15] into 8-way partitionings, using the approaches described above, then execute each dataset's query workload over the appropriate partitioning. The absolute number of inter-partition traversals (*ipt*) suffered when querying each dataset varies significantly. Thus, rather than represent these results directly, in Fig. 5.7 (and 5.8) we present the results for each approach as **relative** to the results for Hash; i.e. how many *ipt* did a partitioning suffer, **as a percentage** of those suffered by the Hash partitioning of the same dataset.

As expected, the naive hash partitioner performs poorly: it produces partitionings which suffer twice as many inter-partition traversals, on average, when compared to partitionings produced by the next best system (LDG). Whilst the LDG partitioner does achieve around a 55% reduction in *ipt* vs our Hash baseline, its produces partitionings of consistently poorer quality than those of Fennel and Loom. Although both LDG and Fennel optimise their partitionings for the balanced min. edge-cut goal (Sec. 5.1), Fennel is the more effective heuristic, cutting around 25% fewer edges than

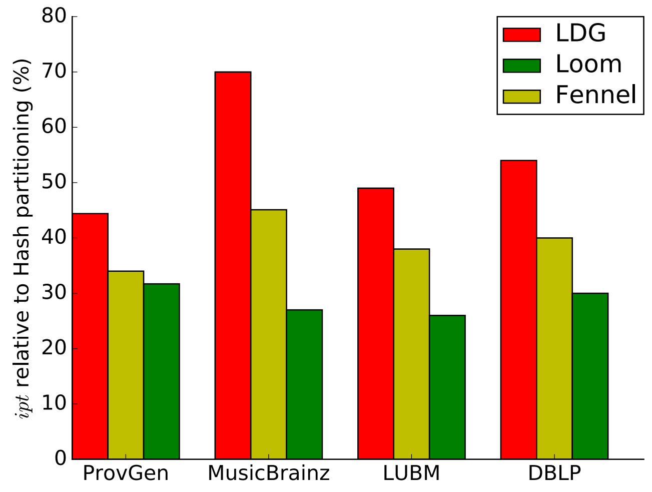
[15]Excluding LUBM-4000



(a) Random order

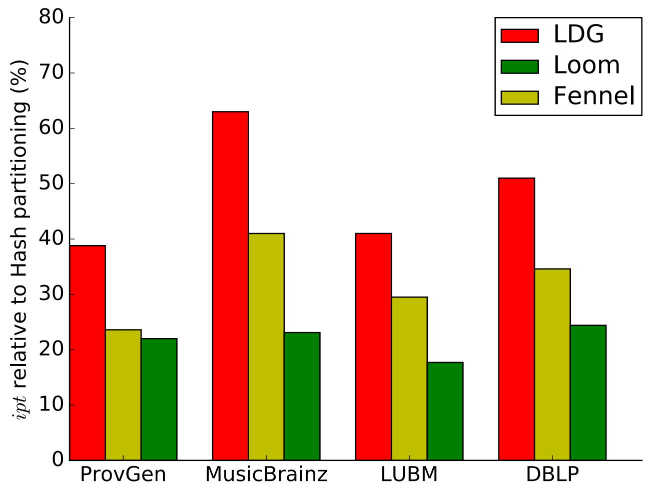


(b) Breadth-first order

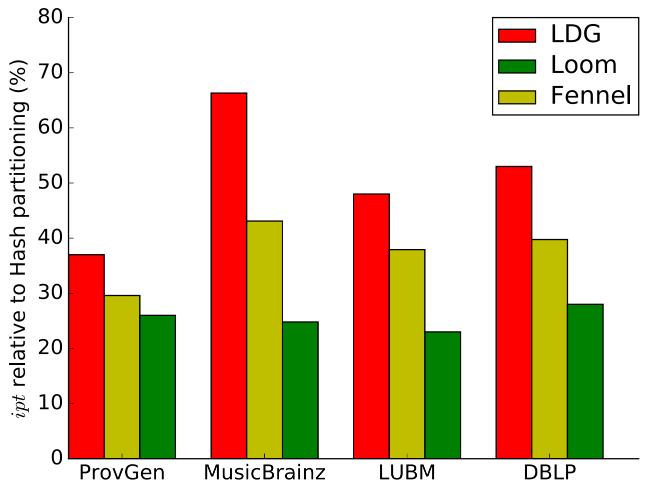


(c) Depth-first order

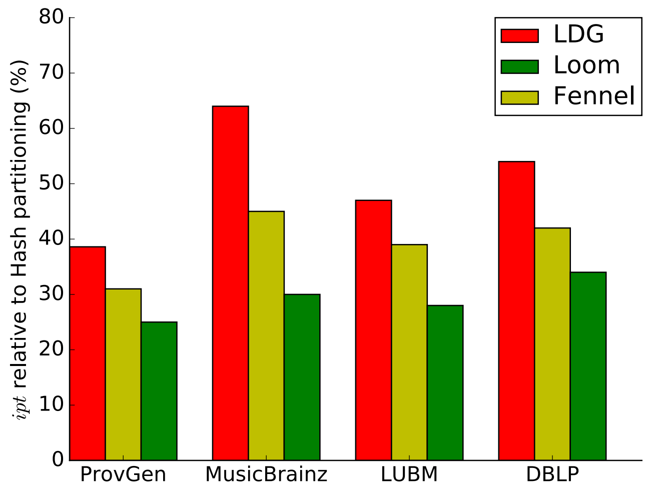
Figure 5.7: *ipt* %, vs. Hash, when executing Q over **8-way** partitionings of graph streams in multiple orders.



(a) $k = 2$



(b) $k = 8$



(c) $k = 32$

Figure 5.8: *ipt* %, vs. Hash, when executing Q over multiple k-way partitionings of **breadth-first** graph streams.

LDG for small numbers of partitions (including $k = 8$) [117]. Intuitively, the likelihood of *any* edge being cut is a coarse proxy for the likelihood of a query $q \in Q$ traversing a cut edge. This explains the disparity in *ipt* scores between the two systems.

Of more interest is comparing the quality of partitionings produced by Fennel and Loom. Fig. 5.7 clearly demonstrates that Loom offers a significant improvement in partitioning quality over Fennel, given a workload Q. Loom's reduction in *ipt* relative to Fennel's is present across all datasets and stream orders, however it is particularly pronounced over ordered streams of more heterogeneous graphs; e.g. MusicBrainz in subfigure 5.8b, where Loom's partitioning suffers from 42% fewer *ipt* than Fennel's. This makes sense because, as mentioned, pattern matching workloads are more likely to exhibit skew over heterogeneous graphs, where query graphs G_q contain a, potentially small, subset of the possible vertex labels. Across all the experiments presented in Fig. 5.7, the median range of Loom's *ipt* reduction relative to Fennel's is $20 - 25\%$.

Additionally, Fig. 5.8 demonstrates that this improvement is consistent for different numbers of partitions. As the number of partitions k grows, there is a higher probability that vertices belonging to a motif match are assigned across multiple partitions. This results in an increase of *absolute ipt* when executing Q over a Loom partitioning. However, increasing k actually increases the probability that any two vertices which share an edge are split between partitions, thus reducing the quality of Hash, LDG and Fennel partitionings as well. As a result, the difference in *relative ipt* is largely consistent between all 4 systems.

On the other hand, neither Fig. 5.7, nor Fig. 5.8, present the runtime costs of producing a partitioning. Table 5.2 presents how long (in ms) each partitioner takes to partition 10k edges. Whilst all 3 algorithms are capable of partitioning many 10s of thousands of edges per second, we do find that Loom is slower than LDG and Fennel by an average factor of 2-3. This is likely due to the more complex map-lookup and pattern-matching logic performed by Loom, or a nascent implementation. The runtime performance of Loom varies depending on the query workload Q used to generate the TPSTry++ (Sec. 5.2), therefore the performance figures presented in Table 5.2 are averaged across many different Q. The minimum slowdown factor observed between Loom and Fennel

Table 5.2: Time to partition 10k edges

Dataset	LDG (ms)	Fennel (ms)	Loom (ms)	Hash (ms)
DBLP	91	96	235	28
ProvGen	144	146	240	33
MusicBrainz	48	52	129	18
LUBM-100	47	51	147	22
LUBM-4000	45	49	138	16

was 1.5, the maximum 7.1. Note that popular non-streaming partitioner METIS [58] is around 13 times slower than Fennel for large graphs [117].

We contend that this performance difference is unlikely to be an issue in an online setting for two reasons. Firstly, most production databases do not support more than around 10k transactions per second (TPS) [68]. Secondly, it is considered exceptional for even applications such as twitter to experience >30k-40k TPS [118]. Meanwhile, the lowest partitioning rate exhibited by Loom in Table 5.2 is equivalent to ˜ 42k edges per second, the highest 72k.

Note that Figures 5.7 and 5.8 do not present the relative *ipt* figures for the LUBM-4000 dataset. This is because measuring relative *ipt* involves reading a partitioned graph into memory, which is beyond the constraints of our present experimental setup. However, we include the LUBM-4000 dataset in Table 5.2 to demonstrate that, as a streaming system, Loom is capable of partitioning large scale graphs. Also note that none of the figures present partitioning imbalance as this is broadly similar between all approaches and datasets [16], with LDG varying between $1\% - 3\%$, Loom and Fennel between 7% and their maximum imbalance of 10% (Sec. 5.4).

5.5.3 Effect of stream order and window size

Fig. 5.7 indicates that Loom is sensitive to the ordering of its given graph stream. In fact, subfigure 5.7a shows Loom achieve a smaller reduction in *ipt* over Fennel and LDG, than in 5.7b and 5.7c. Specifically Loom achieves a 42% greater reduction in relative *ipt* than Fennel given a breadth-first stream of the MusicBrainz graph, but only a 26% when the stream is ordered randomly, despite Fennel and LDG also

[16]Except Hash, which is balanced.

being sensitive to stream ordering [110, 117]. This implies that Loom is particularly sensitive to random orderings: edges which are close to one another in the graph may not be close in the graph stream, resulting in Loom detecting fewer motif matching sub-graphs in its stream window.

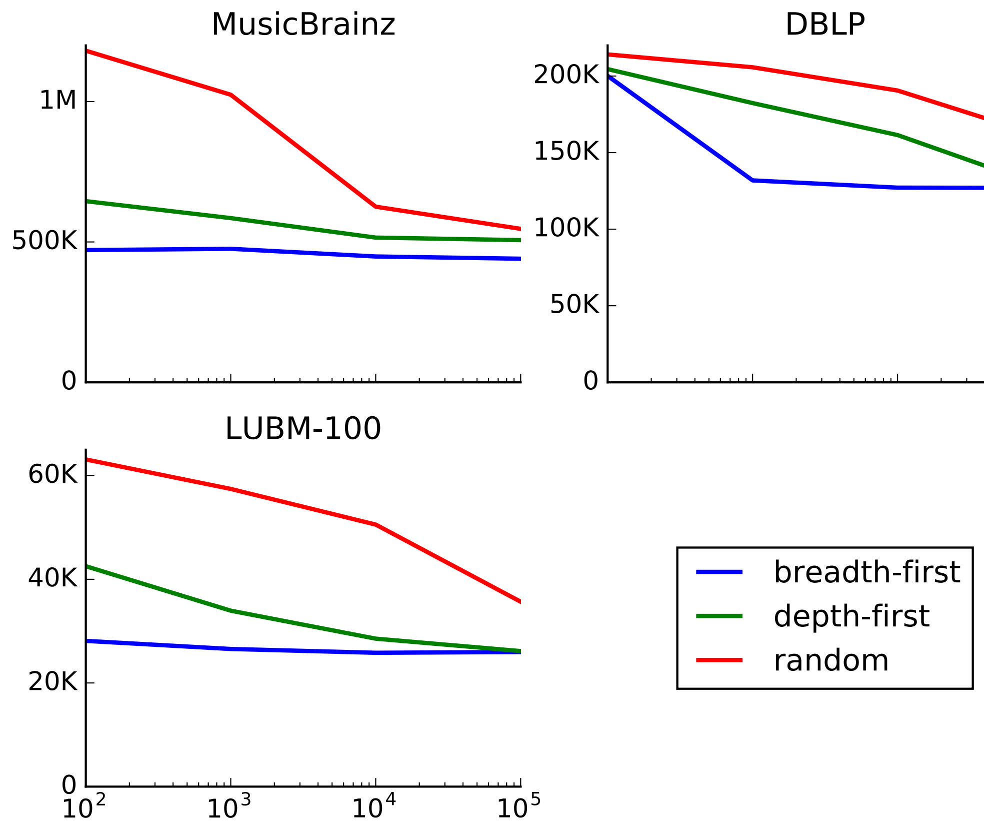


Figure 5.9: *ipt* (*y-axis*) when executing Q over Loom partitionings with multiple window sizes t(*x-axis*)

Intuitively, this sensitivity can be ameliorated by increasing the size of Loom's window, as shown in Fig. 5.9 As Loom's window grows, so does the probability that clusters of motif matching sub-graphs will occur within it. This allows Loom's equal opportunism heuristic to make the best possible allocation decisions for the sub-graph's constituent vertices. Indeed, the number of *ipt* suffered by Loom partitionings improves significantly, by as much as 47%, as the window size grows from 100 to 10k. However, increasing the window size past 10k clearly has little effect on *ipt* suffered to execute Q if your graph stream is ordered. The exact impact of increasing Loom's window size depends upon the degree distribution of the graph being partitioned. However, to gain an intuition consider the naive case of a graph with a uniform average vertex degree of 8, along with a TPSTry++ whose largest motif contains 4 edges. In this case, a breadth-first traversal of 8^4 edges from a vertex a (i.e. window size $t \approx 4k$) is highly likely to include all the motif matches which contain a. Regardless, Fig. 5.9 might seem to suggest that Loom should run with the largest window size possible. However, besides the additional computational cost of detecting more motif matches, remember that Loom's window constitutes a temporary partition (Sec. 5.3). If there exist many edges between other partitions and P_{temp}, then this may itself be a source of *ipt* and poor query performance.

5.6 Conclusion

In this chapter, we have presented Loom: a practical system for producing k-way partitionings of online, dynamic graphs, which are optimised for a given workload of pattern matching queries Q.

By employing frequent sub-graph mining in the form of the TPSTry++ (Sec. 5.2), we are able to identy sub-graphs which are common to many of Q's query graphs: *motifs*. Using an incremental and probabilistic form of pattern matching over the stream of graph updates, we are able to efficiently detect matches for these motifs as they form. Subsquently, we are able to heuristically assign the majority of motif matches within single partitions, reducing *ipt* when executing a random $q \in Q$.

We also argue that a TPSTry++ may be used as a graph index. In this case, detecting motif matches in the stream of graph updates is equivalent to an index update operation, amortizing Loom's computational cost in production systems which already rely on indices.

Our experiments indicate that Loom significantly improves graph partitioning quality with respect to Q, relative to state-of-the-art (workload agnostic) streaming graph partitioners. Furthermore they demonstrate that although Loom's partitioning does cause computational overhead relative to other approaches, its is more than performant enough for application in online data-management contexts.

Consider once again our comparison framework for graph partitioners (Sec. 3.6), summarised in Table. 5.3. Our experiments demonstrate that Loom is both workload-

Table 5.3: Comparison framework properties overview

Key	**Description**
S	Capable of partitioning graph streams
D	Capable of partitioning distributed graphs
SC	Capable of partitioning very large graphs
DY	Capable of partitioning dynamic, growing graphs
RF	Partitionings do not rely upon replication
WS	Partitionings are sensitive to given workloads
OW	Partitionings are sensitive to changing workloads
TX	Partitionings are sensitive to changing data-management workloads

Table 5.4: Properties of the Loom partitioner

System	**S**	**D**	**SC**	**DY**	**RF**	**WS**	**OW**	**TX**
Loom	Y		Y	Y	Y	Y		Y

sensitive (**WS**) and, unlike *TAPER*, capable of partitioning dynamic, growing graphs (**DY**). Table. 5.4 presents all Loom's framework properties; note that it possesses more of these properties, which render a partitioner suitable for addressing our thesis aim, than any other system we consider (Sec. 3.6.2 & 4.7).

Naturally, in spite of its effectiveness, there are several areas in which Loom could benefit from further work. Firstly, as a workload-sensitive technique which may not adapt existing partitionings, Loom is vulnerable to workload change over time (i.e. it lacks the **OW** property). Secondly, due to our reliance upon graph pattern matching in a single stream window, Loom is single threaded. This limits both the distributability and maximum throughput of the technique. We briefly outline the details of future research on several topics, including addressing these shortcomings, in the next chapter (Sec. 6.3).

6

Discussion

Contents

6.1 Thesis Summary

In Chapter 1, we argued that graph partitioning is an important problem due to its potential to improve the network latency, and therefore performance, of many modern applications which rely upon graph data operations. Broadly speaking, the work throughout the rest of the thesis followed one of two threads motivated by this problems' relevance. **Either** demonstrating that existing graph partitioning systems are poorly suited to improving the performance of the subset of graph operations common to online, data-management applications. **Or** proposing, implementing and evaluating a series of techniques which may be combined to produce graph partitionings of high quality with respect to said operation subset, both from scratch and iteratively from previously partitioned graphs. More specifically:

In Chapter 2 We presented background information on areas related to or depended upon by the main topic of this thesis, including formal definitions of graphs, graph workloads and graph partitionings. Additionally, several measures for the quality of a graph partitioning were defined and discussed, including *min. edge-cut* and *partition stability* [24]. Finally, we reiterated our argument from chapter 1: that the number of inter-partition traversals (*ipt*) is an ideal measure of partitioning quality with respect to a given online, data-management focused, graph workload.

In Chapter 3 We presented a thorough survey of existing graph partitioning research and systems, including both fundamental and state-of-the-art work. In this survey, we separated each piece of work into one of 10 potentially overlapping categories. For each such category, we provided a brief explanation of how graph partitionings are produced, along with a critical analysis of category systems' strengths and weaknesses. Finally we extracted 8 distinct properties of the surveyed systems, arguing why each property is important to consider when designing techniques to produce high quality partitionings of online graphs with respect to online workloads: the stated aim of this thesis. These properties constitute a comparison framework which we later used to evaluate our own partitioning techniques against the thesis' aim.

In Chapter 4 We proposed *TAPER*: a practical framework for workload-sensitive graph repartitioning. Firstly, given a stream of regular path queries [81] (RPQs) Q, we described how to construct and maintain a Traversal Pattern Summary Trie (TPSTry), which encodes the likelihood of any given path of labelled vertices being traversed by a random $q \in Q$. Using the TPSTry, *TAPER* is able to discover vertices likely to be the source of costly *ipt* given Q, which we refer to as having high *extroversion*. These *extroverted* vertices are then iteratively traded amongst partitions, attempting to collocate vertices which are often traversed together, thereby reducing *ipt*. As *TAPER* is intended for application to large, distributed graphs, we also described heuristics used to avoid computing all but the most highly extroverted vertices. These heuristics improved framework performance and, importantly, network overhead. Indeed, in the evaluation of this chapter (Sec. 4.6) we demonstrated that *TAPER* is able to approach or even improve the quality of a partitioning produced by a state of the art offline, global system (METIS [58]) with much less than half the network overhead. Finally, as an incremental *workload-sensitive* graph repartitioning system, *TAPER* is able to repartition its own output in order to adapt to the continuous workload change common in online data-management applications. To this end, we presented a further evaluation of the framework, demonstrating consistently high partitioning quality over time given a changing stream of queries and repeated executions of *TAPER*.

In Chapter 5 We propsed *Loom*: another practical system, this time for workload-senstive partitioning of online, dynamic graphs. Loom extends *TAPER*'s TPSTry using a novel method of sub-graph isomorphism checking, deriving the graph patterns (motifs) most frequently traversed by a given a workload of general pattern matching queries Q. Subsequently, the same isomorphism checking method is used to efficiently detect motif matching sub-graphs as they form in the graph stream. *Loom* attempts to assign these matches wholly within single partitions so that queries which cause the corresponding pattern of traversals may later be executed without increasing *ipt*. We conducted a thorough experimental evaluation of Loom using several large datasets and query workloads. These experiments indicated that Loom significantly reduces *ipt* relative to state of the art (but not *workload-sensitive*) streaming partitioners [110, 117]. Finally, we demonstrated that the TPSTry++ is equivalent to a graph feature index [123, 130] and that, therefore, the motif matches computed by *Loom* may be used as index entries. This potentially offsets the computational overhead of *Loom* in online data-management applications where indexing is a common existing requirement.

6.2 Summary of contributions

Recall that the aim of this thesis was to **Design, implement and evaluate techniques for producing partitionings of large graphs which are well optimised for use in an online data-management context**. Subsequently, this was separated into four, more concrete research questions (Sec. 1.3), each of which should be answered in any attempt to effectively address the high-level aim. Throughout this section we reflect upon the extent to which our own work satisfies these questions, along with its impacts and shortcomings in general.

6.2.1 Properties desirable for online graph partitioning techniques

The extensive research survey presented in chapter 3 revealed that no single existing graph partitioner is well suited to the workload-sensitive partitioning of online graphs. More recent systems such as Leopard [52] or the RDF partitioner of Peng et al [91] fare the best but lack key features, such as the ability to practically account for query workloads or partition an online graph (graph stream), respectively. This sort of critical analysis revealed several properties, besides workload sensitivity (**WS**) and stream partitioning (**S**), which are important to systems attempting to address the thesis aim. As mentioned, the survey highlights 8 such properties to form a framework for consistently comparing the functionality of different graph partitioners. Refer to Table 5.3 in the previous chapter for an overview. However, for clarity, the comparison framework only uses those properties which are relevant to the thesis aim, thereby restricting the surveys applicability relative to general pre-existing works [5, 12, 13, 106].

On the other hand, to our knowledge, existing surveys do not consider the fitness of graph partitioners at optimising **specifically** for online workloads; certainly not to the extent that they provide a comparison framework for consistently measuring said fitness. We therefore contend that our survey has value, not only for evaluating our own work thus far, but also the future work of ourselves and others on this topic.

6.2.2 Capturing online query workload information

The choice of approach to capturing and encoding information about a graph query workload was informed by some key insights. Firstly, it is impractical to track an online graph query workload directly, let alone the traversals caused by a workload [97, 129]. Any attempt at such granular tracking would significantly impinge upon the performance of the queries themselves, running counter to this thesis' ultimate aim (Sec. 3.5). Secondly, given a workload which changes over time, encodings need to be easily adaptable to avoid overfitting (Sec. 4.6.2). We have demonstrated that as intensional representations of a query workload's traversals, both the TPSTry and TPSTry++ are highly compact and simple to construct and update (Sec. 4.4 & 5.2).

They are not without their drawbacks. For instance, **as** intensional representations of workload traversal patterns, neither datastructure encodes the traversal likelihoods of individual vertices and edges, only collections of sub-graphs whose structure and labels match a given pattern.

Furthermore, the TPSTry is only able to encode simple paths of traversals and cannot be used with a workload of sub-graph pattern matching queries, which may contain branches, cycles and other features. The TPSTry++ **can** encode complex sub-graphs, but only those which are concretely specified. In other words, unlike the TPSTry, it cannot currently encode traversals arising from queries which make use of operators like exclusive disjunction "|" and the Kleene closure "*".

Despite these drawbacks, we believe that both the TPSTry and TPSTry++ datastructures are valuable foundational contributions towards the aim of this thesis, both for their attractive space-efficiency and for the *TAPER* and *Loom* partitioning techniques they enable.

6.2.3 Workload-sensitive re-partitioning of existing graphs for online workloads

In applications where workload-sensitive partitioners could usefully be applied, there is often a large existing graph partitioning (Sec. 1.3). Furthermore, whilst a workload-sensitive graph partitioning may be high quality with respect to a given query workload, that quality will degrade in the event that the workload varies over time, which is common in online data-management applications (Sec. 4.6). The *TAPER* system was designed as an efficient and repeatable workload-sensitive graph repartitioner in order to address these issues. It constitutes a key tangible contribution towards the aim of our thesis.

Unfortunately, as its partitioning algorithm is based upon the TPSTry, *TAPER* shares that datastructure's limitation to simple paths of vertices: it is only able to account for traversal likelihoods arising from workloads of RPQs in its partitionings. Futhermore, because the *TAPER* algorithm employs no real hill climbing, it is liable to get trapped in local optimisation minima when given a naive input, rather than achieve the highest possible quality of partitioning (Sec. 4.6).

On the other hand, as previously mentioned, we have demonstrated that *TAPER* is able to provide **significant** improvements in partitioning quality, given inputs generated by a) a naive graph partitioner; b) a sophisticated workload-agnostic system; or c) itself, prior to some workload change. Furthermore, **because** *TAPER* performs no hill-climbing, it will never perform a vertex-swap which results in a net decrease in partitioning quality. This renders the repartitioning process interruptible, and particularly well suited for use in an online graph data-management application: repartitioning can be paused in periods of high system load.

6.2.4 Workload-sensitive partitionings of online, growing graphs

Online dynamic graphs grow continuously over time, adding new vertices and edges. Broadly, there are two possible approaches to partitioning such a graph: periodically re-executing a global partitioner over the entire graph, including the new elements, or using a streaming graph partitioner [52, 87, 110, 117] which assigns new elements to partitions as they arrive[1]. As the former is impractical in an online application [55], a streaming graph partitioner was required. To the best of our knowledge, no **workload-sensitive** streaming previously existed, so we designed and implemented *Loom*.

Loom comes with a performance overhead that impacts system throughput. Specifically, it can only partition around 42% as many vertices and edges as state-of-the-art workload agnostic alternatives, in the same time. We have observed, however, that this is not an important limitation, because *Loom* can comfortably sustain a throughput that is much higher than the typical vertex arrival rate (Sec. 5.5.2). Additionally, like other streaming graph partitioners [52, 87, 110, 117], the quality of *Loom*'s partitioning is sensitive to the order of the graph update stream. However, this may be at least partially ameliorated by increasing the size of the window in which motif matches are detected (Sec. 5.5.3).

In spite of these issues, *Loom* is a one-of-a-kind system, capable of producing graph stream partitionings in a single pass which are of significantly higher quality, compared to those produced by existing workload-agnostic alternatives (Sec. 5.5).

In final summary, in order to address the aim of this thesis:

- We undertook significant research evaluating existing graph partitioning techniques, highlighting those lacking features which render them unfit to address our aim themselves.

- We presented a foundational contribution in the form of two novel datastructures (TPSTry and TPSTry++) for capturing and encoding the likely traversal patterns arising from a graph query workload typical of online data-management applications.

- We built upon the TPSTries to propose *Loom* and *TAPER*: two practical techniques for creating and maintaining graph partitionings. We used the partitioner features highlighted in early research to ensure that the techniques are well optimised for use in an online data-management context.

[1]A naive hash based approach can be seen as a workload-agnostic streaming graph partitioner.

6.3 Future research directions

In this section we describe several interesting topics for future research which build upon or extend the work of this thesis.

6.3.1 Integrating TAPER and Loom

Readers may note that we have not presented a **single** output which is capable of both creating and maintaining a high quality workload-sensitive partitioning of an online graph. *Loom* creates high quality but fixed partitionings from a dynamic stream of graph updates, whilst *TAPER* continuously improves the quality of pre-existing graph partitionings. Table 6.1 presents this distinction in terms of the comparison framework from chapter 3.

Table 6.1: Properties of *TAPER* and *Loom*

System	**S**	**D**	**SC**	**DY**	**RF**	**WS**	**OW**	**TX**
TAPER [37]		Y	P		Y	Y	Y	P
Loom [36, 38]	Y		Y	Y	Y	Y		Y

As a strict streaming graph partitioner, *Loom* will never reassign vertices and edges which have previously been assigned to a given partition. As a result, it lacks the **OW** property: the ability to adapt its partitioning to account for change in the makeup or frequencies of queries in an online workload. Meanwhile, as a repartitioning system *TAPER* cannot incrementally create a new partitioning from a stream of graph updates and therefore lacks the Dynamic (**DY**) property.

Between them, the two systems possess **every property** which we have identified as useful to addressing the aim of this thesis; evidently, it is desirable to integrate them. This would require reformulating *TAPER* in terms of the TPSTry++ in order to allow it to account for general sub-graph pattern matching queries. In particular, the algorithm for computing Visitor Matrix values (Sec. 4.5.4) would need to be replaced as, in its current form, it depends strongly on the linear structure of a graph path.

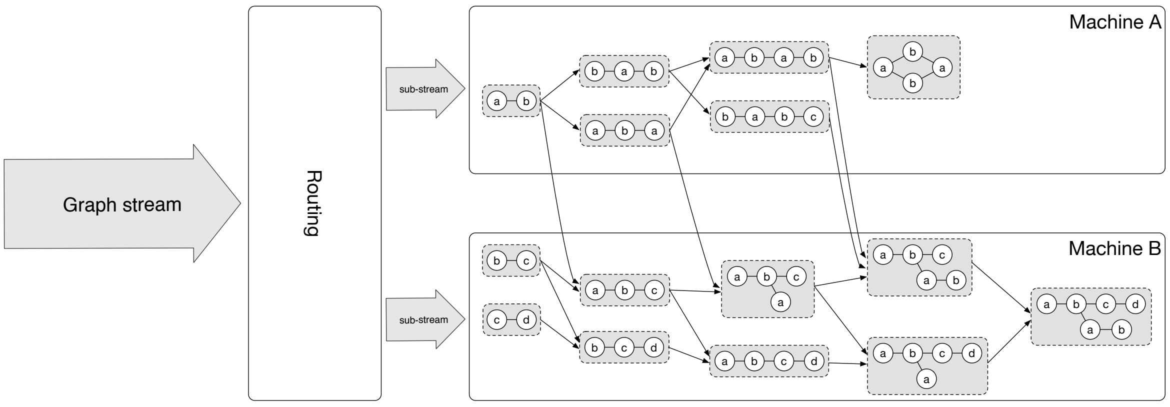


Figure 6.1: Potential architecture for a distributed variant of *Loom*

6.3.2 Distributing Loom across multiple hosts

As with other single-pass[2] partitioners [110, 117], it is not possible for *Loom* to operate whilst distributed amongst several hosts. Consider table 6.1 again: *Loom* lacks support for the Distributed property (**D**). This is due to the fact that *Loom* relies upon maintaining a single *matchList* for detecting motif matching sub-graphs within a graph update stream (Sec. 5.3). If *Loom* were distributed and connected sub-graphs were added to different *matchList*s, motif matches might go undetected.

In general, the lack of distributability is a performance bottleneck for streaming graph partitioners. As throughput is an, admittedly minor, concern for *Loom* (Sec. 5.5.2) it would be beneficial to address this.

One potential approach for distributing *Loom* across a cluster of machines would be to partition the TPSTry++ itself, as per figure 6.1. Each machine would be responsible for detecting matches for the motifs in its subset of the the TPSTry++. Meanwhile the entire cluster would be fronted by lightweight and routing middleware, assigning vertices and edges from the graph stream to the machine where they are most likely to form part of a motif match, based on their labels. The effectiveness and fine details of this approach are currently unknown, however.

[2]As opposed to restreaming partitioners [52, 87].

6.3.3 Restreaming with Loom

Recently, restreaming extensions have been proposed [52, 87] to the popular streaming algorithms LDG [110] and Fennel [117]. As outlined in chapter 3, these extensions depart from the general pattern of streaming graph partitioners by reapplying their partitioning algorithm to some or all of the already partitioned graph. Obviously, this technique has a performance cost relative to traditional single-pass systems, however it also produces substantial improvements in partitioning quality. For example, Leopard [52] reports a reduction in *min. edge-cut*[3] of between 10-70% relative to Fennel [117], which is the algorithm it extends.

It is certainly possible that a restreaming extension to *Loom* would provide a similar benefit to partitioning quality, and is therefore worthy of exploration.

6.3.4 A timeseries approach to triggering TAPER repartitionings

The evaluation of *TAPER* demonstrated that repeated invocations of its algorithm are able to maintain high partitioning quality (low numbers of *ipt*) with respect to a changing query workload (Sec. 4.6.2.5). However, the same evaluation also highlights the importance of effective trigger conditions for invocations. Indeed, naive trigger conditions can actually cause a reduction in partitioning quality by optimising a partitioning immediately prior to rapid workload change.

One potential avenue for avoiding this issue is predicting workload change in advance, and planning *TAPER* invocations accordingly. Workload change may periodic, as in our experiments, the result of high-level application changes, or simple due to random variations. Predicting such change based upon historical workload information is a classic example of nonlinear timeseries analysis [56]. For example, assuming a workload was at least roughly periodic, we could employ sketch datastructures to derive a continuous approximation for this period [53]; using that to dictate TAPER's invocation schedule.

[3]The partitioning quality metric used in that work.

6.3.5 Considering other forms of graph data

Throughout this thesis, we have avoided considering some forms of graph data, such as edge-labelled and multi-graphs (Sec. 2.1). In large part, these omissions simply allow us to convey complex concepts with a greater degree of clarity and concision. Indeed, many of the techniques presented may be trivially extended to, e.g. edge-labelled or directed graphs, including both TPSTries. (Sec. 4.4, and 5.2.1).

However, there remain some areas of the thesis, notably *TAPER*'s Visitor Matrix (Sec. 4.2.3), where the impact of considering these other forms of graph data remains unexplored. This exploration, and any resulting extensions to the VM, are an important direction for future effort.

Bibliography

[1] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *Proceedings of the 13th International Conference on Extending Database Technology - EDBT '10*, page 287, 2010.

[2] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems*, 11(4):319–352, 1993.

[3] Konstantin Andreev and Harald Racke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[4] Renzo Angles and Claudio Gutierrez. Querying RDF Data from a Graph Database Perspective. In *The Semantic Web: Research and Applications, European Semantic Web Conference (ESWC)*, pages 346–360, 2005.

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

[6] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 286–296, 2004.

[7] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, pages 629–638. ACM, 2012.

[8] Pablo Barceló, Leonid Libkin, Anthony W Lin, and Peter T Wood. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Transactions on Database Systems (TODS)*, 37(4), 2012.

[9] Stephen T Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, 1995.

[10] Gaëlle Brevier, Romeo Rizzi, and Stéphane Vialette. Pattern Matching in Protein-Protein Interaction Graphs. In *Fundamentals of Computation Theory*, pages 137–148. 2007.

[11] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, 1998.

[12] Mike Buerli. The Current State of Graph Databases. *Department of Computer Science, Cal Poly San Luis Obispo*, 2012.

[13] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. 2016.

[14] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, pages 74–83, 2004.

[15] Jie Chen and Ilya Safro. Algebraic Distance on Graphs. *SIAM Journal on Scientific Computing*, 33(6):3468–3490, 2011.

[16] Lei Chen. Distance-Join : Pattern Match Query In a Large Graph. *Science And Technology*, 2(1):886–897, 2009.

[17] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318–331, 2008.

[18] C Chevalier and F Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318–331, 2008.

[19] Sutanay Choudhury, Lawrence B Holder, George Chin, Khushbu Agarwal, and John Feo. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. *Proceedings of the 18th International Conference on Extending Database Technology (EDBT)*, pages 157–168, 2015.

[20] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[21] D. J. de Solla Price. Networks of Scientific Papers. *Science*, 149(3683):510–515, 1965.

[22] Jeff Dean. Designs, lessons and advice from building large distributed systems. 2009.

[23] J-C Delvenne, S N Yaliraki, and M Barahona. Stability of graph communities across time scales. *Proceedings of the National Academy of Sciences of the United States of America*, 107(29):12755–60, 2010.

[24] Jean-charles Delvenne, Michael T Schaub, and Sophia N Yaliraki. The Stability of a Graph Partition: A Dynamics-Based Framework for Community Detection. In *Dynamics On and Of Complex Networks*, volume 2, pages 221–242. 2013.

[25] Saumen Dey, Víctor Cuevas-Vicenttín, Sven Köhler, Eric Gribkoff, Michael Wang, and Bertram Ludäscher. On implementing provenance-aware regular path queries with relational query engines. *Proceedings of the Joint EDBT/ICDT 2013 Workshops on - EDBT '13*, page 214, 2013.

[26] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12):1555–1581, 2000.

[27] Reinhard Diestel. *Graph theory {graduate texts in mathematics; 173}*. 2000.

[28] C.H.Q. Ding, X. He, H. Zha, M. Gu, and H.D. Simon. A min-max cult algorithm for graph partitioning and data clustering. In *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2001.

[29] W. E. Donath and a. J. Hoffman. Lower Bounds for the Partitioning of Graphs, 1973.

[30] Orri Erling and Ivan Mikhailov. Virtuoso: Rdf support in a native rdbms. In *Semantic Web Information Management*, pages 501–519. 2010.

[31] Facebook Inc. GraphQL Informal Specification. `http://facebook.github.io/graphql/October2016/`. Accessed: 2018-01-09.

[32] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment*, 3(1-2):1161–1172, 2010.

[33] C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Design Automation Conference*, 1982.

[34] Miroslav Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975.

[35] Hugo Firth and Paolo Missier. ProvGen: Generating Synthetic PROV Graphs with Predictable Structure. In *5th International Provenance and Annotation Workshop, IPAW*, pages 16–27, 2014.

[36] Hugo Firth and Paolo Missier. Workload-aware streaming graph partitioning. In *Proceedings of the Joint EDBT/ICDT Workshops (GraphQ)*, 2016.

[37] Hugo Firth and Paolo Missier. TAPER: query-aware, partition-enhancement for large, heterogenous graphs. *Distributed and Parallel Databases*, 35(2):85–115, 2017.

[38] Hugo Firth, Paolo Missier, and Jack Aiston. Loom: Query-aware Partitioning of Online Graphs. *Proceedings of the 21st International Conference on Extending Database Technology (EDBT)*, 2018.

[39] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the World-Wide Web. *ACM SIGMOD Record*, 27(3):59–74, 1998.

[40] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.

[41] R. G. Gallager, P. a. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[42] Giovanni Gallavotti. *Statistical mechanics: A short treatise*. 2013.

[43] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976.

[44] Gaurav Goel and Jens Gustedt. Bounded Arboricity to Determine the Local Structure of Sparse Graphs. In *Graph-Theoretic Concepts in Computer Science*, pages 159–167. 2006.

[45] Leonid Grujic, Ivana and Bogdanovic-Dinic, Sanja and Stoimenov. Collecting and analyzing data from e-government facebook pages. In *ICT Innovations*, 2014.

[46] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158 – 182, 2005. Selcted Papers from the International Semantic Web Conference, 2004.

[47] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. Real-time twitter recommendation. *Proceedings of the VLDB Endowment*, 7(13):1379–1380, 2014.

[48] Olaf Hartig and Jorge Perez. An initial analysis of facebook's graphql language. In *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web.*, volume 1912 of *CEUR Workshop Proceedings*, 2017.

[49] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, 2008.

[50] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '95*, pages 28–40, 1995.

[51] Bruce Hendrickson and Robert Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[52] Jiewen Huang and Daniel J. Abadi. LEOPARD : Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs. *Proceedings of the VLDB Endowment*, 9(7):540–551, 2016.

[53] Piotr Indyk, Nick Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 363–372, 2000.

[54] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1):75–105, 2013.

[55] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In *Enabling Real-Time Business Intelligence*, pages 65–80. 2012.

[56] Holger Kantz and Thomas Schreiber. *Nonlinear time series analysis*, volume 7. 2004.

[57] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 951–962, 2010.

[58] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[59] George Karypis and Vipin Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[60] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment*, 8(12):1566–1577, 2015.

[61] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell systems technical journal*, 49(2):291—-307, 1970.

[62] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*, page 169, 2013.

[63] Knowledge Discover Lab, UMass Amherst. Proximity dblp database. `https://kdl.cs.umass.edu/display/public/DBLP`. Accessed: 2015-09-15.

[64] Peter Korosec, Jurij Silc, and Borut Robic. Solving the mesh-partitioning problem with an ant-colony algorithm. *Parallel Computing*, 30(5-6):785–801, 2004.

[65] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web - WWW '10*, page 591, 2010.

[66] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 31–46, 2012.

[67] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255, 1950.

[68] S. Lee, B. Moon, C. Park, et al. A case for flash memory ssd in enterprise database applications. In *Proc. SIGMOD*, page 1075, 2008.

[69] Ce Charles E Leiserson, Rl Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to Algorithms, Third Edition.* 2009.

[70] HF Li and SY Lee. Mining Top-K Path Traversal Patterns over Streaming Web Click-Sequences. *Journal of Information Science and Engineering*, 1133(95):1121–1133, 2009.

[71] R Lidl and H Niederreiter. Finite Fields: Encyclopedia of Mathematics and Its Applications. *... & Mathematics with Applications*, page 1983, 1997.

[72] Lightbend Inc. Akka Distributed Application Framework. `https://akka.io`. Accessed: 2018-01-15.

[73] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Capturing topology in graph pattern matching. *Proceedings of the VLDB Endowment*, 5(4):310–321, 2011.

[74] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 135, 2010.

[75] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, pages 135–146, 2010.

[76] Daniel Margo and Margo Seltzer. A scalable distributed graph partitioner. *Proceedings of the VLDB Endowment*, 8(12):1478–1489, 2015.

[77] Erwan Le Martelot and Chris Hankin. Multi-scale community detection using stability optimisation within greedy algorithms. *CoRR*, abs/1201.3307, 2012.

[78] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking Like a Vertex. *ACM Computing Surveys*, 48(2):1–39, 2015.

[79] Brendan D McKay. Practical graph isomorphism, 1981.

[80] Kurt Mehlhorn and Peter Sanders. Algorithms and data structures: The basic toolbox. In *Algorithms and Data Structures: The Basic Toolbox*, chapter 10, pages 191–215. 2008.

[81] Alberto O Mendelzon and Peter T Wood. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[82] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. Graph partitioning and disturbed diffusion. *Parallel Computing*, 35(10-11):544–569, 2009.

[83] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel Graph Partitioning for Complex Networks. In *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pages 1055–1064, 2015.

[84] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 international conference on Management of Data*, pages 145–156, 2012.

[85] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV Data Model. Technical report, World Wide Web Consortium, 2012.

[86] Neo Technology Inc. Neo4j Graph Database. `https://neo4j.com`. Accessed: 2017-11-21.

[87] Joel Nishimura and Johan Ugander. Restreaming graph partitioning. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*, pages 1106–1114, 2013.

[88] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *World Wide Web Internet And Web Information Systems*, 54(1999-66):1–17, 1998.

[89] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 international conference on Management of Data*, page 61, 2012.

[90] François Pellegrini. A Parallelisable Multi-level Banded Diffusion Scheme for Computing Balanced Partitions with Smooth Boundaries. *Euro-Par 2007 Parallel Processing*, 4641:195–204, 2007.

[91] Peng Peng, Lei Zou, Lei Chen, and Dongyan Zhao. Query Workload-based RDF Graph Fragmentation and Allocation. In *Proc. 19th International Conference on Extending Database Technology (EDBT)*, pages 377—-388, 2016.

[92] A Pothen and S Toledo. Elimination structures in scientific computing. In *Handbook on Data Structures and Applications*, page 59. 2004.

[93] A. Poulovassilis, P. Selmer, and P. T. Wood. Flexible querying of lifelong learner metadata. *IEEE Transactions on Learning Technologies*, 5(2):117–129, 2012.

[94] Alexandra Poulovassilis and Peter T. Wood. Combining approximation and relaxation in semantic web path queries. In *The Semantic Web – ISWC 2010*, pages 631–646, 2010.

[95] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing Large Graphs on Multi-cores with Graph Awareness. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, page 4, 2012.

[96] Jm Pujol, Vijay Erramilli, and Georgos Siganos. The little engine (s) that could: scaling online social networks. *Acm Sigcomm'10*, pages 375–386, 2010.

[97] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD. In *Proceedings of the 16th International Conference on Extending Database Technology*, page 430, 2013.

[98] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. JA-BE-JA: A distributed algorithm for balanced graph Partitioning. In *International Conference on Self-Adaptive and Self-Organizing Systems, SASO*, pages 51–60, 2013.

[99] Pedro Ribeiro and Fernando Silva. G-Tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28(2):337–377, 2014.

[100] Marko A Rodriguez and Peter Neubauer. The Graph Traversal Pattern. In Sherif Sakr and Eric Pardede, editors, *Graph Data Management: Techniques and Applications*, chapter 2, pages 29–46. 2011.

[101] Laura A. Sanchis. Multiple-Way Network Partitioning with Different Cost Functions. *IEEE Transactions on Computers*, 42(12):1500–1504, 1993.

[102] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Lecture Notes in Computer Science*, volume 7933, pages 164–175. 2013.

[103] Stefan Schamberger. On partitioning FEM graphs using diffusion. In *Proceedings 18th International Parallel and Distributed Processing Symposium*, number C, pages 277–284, 2004.

[104] K. Schloegel, G. Karypis, and V. Kumar. A Unified Algorithm for Load-balancing Adaptive Scientific Simulations. *ACM/IEEE SC 2000 Conference (SC'00)*, 00(c):59–59, 2000.

[105] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, 1997.

[106] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for high-performance scientific simulations. In *Sourcebook of parallel computing*, pages 491–541. 2003.

[107] Zechao Shang and Jeffrey Xu Yu. Catch the Wind: Graph workload balancing on cloud. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 553–564, 2013.

[108] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, 2013.

[109] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. Event pattern matching over graph streams. *Proceedings of the VLDB Endowment*, 8(4):413–424, 2014.

[110] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2012.

[111] A. Swartz. Musicbrainz: a semantic web service. *IEEE Intelligent Systems*, 17(1):76–77, 2002.

[112] K Tashkova, P Korosec, and J Silc. A distributed multilevel ant-colony algorithm for the multi-way graph partitioning. *International Journal of Bio-Inspired Computation*, 3(5):286–296, 2011.

[113] The Apache Software Foundation. Apache Tinkerpop. `https://tinkerpop.apache.org`. Accessed: 2018-01-15.

[114] Think Aurelius. TitanDB Graph Database. `http://titan.thinkaurelius.com/`. Accessed: 2017-04-07.

[115] Lini T Thomas, Satyanarayana R Valluri, and Kamalakar Karlapalem. Margin: Maximal frequent subgraph mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):10, 2010.

[116] Hanghang Tong, Brian Gallagher, Christos Faloutsos, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 737, 2007.

[117] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014.

[118] Twitter Engineering. Tweets per second in 2013. `https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html`. Accessed: 2016-06-03.

[119] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[120] Luis M. Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings - International Conference on Distributed Computing Systems*, pages 144–153, 2014.

[121] A Vázquez, A Flammini, A Maritan, and A Vespignani. Modeling of Protein Interaction Networks. *ComPlexUs*, 1:38–44, 2003.

[122] Venkateshwaran Venkataramani, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*, page 791, 2012.

[123] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proceedings IEEE 23rd International Conference on Data Engineering*, pages 976–985, 2007.

[124] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, 2013.

[125] Ning Xu, Lei Chen, and Bin Cui. LogGP: A Log-based Dynamic Graph Partitioning Method. *Proceedings of the VLDB Endowment*, 7(14):1917–1928, 2014.

[126] Ning Xu, Bin Cui, Lei Chen, Zi Huang, and Yingxia Shao. Heterogeneous environment aware streaming graph partitioning. *IEEE Transactions on Knowledge and Data Engineering*, 27(6):1560–1572, 2015.

[127] Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[128] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346, 2004.

[129] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 international conference on Management of Data*, pages 517–528, 2012.

[130] Dayu Yuan and Prasenjit Mitra. Lindex: A lattice-based index for graph databases. *VLDB Journal*, 22(2):229–252, 2013.

[131] A. Zheng, A. Labrinidis, P. Pisciuneri, P. K. Chrysanthis, and P. Givi. Paragon: Parallel architecture-aware graph partition refinement algorithm. In *Proceedings of the 19th International Conference on Extending Database Technology (EDBT)*, pages 365–376, 2016.

[132] Angen Zheng, Alexandros Labrinidis, Panos Chrysanthis, and K. Architecture-aware graph repartitioning for data-intensive scientific computing. In *Proceedings - IEEE International Conference on Big Data*, pages 78–85, 2014.