Loom: Query-aware Partitioning of Online Graphs

Hugo Firth
School of Computing Science
h.firth@ncl.ac.uk

Paolo Missier
School of Computing Science
paolo.missier@ncl.ac.uk

Jack Aiston
School of Computing Science
j.aiston@ncl.ac.uk

ABSTRACT

As with general graph processing systems, partitioning data over a cluster of machines improves the scalability of graph database management systems. However, these systems will incur additional network cost during the execution of a query workload, due to inter-partition traversals. Workload-agnostic partitioning algorithms typically minimise the likelihood of any edge crossing partition boundaries. However, these partitioners are sub-optimal with respect to many workloads, especially queries, which may require more frequent traversal of specific subsets of inter-partition edges. Furthermore, they are largely unsuited to operating incrementally on dynamic, growing graphs.

We present a new graph partitioning algorithm, Loom, that operates on a stream of graph updates and continuously allocates the new vertices and edges to partitions, taking into account a query workload of graph pattern expressions 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 partitions of significantly better quality than existing streaming graph partitioning algorithms Fennel & LDG.

1 INTRODUCTION

Subgraph pattern matching is a class of operation fundamental to many "real-time" applications of graph data. For example, in social networks [9], and network security [3]. Answering a subgraph pattern matching query usually involves exploring the subgraphs of a large, labelled graph \( G \) then finding those which match a small labelled graph \( q \). Fig.1 shows an example graph \( G \) and a set of query graphs \( Q \) which we will refer to throughout.

**Efficiently partitioning large, growing graphs to optimise for given workloads of such queries** is the primary contribution of this work.

In specialised graph database management systems (GDBMS), pattern matching queries are highly efficient. They usually correspond to some index lookup and subsequent traversal of a small number of graph edges, where edge traversal is analogous to pointer dereferencing. However, as graphs like social networks may be both large and continually growing, eventually they saturate the memory of a single commodity machine and must be partitioned and distributed. In such a distributed setting, queries which require inter-partition traversals, such as \( q_2 \) in Fig. 1, incur network communication costs and will perform poorly. A widely recognised approach to addressing these scalability issues in graph data management is to use one of several \( k \)-way balanced graph partitioners [2, 11, 14, 18, 30–32]. These systems distribute vertices, edges and queries evenly across several machines, seeking to optimise some global goal, e.g. minimising the number of edges which connect vertices in different partitions (a.k.a min. edge-cut). In so doing, they improve the performance of a broad range of possible analyses.

Whilst graphs partitioned for such global measures mostly work well for global forms of graph analysis (e.g. Pagerank), no one measure is optimal for all types of operation [28]. In particular, the workloads of pattern matching query workloads, common to GDBMS, are a poor match for these kinds of partitioned graphs, which we call workload agnostic. This is because, intuitively, a min. edge-cut partitioning is equivalent to assuming uniform, or at least constant, likelihood of traversal for each edge throughout query processing. This assumption is unrealistic as a query workload may traverse a limited subset of edges and edge types, which is specific to its graph patterns and subject to change.

To appreciate the importance of a workload-sensitive partitioning, consider the graph of Fig.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 \( q_1 \) matches the subgraphs \((1,2), (2,3)\) and \((6,2), (2,3)\) in \( G \). Given a workload which consisted entirely of \( q_1 \) 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.

Mature research of workload-sensitive online database partitioning is largely confined to relational DBMS [4, 23, 26]

1.1 Contributions

Given the above motivation, we present Loom: a partitioner for online, dynamic graphs which optimises vertex placement to improve the performance of a given stream of sub-graph pattern matching queries.

The simple goals of Loom are threefold: a) to discover patterns of edge traversals which are common when answering queries...
from our given workload $Q$; b) to efficiently detect instances of
these patterns in the ongoing stream of graph updates which con-
stitutes an online graph; and c) to assign these pattern matches
wholly within an individual partition or across as few partitions
as possible, thereby reducing the number of $ipt$ and increasing
the average performance of any $q \epsilon Q$.

This work extends an earlier "vision" work [7] by the authors,
providing the following additional contributions:

- A compact trie-based encoding of the most frequent traversal
  patterns over edges in $G$. We show how it may
  be constructed and updated given an evolving workload $Q$.
- A method of sub-graph isomorphism checking, extending
  a recent probabilistic technique [29]. We show how this
  measure may be efficiently computed and demonstrate
  both the low probability of false positives and the impos-
sibility of false negatives.
- A method for efficiently computing matches for our fre-
  quent traversal patterns in a graph stream, using our trie
  encoding and isomorphism method, and then assign these
  matching sub-graphs to graph partitions, using heuris-
tics to preserve balance. Resulting partitions do not rely
  upon replication and are therefore agnostic to the complex
  replication schemes often used in production systems.

As online graphs are equivalent to graph streams, we present
an extensive evaluation comparing Loom to popular streaming
graph partitioners Fennel [31] and LDG [30]. We partition real
and synthetic graph streams of various sizes and with three dis-
tinct stream orderings: breadth-first, depth-first and random or-
der. 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.

### 1.2 Related work

Partitioning graphs into $k$ balanced subgraphs is clearly of
practical importance to any application with large amounts of
graph structured data. As a result, despite the fact that the prob-
lem is known to be NP-Hard [1], many different solutions have
been proposed [2, 11, 14, 18, 30–32]. We classify these partitioning
approaches into one of three potentially overlapping categories:
streaming, non-streaming and workload sensitive. Loom is both a
streaming and workload-sensitive partitioner.

**Non-streaming** graph partitioners [2, 14, 18] typically seek to
optimise an objective function global to the graph, e.g. minimising
the number of edges which connect vertices in different partitions
(*min. edge-cut*)

A common class of these techniques is known as multi-level
partitioners [2, 14]. These partitioners work by computing a suc-
cession of recursively compressed graphs, tracking exactly how
the graph was compressed at each step; then trivially partition-
ing the smallest graph with some existing technique. Using the
knowledge of how each compressed graph was produced from
the previous one, this initial partitioning is then "projected" back
onto the original graph, using a local refinement technique (such as
Kernighan-Lin [15]) to improve the partitioning after each step.
A well known example of a multilevel partitioner is METIS [14],
which is able to produce high quality partitionings for small and
medium graphs, but performance suffers significantly in the pres-
ence of large graphs [31]. Other multilevel techniques [2] share
broadly similar properties and performance, though they differ
in the method used to compress the graph being partitioned.

Other types of non-streaming partitioner include Sheep [18]:
A graph partitioner which creates an elimination tree from a
distributed graph using a map-reduce procedure, then partitions
the tree and subsequently translates it into a partitioning of
the original graph. Sheep optimises for another global objective
function: minimising the number of different partitions in which
a given vertex $v$ has neighbours (*min. communication volume*).

These non-streaming graph partitioners suffer from two main
drawbacks. Firstly, due to their computational complexity and
high memory usage [30], they are only suitable as offline opera-
tions, typically performed ahead of analytical workloads. Even
those partitioners which are distributed to improve scalability,
such as Sheep or the parallel implementation of METIS (ParMETIS)
[14], make strong assumptions about the availability of global
graph information. As a result they may require periodic re-
execution, i.e. given a dynamic graph following a series of graph
updates, which is impractical online [31]. Secondly, as mentioned,
partitioners which optimise for such global measures assume
uniform and constant usage of a graph, causing them to "leave
performance on the table" for many workloads.

**Streaming** graph partitioners [11, 30, 31] have been proposed
to address some of the problems with partitioners outlined above.
Firstly, the strict streaming model considers each element of a
graph stream as it arrives, efficiently assigning it to a partition.
Additionally, streaming partitioners do not perform any refine-
ment, i.e. later reassigning graph elements to other partitions,
nor do they perform any sort of global introspection, such as
spectral analysis. As a result, the memory usage of streaming
partitioners is both low and independent of the size of the graph
being partitioned, allowing streaming partitioners to scale to very
large graphs (e.g. billions of edges). Secondly, streaming
partitioners may trivially be applied to continuously growing
graphs, where each new edge or update is an element in the
stream.

Streaming partitioners, such as Fennel [31] and LDG [30],
make partition assignment decisions on the basis of inexpensive
heuristics which consider the local neighbourhood of each new
element at the time it arrives. For instance, LDG assigns vertices
to the partitions where they have the most neighbours, but pe-
nalises that number of neighbours for each partition by how full
it is, maintaining balance. By using the local neighbourhood of a
graph element $e$ at the time $e$ is added, such heuristics render
themselves sensitive to the ordering of a graph stream. For ex-
ample, a graph which is streamed in the order of a breadth-first
traversal of its edges will produce a better quality partitioning
than a graph which is streamed in random order, which has been
shown to be pseudo adversarial [31].

In general, streaming algorithms produce partitionings of
lower quality than their non-streaming counterparts but with
much improved performance. However, some systems, such as
the graph partitioner Leopard [11], attempt to strike a balance
between the two. Leopard relies upon a streaming algorithm (Fen-
nel) for the initial placement of vertices but drops the "one-pass"
requirement and repeatedly considers vertices for reassignment;
 improving quality over time for dynamic graphs, but at the cost
of some scalability. Note that these Streaming partitioners, like
their non-streaming counterparts, are workload agnostic and so
share those disadvantages.
Workload sensitive partitioners [4, 23, 24, 26, 28, 32] attempt to optimise the placement of data to suit a particular workload. Such systems may be streaming or non-streaming, but are discussed separately here because they pertain most closely to the work we do with Loom.

Some partitioners, such as LogGP [32] and CatchW [28], are focused on improving graph analytical workloads designed for the bulk synchronous parallel (BSP) model of computation\(^2\). 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 and proposes techniques for predicting the set of edges likely to be traversed in the next superstep, given the category of workload and edges traversed in the previous one. CatchW then moves a small number of these predicted edges between supersteps, minimising inter-partition communication. LogGP uses a similar log of activity from previous supersteps to construct a hypergraph where vertices which are frequently accessed together are connected. LogGP then partitions this hypergraph to suggest placement of vertices, reducing the analytical job’s execution time in future.

In the domain of RDF stores, Peng et al. [24] 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 at the cost of having to replicate (potentially many) sub-graphs which form part of multiple frequent patterns. Harbi et al. [10] also detect patterns common to workloads of SPARQL queries; their system, called AdPart, redistributes data between partitions over time such that more queries may be executed without ipt. However, like Peng et al.’s work, AdPart makes significant use of replication. Additionally, AdPart’s re-partitioning approach relies upon an initial input generated by a naive Hash partitioner. Thus, it could potentially be used in conjunction with a streaming system like Loom: a workload-aware initial partitioning reducing the amount of data redistribution required later.

For RDBMS, systems such as Schism [4] and SWORD [26] capture query workload samples ahead of time, modelling them as hypergraphs where edges correspond to sets of records which are involved in the same transaction. These graphs are then partitioned using existing non-streaming techniques (METIS) to achieve a min. edge-cut. When mapped back to the original graph, these partitionings represent an arrangement of records which causes a minimal number of transactions in the captured workload to be distributed. Other systems, such as Horticulture [23], 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. In addition to a high upfront cost [4, 23], these techniques focus on the relational data model, and so make simplifying assumptions, such as ignoring queries which traverse > 1-2 edges [26] (i.e. which perform nested joins). Larger traversals are common to sub-graph pattern matching queries, therefore it’s unclear how these techniques would perform given such a workload.

Overall, the works reviewed above either focus on different types of workload than we do with Loom (namely offline analytical or relational queries), or they make extensive use of replication. Loom does not use any form of replication, both to avoid potentially significant storage overheads [25] and to remain interoperable with the sophisticated replication schemes used in production systems.

1.3 Definitions

Here we review and define important concepts used throughout the rest of the paper.

A labelled graph \( G = (V, E, L_V, f_I) \) is of the form: a set of vertices \( V = \{v_1, v_2, ..., v_n\} \), a set of pairwise relationships called edges \( E = \{(v_i, v_j) \in E \) and a set of vertex labels \( L_V \). The function \( f_I : V \rightarrow L_V \) is a surjective mapping of vertices to labels. Note that throughout this work, for simplicity, we consider only undirected graphs. However, all techniques subsequently presented may be extended to directed graphs, which we highlight inline. We view an online graph simply as a (possibly infinite) sequence of edges which are being added to a graph \( G \) over time. We consider fixed width sliding windows over such a graph, i.e. a sliding window of time \( t \) is equivalent to the \( t \) most recently added edges. Note that an online may be viewed as a graph stream and we use the two terms interchangeably.

A pattern matching query is defined in terms of sub-graph isomorphism. Given a pattern graph \( q = (V_q, E_q) \), a query should return \( R \) a set of sub-graphs of \( G \). 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 \( q, l(v) = l(f(v)) \). A query workload is simply a multiset of these queries \( Q = \{(q_1, n_1), .., (q_k, n_k)\} \), where \( n_i \) is the relative frequency of \( q_i \) in \( Q \).

A query motif is a graph which occurs, with a frequency of more than some user defined threshold \( T \), as a sub-graph of query graphs from a workload \( Q \).

A vertex centric graph partitioning is defined as a disjoint family of sets of vertices \( P_k(G) = \{V_1, V_2, ..., V_k\} \). 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 \( S = (V, E), V_i \subseteq V \) and \( E_i \subseteq E \). We define the quality of a graph partitioning relative to a given workload \( Q \). Specifically, the number of inter-partition traversals (ipt) which occur while executing \( Q \) over \( P_k(G) \). Whilst min. edge-cut is the standard scale free measure of partition quality [14], it is intuitively a proxy for ipt and, as we have argued (Sec. 1), not always an effective one.

1.4 Overview

Once again, Loom continuously partitions an online graph \( G \) into \( k \) parts, optimising for a given workload \( Q \). The resulting partitioning \( P_k(G, Q) \) reduces the probability of expensive ipt, when executing a random \( q \in Q \), using the following techniques.

Firstly, we employ a trie-like datastructure to index all of the possible sub-graphs of query graphs \( q \in Q \), then identify those sub-graphs which are motifs, i.e. occur most frequently (Sec. 2). Secondly, we buffer a sliding window over \( G \), then use an efficient graph stream pattern matching procedure to check whether each new edge added to \( G \) creates a sub-graph which matches one of our motifs (Sec. 3). Finally, we employ a combination of novel and existing partitioning heuristics to assign each motif matching
sub-graph which leaves the sliding window entirely within an individual partition, thereby reducing ipt for Q (Sec. 4).

2 IDENTIFYING MOTIFS

We now describe the first of the three steps mentioned above, namely the encoding of all query graphs found in our pattern matching query workload Q. For this, we use a trie-like datastructure which we have called the Traversed Pattern Summary Trie (TPSTry++). In a TPSTry++, every node 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. 1 is shown in Fig. 2.

This structure not only encodes all sub-graphs found in each q ∈ Q, it also associates a support value with each of its nodes, to keep track of the relative frequency of occurrences of each sub-graph in our query graphs.

Given a threshold T for the frequency of occurrences, a motif is a sub-graph that occurs at least T times in Q. As an example, for T = 40%, Q’s motifs are the shaded nodes in Fig. 2.

Intuitively, a sub-graph of G which is frequently traversed by a query workload should be assigned to a single partition. We can identify 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.3. In the rest of this section we explain how a TPSTry++ is constructed, given a workload Q.

A TPSTry++ extends a simpler structure, called TPSTry, which we have recently proposed in a similar setting [8]. It employs frequent sub-graph mining [12] to compactly encode general labelled graphs. The resulting structure is a Directed Acyclic Graph (DAG), to reflect the multiple ways in which a particular query pattern may extend shorter patterns. For example in Fig. 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. In contrast, a TPSTry is a tree that encodes the space of possible traversal paths through a graph as a conventional trie of strings, where a path is a string of vertex labels, and possible paths are described by a stream of regular path queries [20].

Note that the trie is a relatively compact structure, as it grows with |LV|^t, where t is the number of edges in the largest query graph in Q and |LV| is typically small. Also note that the TPSTry++ is similar to, though more general than, Ribiero et al’s G-Trie [27] and Choudhury et al’s SJ-Tree [3], which use trees (not DAGs) to encode unlabelled graphs and labelled paths respectively.

2.1 Sub-graph signatures

We build the trie for Q by progressively building and merging smaller tries for each q ∈ Q, as shown in Fig. 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 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 [19], 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[27]. Instead, we propose a probabilistic, but computationally more efficient approach based on number theoretic signatures, which extends recent work by Song et al [29]. 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 ∈ LV, from our data graph G; recall that the function f_j maps vertices in G to these labels. We then perform three steps:

(1) Calculate a factor for each edge e = (v_j, v_j) ∈ E_q, according to the formula:

\[ e \text{Fac}(e) = (r(f_j(v_j)) - r(f_j(v_j))) \mod p \]

(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:

\[ d_v = \left( (r(f_j(v))) + 1 \right) \mod p \]

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

\[ \left( \prod_{e \in E_1} \text{Fac}(e) \right) \cdot \prod_{v \in V_2} d_v \]

Note that for the factors of directed edges, the random value for the target vertex’s label is subtracted from the random value for the source vertex’s label (i.e. v_j is the target vertex). For undirected edges the ordering of subtraction does not matter, provided it is consistent (e.g. lexicographical).

To illustrate this signature calculation process, consider query q_1 from Fig. 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:

\[ \text{Fac}(a, b) = (3 - 10) \mod 11 = 7 \]

As q_1 consists of four a-b edges, its total edge factor is 7^4 = 2401. Then we calculate the
degree factors, starting with a b labelled vertex with degree 2: 

degFac(b) = ((10+1) mod 11) ·((10+2) mod 11) = 11, ... factors to the relevant trie branches. This allows 
us to represent signatures as sets of their constituent factors,

Recursively add a query graph

116208400

3

its edge and degree factors, initially assuming a degree of 1 for

starting from each edge

DAG which encodes the sub-graphs of all G

tally compute signatures for sub-graphs of each query graph

Section 2.3.

cause all the factors are mapped to a finite field between 1 and

costly and slow. A small choice of

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 q1 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 
vertex ((10 + 2) mod 11). The combined signature for a-b-a is 
therefore 308 - 7 · 4 · 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 q1 until 
there are no edges left in q1 which are not in the sub-graph, at 
which point, q1 has been added to the TPSTry++.

2.3 Avoiding signature collisions

As mentioned, number theoretic signatures are a probabilistic method of isomorphism 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; c) two different graphs have identical sets of edges, vertices and vertex degrees.

The original approach to graph isomorphic checking makes use of an expensive authoritative pattern matching method 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. 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. 2 and Alg. 1, we can see that all possible sub-graphs of a query graph Gq 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\(^4\). 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 \(C\%\) of a signature’s factors are collisions. From Section 2.1 there are three scenarios in which a factor collision may occur: \(a)\) two 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 \(\frac{p}{2}\).

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 \(\frac{p}{2}\). The Handshaking lemma 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 \(\text{Binomial}(3|E|, \frac{p}{2})\). Binomial distributions tell us the probability of exactly \(x\) "successes" occurring, however we want the probability that no more than \(C_{\text{max}} = C\% \cdot 3|E|\) factors collide and so must sum over all acceptable outcomes \(x \in C_{\text{max}}\):

\[
\sum_{x=0}^{C_{\text{max}}} \Pr(X = x) \text{ where } X \sim \text{Binomial}(3|E|, \frac{p}{2})
\]

Figure 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 negligible probability of significant factor collisions.

\(^4\)Of the order of 10 edges.

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 \(Q\) 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 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 \(P_{\text{temp}}\). 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 ancestors 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 ancestors. 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. 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 \(P_{\text{temp}}\).

If, on the other hand, \(e\) does match a single-edge motif then we record the match into a map, \(\text{matchList}\), and add \(e\) to the window. The \(\text{matchList}\) maps vertices \(v\) to the set of motif matching sub-graphs in \(P_{\text{temp}}\) 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 \(\text{matchList}\) entries for both \(v_1\) and \(v_2\). Additionally, alongside every sub-graph in \(\text{matchList}\), we store a reference to the TPSTry++ node which represents the matching motif. Therefore, entries in \(\text{matchList}\) take the form \(v \rightarrow \{(E_1, m_1), (E_1, m_1), \ldots\}\), where \(E_1\) is a set of edges in \(P_{\text{temp}}\) 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 \(P_{\text{temp}}\) must at least match a single edge motif. However, if \(e\) is incident to other edges already in \(P_{\text{temp}}\), when its addition may also form larger motif matching sub-graphs which we must also detect and add to \(\text{matchList}\). Thus, having added \(e = (v_1, v_2)\) to \(\text{matchList}\), 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. 2, along with the TPSTry++,
Algorithm 2: Mine motif matches from each new edge \(e \in G\)

1. \(\text{factors}(e, g) \leftarrow \text{degree/edge factors to multiply a graph } g\)’s signature when adding edge \(e\)
2. \(\text{tpstry} \leftarrow \text{filtered TPSTry++ of motifs for workload } Q\)
3. for each new edge \(e(v_1, v_2)\) do
4. \(\text{matches} \leftarrow \text{matchList}(v_1) \cup \text{matchList}(v_2)\)
5. for each sub-graph \(m\) in matches do
6. \(n \leftarrow \text{the tpstry node for } m\)
7. if \(n\) has child \(c\) w. \(f\)actor = \(f\)actors\((e, m)\) then
8. add \((m + c, e)\) to \(\text{matchList}\) for \(v_1\) & \(v_2\) /Match found!
9. \(m_1 \leftarrow \text{matchList}(v_1)\)
10. \(m_2 \leftarrow \text{matchList}(v_2)\)
11. for all possible pairs \((m_1, m_2)\) from \((m_1, m_2)\) do
12. \(n_1 \leftarrow \text{the tpstry node for } m_1\)
13. \(\text{recurse}(\text{tpstry}, m_2, n_1)\)
14. for each edge \(e_2\) in \(m_2\) do
15. if \(n_1\) has child \(c_1\) w. \(f\)actor = \(f\)actors\((e_2, m_1)\) then
16. \(\text{recurse}(\text{tpstry}, m_2 - e_2, m_1 + e_2, c_1)\)
17. if \(m_2\) is empty then /Match found!
18. add \((m_1 + m_2, n_1)\) to \(\text{matchList}\) for \(v_1\) & \(v_2\)

Figure 5: \(t\)-length window over \(G\) (left), Motifs from TPSTry++ (center) and motif matchList for window (right)

Algorithm 2 determines 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 degree and edge factors \(f\)actors\((e, g_i)\) which would multiply the signature of \(g_i\) upon the addition \(e\), as in Sec. 2. Recall, also from Sec. 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. \(f\)actors\((e, g_i) = c\) signatures -- \(n\).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 \(G\) in this manner forms yet another new motif matching sub-graph \(g_1 \oplus e\).

We also detect if the joining of two existing multi edge motif matches \((\{(e_1, m_1), (e_2, m_2)\})\) 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 \((E_2, m_2)\), checking if the addition of any of these edges to \(E_1\), \(E_2\), constitutes yet another match; 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 \(G\). Note that as more edges are added to \(G\), motif 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. 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 add \(e_1\) to \(P_{\text{temp}}\), and add the entry \((e_1, m_1)\) to matchList for both \(e_1\)’s vertices 1.2. As \(e_1\) is not yet connected to any other edges in \(P_{\text{temp}}\), 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 \(P_{\text{temp}}\) and matchList accordingly. However, \(e_3\) is connected to existing motif matching sub-graphs in \(P_{\text{temp}}\) therefore the union of matchList entries for \(e_3\)’s vertices 4.5 (line 4 Alg. 2) returns \((\{e_2, m_1\})\). 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 \(b\) (4) (Sec. 2.1). Subsequently 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 \(P_{\text{temp}}\) which matches the motif \(m_3\), and must add \((\{e_2, e_3\}, m_3)\) 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 \((e_4, m_2)\) and \((e_1, e_4)\), as can be seen in our example matchList.

Finally, the addition of our last edge, \(e_5\), creates several new motif matches (e.g. \((e_5, m_2), (e_2, e_5), m_3\) etc…). In particular, notice that the addition of \(e_5\) creates a match for the motif \(m_5\), combining the new motif match \((e_5, m_5)\) with an existing one \((e_2, m_3)\). To understand how we discover these slightly more complex motif matches, consider Alg. 2 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 \((\{e_1, e_3\}, m_1)\), \((e_2, m_3)\), 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 \( (\{e_1, e_2, e_5\}, m_6) \) 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" it with new edges from the smaller match, updating matchList only if all edges from the smaller match have been successfully added.

### 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. 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 \( M_e = \{(E_1, m_1), \ldots (E_n, m_n)\}, e \in E_1 \) and \( E_i \) is a match for \( m_i \). Finally, treating these matches as a single sub-graph, assign them to the partition 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, portentiously 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 [30]. Similar to our naive solution above, LDG seeks to assign edges\(^*\) 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 \( |V(S_i)| \), and a partition capacity constraint \( C : r(S_i) = 1 - \frac{|V(S_i)|}{|V(S)|} \). 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:

\[
\max_{S_i \in P(G)} \frac{N(S_i, e) \cdot (1 - \frac{|V(S_i)|}{C})}{(\sum_{k=0}^{M_n} bid(S_i, (E_k, m_k)))}
\]

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 \( M_e = \{(E_1, m_1), \ldots (E_n, m_n)\}, \) 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 TPSTTry++.

Formally, given the motif matches \( M_e \) we compute a score for each partition \( S_i \) and motif match \((E_k, m_k) \in M_e \), which we call a bid. Let \( N(S_i, E_k) = |V(S_i) \cap V(E_k)| \) denote the number of vertices in the edge set \( E_k \) (which is itself a graph) that are already assigned to \( S_i \).\(^\circ\) Additionally, let \( supp(m_k) \) refer to the support of motif \( m_k \) in the TPSTTry++ and recall that \( C \) is a capacity constraint defined for each partition. We define the bid for partition \( S_i \) and motif match \((E_k, m_k) \) as:

\[
\text{bid}(S_i, (E_k, m_k)) = N(S_i, E_k) \cdot (1 - \frac{|V(S_i)|}{C}) \cdot \text{supp}(m_k) \quad (1)
\]

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 \( \text{min}_S = \text{min}_{S \in P(G)} |V(S)| \), i.e. if \( S_i \) is as small as \( \text{min} \) 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:

\[
l(S_i) = \frac{|V(S_i)|}{\text{min}_S} \cdot a, \quad a = \begin{cases} 1, & |V(S_i)| = |V(S_{\text{min}})| \\ 0, & |V(S_i)| > |V(S_{\text{min}})| \cdot b \\ a, & \text{otherwise} \end{cases} \quad (2)
\]

where \( a \) is a user specified number \( 0 < a \leq 1 \) which controls the aggression with which \( l \) penalises larger partitions and \( b \) limits the maximum imbalance. Throughout this work we use an empirically chosen default of \( a = \frac{5}{8} \) and set the maximum imbalance to \( b = 1.1 \), emulating Fennel [31].

Given definitions (1) and (2), we can now simply state the output of equal opportunism for the sorted set of motif matches \( M_e \), as:

\[
\max_{S_i \in P(G)} \frac{l(S_i) \cdot |M_e|}{(\sum_{k=0}^{M_n} bid(S_i, (E_k, m_k)))} \quad (3)
\]

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 \( P_{temp} \).

\(^*\)Note that \( N \) is a generalisation of LDG’s function \( N \).
To understand how 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 $P_{\text{temp}}$ 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. 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 $(e_1, m_1), (e_1, e_2), m_3)$, $(e_1, e_4), m_4)$ and $(e_1, e_5), m_5)$. 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, $S_2$ is guaranteed to win all bids, as $S_2$ contains no vertices from $M_e$, and therefore $N(S_2, \ldots)$ will always equal 0. However, rather than greedily assign all matches to the already large $S_1$, we compute a match in $S$ with $e_1$. In other words, we only assign edges from the first half of $M_{e_1}$, 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 workload $Q$. Instead, we compute a match in $P_{\text{temp}}$ between $(e_5, e_6)$ and the motif $m_5$, 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 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 when executing a realistic workload of pattern matching queries over each graph.

Loom consistently produces partitionings of around 20% superior quality when compared to those produced by state of the art alternatives: LDG [30] and Fennel [31]. 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.

#### 5.1 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; and 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 [11, 22, 30, 31], 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. As this is the default partitioner used by many existing partition graph databases\(^8\), 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, we use the Fennel parameter value $\gamma = 1.5$ throughout our evaluation. **Loom**: our own partitioner which, unless otherwise stated, we invoke as a baseline for our comparisons.

All experiments are performed on a commodity machine with a 3.1Ghz Intel i7 CPU and 16GB of RAM.

#### 5.1.1 Graph datasets

Remember that the workload-agnostic partitioners which we aim to supersede with Loom are likely 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\(^9\). All our experiments are performed on the Titan graph database: http://bit.ly/2ejypXV, MusicBrainz database: http://bit.ly/1J0w4INR

---


---

Figure 6: Examples of $q$ for MusicBrainz, DBLP & ProvGen
labels such as Artist, Country, Album and Label. ProvGen[6] is a synthetic generator for PROV metadata [21], which records detailed provenance for digital artifacts.

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. 1.3). 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. The full details of these query patterns are elided for space, however Fig. 6 presents some examples. Note that whilst the TPSTry++ may be trivially updated to account for change in the frequencies of workload queries (Sec. 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. 1). However, more complete tests with changing workloads are an important area for future work.

5.2 Comparison of systems

Figures 7 and 8 present the improvement in partitioning quality achieved by Loom and each of the comparable systems we describe above. Initially, consider the experiment depicted in Fig. 7. We partition ordered streams of each of our first 4 graph datasets into 8-way partitionings, 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. 7 (and 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. 1), Fennel is the more effective heuristic, cutting around 25% fewer edges than LDG for small numbers of partitions (including k = 8) [31]. 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. 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

Table 1: Graph datasets, incl. size & heterogeneity

| Dataset          | ~|V| | ~|E| | V | | L | | Y | | Real       | Description               |
|------------------|---|---|---|---|---|---|---|---|---|---|-----------|---------------------------|
| DBLP             | 1.2M | 2.5M | 8 | Y | | Publications & citations |
| ProvGen          | 0.5M | 0.9M | 3 | N | | Wiki page provenance     |
| MusicBrainz      | 31M  | 100M | 12| Y | | Music records metadata   |
| LUBM-100         | 2.6M | 11M  | 15| N | | University records       |
| LUBM-4000        | 131M | 534M | 15| N | | University records       |

Figure 7: ipt %, vs. Hash, when executing Q over 8-way partitionings of graph streams in multiple orders.

Figure 8: ipt %, vs. Hash, when executing Q over multiple k-way partitionings of breadth-first graph streams.

---

If possible, workloads are drawn from the literature, e.g. common PROV queries [5]

Excluding LUBM-4000
it is particularly pronounced over ordered streams of more heterogeneous graphs; e.g. MusicBrainz in Sub-figure 8(b), 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. 7, the median range of Loom’s ipt reduction relative to Fennel’s is 20 – 25%. Additionally, Fig. 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. 7, nor Fig. 8, present the runtime costs of producing a partitioning. Table 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. 2), therefore the performance figures presented in Table 2 are averaged across many different $Q$. The minimum slowdown factor observed between Loom and Fennel was 1.5, the maximum 7.1. Note that popular non-streaming partitioner METIS [14] is around 13 times slower than Fennel for large graphs [31].

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) [16]. Secondly, it is considered exceptional for even applications such as twitter to experience >30k-40k TPS [14]. Meanwhile, the lowest partitioning rate exhibited by Loom in Table 2 is equivalent to ~ 42k edges per second, the highest 72k.

Note that Figures 7 and 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 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 [15], with LDG varying between 1% – 3%, Loom and Fennel between 7% and their maximum imbalance of 10% (Sec. 4).

### 5.3 Effect of stream order and window size

Fig. 7 indicates that Loom is sensitive to the ordering of its given graph stream. In fact, Sub-figure 7(a) shows Loom achieve a smaller reduction in ipt over Fennel and LDG, than in 7(b) and 7(c). 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 being sensitive to stream ordering [30, 31]. 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 subgraphs in its stream window.

Intuitively, this sensitivity can be ameliorated by increasing the size of Loom’s window, as shown in Fig. 9 As Loom’s window grows, so does the probability that clusters of motif matching subgraphs will occur within it. This allows Loom’s equal opportunism heuristic to make the best possible allocation decisions for the subgraph’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 $t = 4k$) is highly likely to include all the motif matches which contain $a$. Regardless, Fig. 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. 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.

### Table 2: Time to partition 10k edges

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LDG (ms)</th>
<th>Fennel (ms)</th>
<th>Loom (ms)</th>
<th>Hash (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>91</td>
<td>96</td>
<td>235</td>
<td>28</td>
</tr>
<tr>
<td>ProvGen</td>
<td>144</td>
<td>146</td>
<td>240</td>
<td>33</td>
</tr>
<tr>
<td>MusicBrainz</td>
<td>48</td>
<td>52</td>
<td>129</td>
<td>18</td>
</tr>
<tr>
<td>LUBM-100</td>
<td>47</td>
<td>51</td>
<td>147</td>
<td>22</td>
</tr>
<tr>
<td>LUBM-4000</td>
<td>45</td>
<td>49</td>
<td>138</td>
<td>16</td>
</tr>
</tbody>
</table>

![Figure 9: ipt (y-axis) when executing Q over Loom partitionings with multiple window sizes t(x-axis)](image-url)

14Tweets per second in 2013: http://bit.ly/2hQH5JJ

15Except Hash, which is balanced.
6 CONCLUSIONS AND FUTURE WORK

In this paper, 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. Our experiments indicate that Loom significantly reduces the number of inter-partition traversals (ipt) required when executing Q over its partitionings, relative to state of the art (workload agnostic) streaming partitioners.

There are several ways in which we intend to expand our current work on Loom. In particular, as a workload sensitive technique, Loom generates partitionings which are vulnerable to workload change over time. In order to address this we must integrate Loom with an existing, workload sensitive, graph re-partitioner [8, 10] or consider some form of restreaming approach [11]. In addition to the query workloads already considered, it would be necessary to evaluate such an integrated approach using a dynamic, changing query workload.

Furthermore, due to our approaches reliance upon graph pattern matching in a single stream window, Loom is single threaded. The ability to have multiple instances of the Loom algorithm assign motif matches to the same graph partitioning would doubtless increase system scalability, and is therefore an important focus of ongoing research.

REFERENCES