Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs

Laurent Bindschaedler
bindscha@mit.edu
Massachusetts Institute of Technology

Jasmina Malicevic
jasmina.malicevic@swisscom.com
Swisscom

Baptiste Lepers
baptiste.lepers@sydney.edu.au
University of Sydney

Ashvin Goel
ashvin@eecg.toronto.edu
University of Toronto

Willy Zwaenepoel
willy.zwaenepoel@sydney.edu.au
University of Sydney

Abstract

Tesseract is the first *distributed* system for executing *general* graph mining algorithms on *evolving* graphs. Tesseract scales out by decomposing a stream of graph updates into per-update mining tasks and dynamically assigning these tasks to a set of distributed workers. We present a novel approach to change detection that efficiently determines the exact modifications to the algorithm's output for each update to the input graph. We use a disaggregated, multiversioned graph store to allow workers to process updates independently, without producing duplicates. Moreover, Tesseract provides interactive mining insights for complex applications using an incremental aggregation API. Finally, we implement and evaluate Tesseract and demonstrate that it achieves orders-of-magnitude improvements over state-of-the-art systems.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; **Data flow architectures**; • **Theory of computation** → *Dynamic graph algorithms*; • **Mathematics of computing** → **Graph algorithms**; **Graph enumeration**; • **Information systems** → *Graph-based database models.*

Keywords: Tesseract, graph, pattern, mining, processing, distributed, evolving, dynamic, temporal, graph store, differential, incremental, streaming, subgraph matching

ACM Reference Format:
Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456253

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456253

1 Introduction

Graph pattern mining has wide-ranging applications, such as discovering chemical interactions or 3D protein structures [22, 55], analyzing communities in social networks [26], mining frequent motifs in networks [48], analyzing semantic data [50], or detecting suspicious credit card transactions [54]. Examples of algorithms include motif counting [9], frequent subgraph mining [32], graph keyword search [36, 66], and clique or diamond mining [19, 30].

This paper focuses on distributed graph mining of large evolving graphs. Mining a graph involves enumerating all its subgraphs that match a pattern of interest, called *matches*. Mining an evolving graph requires continuously updating the set of matches that have been mined, as vertices, edges, and labels are added, deleted, or modified. Efficiently computing these changes in the match set is challenging because a single graph update can create new and delete existing matches. Simply recomputing all matches in the entire graph on every update is prohibitively expensive.

Table 1 presents recent, state-of-the-art graph mining systems and compares their features. Most graph mining systems focus on processing static graphs [24, 64] or run on single nodes [31, 34, 39, 46, 67]. Delta-BigJoin [10] is the only distributed system to support evolving graphs. However, it is not a general mining system and only supports a subclass of problems where the pattern is expressed as a fixed subgraph.

System	Evolving	Distributed	General
TurboISO [31]	✓		
Turboflux [39]	✓		
BigJoin [10]		✓	
RStream [67]			✓
AutoMine [46]			✓
Peregrine [34]			✓
Pangolin [21]			✓
Delta-BigJoin [10]	✓	✓	
Arabesque [64]		✓	✓
Fractal [24]		✓	✓
Tesseract	✓	✓	✓

Table 1. A comparison of state-of-the-art graph mining systems in terms of support for evolving graphs, distributed execution, and generality of the programming model.

We present Tesseract, the first *distributed* streaming system designed for *general* pattern mining on large *evolving* graphs. Tesseract supports general graph mining by allowing patterns to be expressed as arbitrary code. Furthermore, developers write the same code as they would for static graph mining, without worrying about the graph evolving. Then, Tesseract executes the mining algorithms incrementally on a stream of graph updates and streams out the corresponding changes in the match set. Finally, Tesseract offers a novel streaming aggregation API that makes it easy for programmers to incrementally post-process matches.

The key innovation in Tesseract is the notion of *update-based exploration*: we explore the neighborhood of a graph update, starting from a subgraph rooted at the update and expand the subgraph using neighboring vertices to enumerate all changes in the match set. Update-based exploration relies on the observation that changes in the match set must necessarily contain the graph update. Furthermore, since graph mining problems are localizable and bounded [25], a graph update only affects a limited subset of matches in its neighborhood, unlike graph analytics problems [53, 65] where a single graph update may affect the entire result. This targeted exploration is much more efficient than recomputing all matches in the entire updated graph.

Update-based exploration raises a number of challenges. The first challenge is to accurately and efficiently determine any new matches resulting from the update and any pre-update matches that are invalidated by the update. Tesseract uses a multiversioned graph store in which, roughly speaking, each update produces a new version of the graph. We present an efficient differential processing technique that uses the pre- and post-update graph versions to determine for each match whether it exists before and/or after the graph update. The second challenge is to prevent duplicate exploration, which leads to redundant work and duplicate outputs. Tesseract avoids duplicate exploration by using a combination of a canonical exploration order and clever use of the multiversioned store.

Besides its incremental nature, another major advantage of Tesseract's update-based exploration is that it can be scaled out efficiently, because Tesseract's differential processing and duplicate elimination ensure that each update can be processed in any order independently. Tesseract distributes updates dynamically across workers in order to achieve good load balance. Superficially, it would seem that this requires the entire graph to be replicated. However, we do not replicate the graph across all workers, nor do we attempt to partition the graph. Instead our multiversioned graph store is sharded but fully accessible to all workers. This approach entirely avoids communication and synchronization between workers — a major bottleneck prevalent in existing distributed graph mining systems [10, 64, 67].

We have implemented the above-mentioned techniques in Tesseract and integrated our system with Apache Spark [71] and Apache Kafka [37] to provide a complete software solution for mining evolving graphs. The resulting system achieves good scalability and has a small memory footprint as workers need not keep any update-related state in memory beyond their useful life cycle.

We show that Tesseract can output changes in the match set at a rate of millions per second on large input graphs and with low latency. We demonstrate the benefits of incremental computation over full computation by comparing Tesseract with Fractal [24], the fastest static distributed graph mining system. For instance, for a clique mining application on the LiveJournal dataset [1], Tesseract maintains the match set 51× faster for a change of 1% to the input graph and 483× faster for a change of 0.1%. On the same dataset with a frequent subgraph mining application, Tesseract is 110× faster for a change of 1% and 1,067× faster for a change of 0.1%. Thanks to Tesseract's low memory requirements and communication, we outperform static mining systems on the entire graph. We also compare Tesseract against Delta-BigJoin [10], which supports evolving graphs but is specialized for fixed relational subgraph queries, a subclass of graph mining problems. Tesseract mines cliques 1.1× faster and counts motifs 26× faster than BigJoin. Finally, Tesseract offers comparable performance to Peregrine [34], the fastest general mining system for a single node.

We make the following contributions in this paper:

- We present Tesseract, the first distributed, streaming general graph mining system for large evolving graphs.
- We propose an incremental mining approach to enumerate the exact set of changes in the match set resulting from each graph update.
- We show how our multiversioned graph store allows workers to operate independently and avoid duplicates.
- We outline how Tesseract supports aggregation in the presence of graph updates.
- We show that Tesseract incrementally maintains the mining result at a fraction of the full recomputation cost and is capable of emitting changes in the match set at a rate of millions per second and with low latency.

2 Background

Graph mining problems aim to discover instances of interesting patterns in an input data graph. The input graph can be either directed or undirected, with labels attached to vertices and edges. Labels include identifiers (usually integers) as well as user-defined properties. Patterns are arbitrarily connected subgraphs. Mining a graph is done via subgraph matching, i.e., enumerating all subgraphs that match some criteria of interest, such as a specific pattern or certain graph properties (e.g., frequent occurrences in the graph). A *match* is a subgraph of the graph that corresponds to a given pattern, i.e., the match and the pattern have the same number of vertices, the edge structures, and labels.

The pattern can either be defined as a fixed graph (e.g., a cycle), as a property (e.g., frequent occurrence in the graph), or as arbitrary code. A graph mining system is general if it supports arbitrary code definitions of the patterns of interest. Similarly, a system that only supports matching fixed patterns, defined as *pattern graphs*, is a subgraph query system [10, 31, 39]. For example, a general graph mining system can match cliques of any size based on the property that "all vertices in the subgraph must be connected to all other vertices in the subgraph". A subgraph query system requires a separate pattern graph for every clique size. Some recent systems can also support arbitrary patterns by generating programs that can match multiple patterns in a single execution [34, 46].

Matches can either be *vertex-induced*, where the subgraph contains *all* edges connecting the vertices in the data graph, or *edge-induced*, where the subgraphs need not contain all edges present in the data graph. Most graph mining algorithms use vertex-induced matches. However, some graph mining algorithms [32] require enumerating edge-induced subgraphs instead.

Motivating example Consider, for example, the popular problem of *graph keyword search* illustrated in Figure 1. Given a set of labels, graph keyword search finds all subgraphs of the input graph whose vertices contain all the labels of interest. These subgraphs must be *minimal*, i.e., not contain any unnecessary vertices. This problem has many practical applications in social networks, recommender systems, and the semantic web [66].

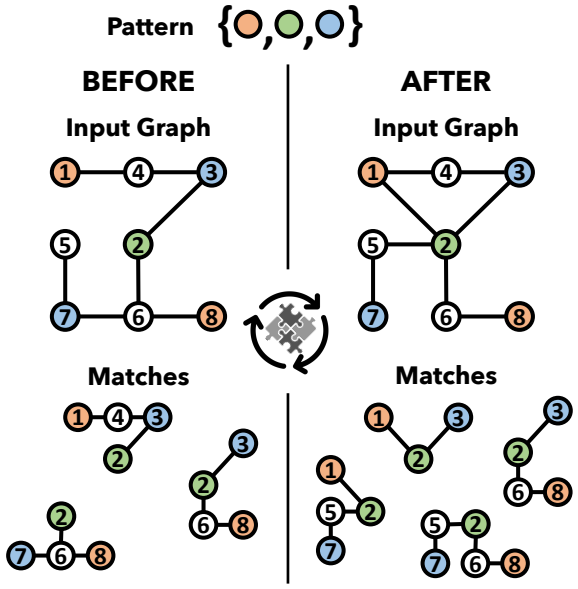

Figure 1. Graph keyword search example.

In the example in Figure 1, the pattern consists of three labels that are represented as colors (orange, green, or blue). The pattern matches any subgraph that contains exactly one vertex of each label. Subgraphs may contain other vertices (represented in white), but must remain minimal. On the left, we show an input graph and its matches: (1, 2, 3, 4), (2, 3, 6, 8), and (2, 6, 7, 8).

In Figure 1, on the right, we apply three graph updates: (+(1, 2), +(2, 5), −(6, 7)) and show the changes in the match set. Visually, it is easy to see that these updates invalidate two matches, (1, 2, 3, 4) and (2, 6, 7, 8), and create three more: (1, 2, 3), (1, 2, 5, 7), and (2, 5, 6, 7, 8).

3 Programming Model

We first describe the core API and show how to implement two common graph mining applications in Tesseract. Next, we present the output processing and aggregation API and demonstrate its use in two other applications.

By convention, in the following algorithms and text, API functions (implemented by developers) are represented in `typewriter` font and internal functions (implemented in the system) are represented in SMALL CAPS.

3.1 Core Mining API

Tesseract allows programmers to express algorithms as static graph mining programs without worrying about graph updates. The additional complexity of incremental computation is hidden from the user. In fact, most algorithms written for static systems can be directly ported to Tesseract or require only minimal modifications.

Tesseract applications are specified using two programmer-defined API functions: `filter` that determines whether to stop exploring (or prune) a subgraph and its extensions, and `match` that decides whether a subgraph is a match. This filter-match model is expressive enough to implement a wide variety of algorithms and is not limited to subgraph queries [10]. With these two functions, Tesseract automatically executes the algorithm on the input graph and derives the changes in the match set as graph updates arrive. The execution of a mining algorithm is conceptually an exhaustive search for all `matching` patterns that prunes (`filters`) subgraphs that cannot lead to matches. As in many existing graph mining systems, Tesseract requires that applications satisfy two standard properties for completeness and correctness:

- Anti-monotonicity [64]: if `filter`(s) is false for a subgraph s, then `filter`(s') must be false for any expansion s' of s.
- Boundedness [25]: `filter` returns false after exploring a bounded set of neighbors around the update. This is typically achieved using a maximum subgraph size.

Tesseract supports two execution modes: vertex-induced and edge-induced. The following description assumes vertex-induced subgraphs unless otherwise stated.

Tesseract receives timestamped graph updates in a streaming fashion and emits matches corresponding to these updates. Internally, Tesseract uses the `filter` and `match` functions in its exploration algorithm, which takes the graph updates as input and outputs a 3-tuple for each change to the match set: `(timestamp, status, subgraph)`. Tesseract identifies each emitted match by the timestamp of the graph

update that produced it. The second tuple element is a differential match status which can be NEW (new match) or REM (removed match). The final element is the subgraph that matched the pattern, including vertices, edges, and labels.

Tesseract supports two output stream modes: unordered and ordered. The unordered stream emits matches immediately, irrespective of the timestamp order. This stream provides lower latency and is useful for applications that can handle an eventually consistent result, e.g., graph keyword search. In ordered mode, matches are emitted in timestamp order. This mode is used by applications such as frequent subgraph mining [32] that cannot handle out-of-order matches. The programmer specifies the output mode to use.

3.2 Graph Mining Applications

Algorithm 1 shows two applications implemented using our programming model: graph keyword search with three labels (see Figure 1) and a clique mining algorithm. The implementation of these two algorithms in the Arabesque [64] static graph mining system is identical, barring function names.

Algorithm 1: Examples of graph mining applications

```
1  algorithm graph_keyword_search
2      function filter(s)
3          return len(s) <= MAX and
               num_orange(s) <= 1 and num_green(s) <= 1
               and num_blue(s) <= 1
4      function match(s)
5          if num_green(s) !=1 or num_orange(s) != 1
               or num_blue(s) != 1 then return false
6          foreach vertex v in s if color(v) == white do
7              if IS_CONNECTED(s \ v) then return false
8          return true
9  algorithm clique_mining
10     function filter(s)
11         return len(s) <= MAX and
12         num_edges(s) == len(s)*(len(s)-1)/2
13     function match(s)
14         return true
```

In the graph keyword search example, the `filter` function prunes subgraphs with more than one vertex of a given color since these can never match. We limit the maximum subgraph size for bounded execution. The `match` function checks that a subgraph has exactly one vertex of each color, and that it is minimal, i.e., it does not contain any unnecessary vertices with other labels (represented as white here). It does so by checking for each white vertex whether the subgraph remains connected if that vertex is removed.

A clique is a subgraph where each vertex is connected to all other vertices. In this example, the `filter` function checks that the number of edges in the subgraph is equal to the number of edges that should be present in a clique of the same size (a clique with n vertices must have exactly $\frac{n(n-1)}{2}$ edges). The `len` function returns the number of vertices in the subgraph. The `filter` function checks for cliques of any size, thus allowing mining patterns of varying sizes. We set a maximum clique size for bounded execution. The `match` function returns true since every filtered subgraph is a match.

3.3 Output Processing & Aggregation API

A common task in graph mining algorithms involves processing the output, such as transforming, filtering, or aggregating matches. For example, in the graph keyword search example (Figure 1), we may want to count the number of matches. Another popular algorithm is frequent subgraph mining [32] that enumerates all subgraphs that appear more times than a threshold value, requiring post-processing and aggregating matches to provide a feedback loop to the exploration process.

Many existing systems [10, 39] do not support output processing and aggregation and leave it to the user, while others [64, 67] perform aggregation as a separate post-processing step once the entire match set is available. However, with an evolving graph, it is desirable to perform output processing and aggregation in a streaming manner. The incremental nature of the computation presents several challenges, particularly for maintaining the aggregation state as matches are updated.

Operations	
MAP	Transform each match
FILTER	Keep matches satisfying the predicate
FLATMAP	Transform each match and flatten
JOIN	Join with table or other stream
GROUPBY	Group matches
COUNT	Sum matches
AGG	Custom aggregation
...	
Helpers	
MOTIF	Identify the motif for a match

Table 2. Subset of the Output Processing and Aggregation API. Tesseract uses and extends the Spark Structured Streaming API [2].

Tesseract makes it easy for programmers to express the desired processing and aggregation operations using the API presented in Table 2. Tesseract uses the Spark Structured Streaming API [2] to process output tuples continuously. Thus we support common operators such as MAP and FILTER[1], as well as COUNT and AGG (custom aggregation). Moreover, these functions can be combined with specific graph mining helpers such as MOTIF that identifies the motif[2] for a given match. Tesseract automatically maintains aggregation state.

For most simple aggregation tasks, the programmer need not worry about the incremental nature of the computation

[1]Not to be confused with the user-defined `filter` function.

[2]Each match is isomorphic to a single fixed subgraph called a *motif*.

and merely writes code for static aggregation. Tesseract handles differential counting using the NEW and REM status emitted along with matches. Programmers must provide the appropriate differential semantics for custom aggregations as the system cannot automatically determine them.

Examples Counting the number of matches in our graph keyword search example (Figure 1) is easily implemented by calling stream.COUNT() on the output stream.

Similarly, implementing motif counting in Tesseract is fairly straightforward. We use `filter` to keep subgraphs and `match` every subgraph to enumerate all subgraphs as matches. As the matched tuples are emitted in a stream, we first group (GROUPBY) them by the MOTIF of the match and `count` the tuples in each group. This is implemented in a single line of code for tuple t as shown below:

stream.GROUPBY(t→MOTIF(t.subgraph)).COUNT()

We conclude this section by looking at a more complicated example: frequent subgraph mining (FSM). We implement FSM using the minimum image-based support metric [18]. FSM differs from motif counting because it emits the matches for frequent motifs whose support is above a user-defined threshold value instead of only maintaining a per-motif count. Tesseract executes FSM using edge-induced subgraphs. We implement FSM using a custom aggregation operator (AGG) that computes the support for the pattern and outputs the matches if the support is above the threshold. Otherwise, the matches are filtered out, but we still update and maintain the support value for the associated pattern. Discarding non-frequent subgraphs saves storage space and is generally done by static systems as well. However, unlike static systems that can safely discard these subgraphs since the input graph does not change, we may end up in a situation where a pattern becomes frequent after the graph changes and these previously discarded matches should then be output. Our FSM implementation automatically handles this case by recomputing all the subgraphs matching the pattern in question, and emits them when the pattern's support value crosses the threshold. Note that computing all subgraphs for a single pattern can be done significantly faster than computing all subgraphs (i.e., all patterns) for the entire graph, especially since the number of such subgraphs corresponds to the value of the support threshold. When a pattern becomes infrequent (due to removed matches), we do not recompute all associated subgraphs in order to invalidate them but simply indicate that the pattern has lost support.

4 The Tesseract System

We first provide an overview of the system (§4.1). Then, we introduce the single-worker version of Tesseract, including its exploration algorithm (§4.2), its differential processing technique (§4.3) and its duplicate elimination mechanisms (§4.4). Finally, we consider parallel exploration across distributed workers (§4.5).

4.1 Overview

Tesseract incrementally mines patterns in evolving graphs by decomposing the stream of graph updates into independent exploration tasks that are dynamically assigned across a set of distributed workers in a cluster. Tesseract supports the following updates: 1) addition and deletion of edges, 2) addition and deletion of vertices, 3) addition, deletion, and modification of labels. In the following discussion, we assume that updates are expressed as edge updates: vertex updates add or delete the associated edges, label updates delete the associated vertex or edge before adding it with the new label.

Figure 2 presents the architecture and the different components of Tesseract as the system executes the graph keyword search example from Figure 1. Updates are streamed from one or multiple data sources and are received by an ingress node that assigns timestamps to them in increasing order before applying them to a timestamp-based, multiversioned graph store. Then, it inserts the updates into the work queue.

A worker executes the graph mining application (such as described in Algorithm 1) using an exploration algorithm that operates on each update in the work queue (§4.2) and uses graph snapshots to compute the differences (§4.3) before emitting them to a publish-subscribe system. These differences are processed using the aggregation API (Table 2) to produce live mining results.

The graph store is sharded across all cluster nodes where workers are executing. Each worker has read-only access to any part of the graph and can process any update, which simplifies load balancing. Our approach does not require replicating the entire input graph [24, 64], which limits the size of the graph that can be mined to the memory available on a single machine. We also do not partition the graph across workers [10], thus avoiding synchronization and shuffling of candidate subgraphs that cross partition boundaries.

At first glance, the introduction of a total order for updates based on timestamps appears counter-intuitive as it may introduce an unnecessary scaling bottleneck. However, as we show later (§6.5.5), the exploration of updates makes graph mining overwhelmingly CPU-bound and, therefore this ordering is not a bottleneck. Moreover, by imposing a total order on updates, we remove all synchronization requirements for update exploration, allowing explorations to run in isolation with good scalability.

4.2 Update-Based Exploration

Our approach is based on the observation that most graph mining algorithms are localizable or bounded [25]. Unlike graph analytics algorithms such as PageRank [53] or Single Source Shortest Path [65], each match is a small subgraph of the input graph, so a single update to the input graph only impacts the matches in the neighborhood of the update. As a result, computing only the changes in the match set is much faster than computing all matches in the updated graph from scratch.

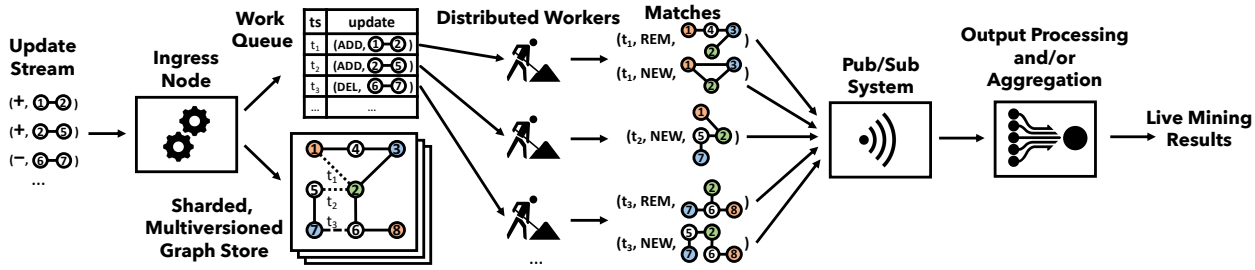

Figure 2. Tesseract architecture and graph keyword search example from Figure 1.

Algorithm 2 shows the incremental exploration strategy used by Tesseract to compute the changes in the match set resulting from a graph update for vertex-induced subgraphs. We discuss edge-induced subgraphs at the end of this section.

At a high level, Tesseract recursively explores the neighborhood of the graph update, using depth-first expansion and backtracking. In more detail, Tesseract invokes the function EXPLORE for each edge update, with as arguments the initial subgraph s, containing this edge and its two endpoint vertices, the timestamp ts of the update, and G, the snapshot of the graph at ts[3]. This function also uses two continuation variables (c_{pre} and c_{post}) to determine when to stop exploring. For the purposes of exploration, we treat both added and deleted edges in the same way.

EXPLORE starts by considering one neighboring vertex v of s in G. The CAN_EXPAND function filters out redundant exploration and duplicate matches (§4.4). The EXPAND function expands s to s' by adding v and all edges that connect v to s. Next, we use the DETECT_CHANGES function to determine whether the expanded subgraph s' results in a change to the match set and whether further expansions of s' may still contain matches (c'_{pre} or c'_{post} is true). If the latter condition holds, we invoke EXPLORE recursively. Otherwise, we backtrack and try another neighbor.

EXPLORE completes once all neighbors of the updated edge, up to a maximum subgraph size, have been explored. This guarantees that all subgraphs that include the graph update and match the pattern have been explored.

Tesseract supports edge-induced subgraph enumeration by converting each vertex expansion into the equivalent edge-induced expansions. To do so, the system expands one edge-induced subgraph for each permutation of edges connecting the expansion vertex by adding a loop over these permutations before line 5 of Algorithm 2.

4.3 Change Detection

Our incremental model aims to ensure that after each update, the match set is the same as if a static algorithm had produced all the matches on the entire updated graph. The DETECT_CHANGES function achieves this correctness goal by ensuring that the changes in the match set after each graph update correspond exactly to the difference between the match sets

[3]The use of the timestamp and the snapshot is discussed in Section 4.4.2

Algorithm 2: The EXPLORE Algorithm

input : G data graph snapshot at timestamp ts
input : ts update timestamp
input : s subgraph (initialized to the edge update)
input : c_{pre} continue pre-update (initialized to true)
input : c_{post} continue post-update (initialized to true)

1 **function** EXPLORE(G, ts, s, c_{pre}, c_{post}) **is**
2 **foreach** *neighbor v of s in G* **do**
3 **if** CAN_EXPAND(G, ts, s, v) **then**
4 $s' \leftarrow$ EXPAND(G, s, v)
5 $(c'_{pre}, c'_{post}) \leftarrow$ DETECT_CHANGES(G, ts, s', c_{pre}, c_{post})
6 **if** c'_{pre} ***or*** c'_{post} **then**
7 EXPLORE(G, ts, s', c'_{pre}, c'_{post})
8 **function** DETECT_CHANGES(G, ts, s', c'_{pre}, c'_{post}) **is**
9 $s'_{pre} \leftarrow$ SUBGRAPH_AT_PREVIOUS_SNAPSHOT(G, s')
10 **if** c'_{pre} ***and*** `filter`(s'_{pre}) **then**
11 **if** IS_CONNECTED(s'_{pre}) **and** `match`(s'_{pre}) **then**
12 EMIT (ts, REM, s'_{pre})
13 **else** $c'_{pre} \leftarrow$ **false**
14 **if** c'_{post} ***and*** `filter`(s') **then**
15 **if** IS_CONNECTED(s') **and** `match`(s') **then**
16 EMIT (ts, NEW, s')
17 **else** $c'_{post} \leftarrow$ **false**
18 **return** (c'_{pre}, c'_{post})

before and after applying the update. This difference may contain additions, deletions, or both.

DETECT_CHANGES implements a differential processing technique that finds all the changes caused by an update to the graph by exploring the graph in two states: before the update (s'_{pre}) and after the update (s'). In doing so, it determines for each candidate subgraph under consideration whether it is a match and, if so, whether it is newly created (NEW) or removed (REM). DETECT_CHANGES obtains the previous version of the subgraph s'_{pre} by calling the SUBGRAPH_AT_PREVIOUS_SNAPSHOT function which computes the subgraph corresponding to the vertices in s' in the graph snapshot

immediately preceding ts. Computing s'_{pre} can be done efficiently from G by simply excluding the update (removing an added edge or adding a deleted edge).

For each pre-update and post-update subgraph, we check whether further expansions of that subgraph are needed using the c'_{pre} and c'_{post} variables, and whether the subgraph passes `filter`. If so, we then check that it is connected and passes the `match` function, as shown in Algorithm 2. If not, we set the respective continuation variable (c'_{pre} or c'_{post}) to false (line 13 or 17) to stop further exploration as per anti-monotonicity of `filter`. If the pre-update subgraph passes all tests, it corresponds to a match that was present in the graph prior to the update, and is no longer present due to the graph update, and, therefore, must be emitted as a removed match (REM). Similarly, if a post-update subgraph passes all tests, the corresponding match is present in the graph after applying the graph update and is emitted as a newly created match (NEW). Note that in vertex-based exploration, it is possible for both s'_{pre} and s' to match the pattern. For example, consider a path mining application (looking for sequences of connected vertices) and a graph consisting of two edges connecting three vertices (1, 2, 3). Initially, there are edges connecting (1, 2) and (2, 3) but not (1, 3). An edge update for (1, 3) causes both s'_{pre} and s' to match the pattern. In this case, the function simply emits both matches (one REM and one NEW). Furthermore, if either the pre- or the post-update subgraph passes `filter`, at least one of c'_{pre} or c'_{post} remains true (line 18) and EXPLORE is invoked recursively. To satisfy our correctness criterion, we must continue expanding as long as at least one version of the subgraph can be expanded since it can still yield matches. Otherwise, c'_{pre} and c'_{post} are false, and EXPLORE backtracks.

Example In Figure 1, consider match (1, 2, 3, 4) in the original graph and consider adding edge (1, 2). DETECT_CHANGES first expands (1, 2) with vertex 3. s'_{pre} = ((1), (2, 3)) passes `filter`, but is disconnected due to the absence of edge (1, 2) (the condition at line 11 fails). s' = ((1, 2), (2, 3)) also passes the filter (the condition at line 14 succeeds). s' is connected and also passes the `match` function (line 15) and, therefore Tesseract EMITS this match as NEW (line 16). c'_{pre} and c'_{post} are true, and the exploration continues. Tesseract invokes EXPLORE recursively (line 7), now expanding the subgraph with vertex 4 to (1, 2, 3, 4). The subsequent call to DETECT_CHANGES finds that s'_{pre} = ((1, 4), (4, 3), (3, 2)), i.e., the subgraph without edge (1, 2), passes the filter, is connected and is a match. Therefore, Tesseract EMITS the match as REM (line 12). s' = ((1, 2), (1, 4), (4, 3), (3, 2)), i.e., the subgraph with edge (1, 2), passes the filter, is connected but is not a match, since it is not minimal ((1, 2, 3) is minimal). c'_{pre} and c'_{post} are true, and the exploration continues. The subsequent recursive calls to EXPLORE consider expansions where ultimately DETECT_CHANGES always returns with c'_{pre} and c'_{post} false, ending this branch of the exploration.

Correctness We make an informal argument that change detection is correct, i.e., it finds all the changes caused by a graph update. First, by exploring both pre- and post-update subgraphs, it follows that any matching pre-update subgraph is invalidated, and any matching post-update subgraph is a new match. Second, DETECT_CHANGES only stops exploring when neither pre- nor post-update subgraphs can yield further matches, and, therefore, by anti-monotonicity, all changes are detected. To avoid the cost of expanding the graph twice, Algorithm 2 interleaves the inspection of the pre- and post-update subgraphs using a single expansion and uses c_{pre} and c_{post} to avoid unnecessary calls to `filter`.

4.4 Avoiding Duplicate Exploration

We now describe how Tesseract prevents exploration of duplicates by a single worker thread. Duplicates (also known as automorphisms) are identical matches (same vertices and edges) up to the order in which the subgraph was constructed. Avoiding duplicates is necessary to ensure the correctness of graph mining algorithms. These duplicate matches should not be exposed to the user, and there is no benefit to exploring a subgraph multiple times. At a single worker thread, duplicates can occur for two reasons. The first reason is pattern symmetry, i.e., finding the same match twice by exploring the subgraph in different orders. The second reason is that the same match can be explored from two different updates, i.e., the match contains two updated edges and each call to EXPLORE finds it.

Besides finding and emitting duplicates, a single worker may also explore the same subgraph multiple times unsuccessfully due to the close proximity of updates in the graph. Although this issue does not affect correctness, since the worker only explores the same subgraph repeatedly without emitting a match, it is desirable to minimize its occurrence for performance reasons.

In the following subsections, we present Tesseract's approach to avoid duplicates in the case of pattern symmetry (§4.4.1) and in the presence of multiple updates to the same match (§4.4.2). We next present a snapshot-based exploration technique that reduces duplicate unsuccessful explorations (§4.4.3). Finally, implement `can_expand` (§4.4.4).

4.4.1 Breaking Symmetry

Static systems often rely on symmetry breaking techniques [23] or post-processing to remove duplicates. Tesseract solves this problem for evolving graphs by enforcing a single expansion order that we call the *update canonical* order when exploring the graph. The CAN_EXPAND check in Algorithm 2 rejects all non-canonical expansion orders, thereby avoiding automorphic (duplicate) subgraph exploration. Given a subgraph $s = (v_1 \ldots v_k)$ and an expansion vertex v, CAN_EXPAND ensures that the expansion follows the following two *update canonicality* rules:

Rule 1: (v_1, v_2) is the updated edge, with $(v_1 < v_2)$.

Rule 2: The vertex v is added if, ignoring v_1 and v_2, it has the smallest id among all neighbors of s and has not been considered for expansion yet.

Example In Figure 1, the (1, 2) edge can be expanded using vertices 3, 4, or 5. If we expand using vertex 3, then we can expand further using vertices 4 or 5. However, if we first expand using 4, then we can expand using vertex 5, but we cannot expand using vertex 3 as this would violate the second canonicality rule. As a result, there is only one valid expansion order for the subgraph (1, 2, 3, 4). Therefore, each match is only explored once (in the same canonical order). Update canonicality is similar to techniques found in existing static mining systems [24, 34, 35, 64], but unlike those systems, supports updates to the graph.

Correctness We make an informal argument that update canonicality is correct, i.e. it does not prune any matches that should be explored and does not lead to duplicates. Our update canonicality rules cannot prune any match candidates because they only enforce the expansion order of the subgraphs to explore. Then the question is whether update canonicality can lead to duplicate matches. Such matches can be found in two ways: by executing exploration from different starting points or by choosing expansion vertices in different orders from the same exploration. Clearly, the former does not apply since we are considering a single exploration rooted at the updated edge. The latter also cannot happen since we are enforcing canonicality on any expansion vertices, guaranteeing that two vertices in the neighborhood can only both be added to the subgraph in the same order.

4.4.2 Avoiding Overlapping Explorations

Tesseract prevents overlapping exploration and the output of duplicate matches using a multiversioning strategy that orders updates in the graph store to ensure future updates are not visible to workers. A worker exploring an edge update only sees expansions involving edges with a timestamp lower than the update timestamp because the graph G in Algorithm 2 is a snapshot taken at the timestamp of the update. This graph snapshot prevents workers from including updated edges from the future in matches, guaranteeing that duplicate matches are not produced. Moreover, this approach removes exploration overlap for updates with different timestamps involving the same match since only the exploration with the highest timestamp is discovered. For example, in a triangle mining application with two edge additions that form a triangle, only the exploration task processing the second update finds a matching triangle.

Correctness We provide an argument that overlapping explorations starting from different edges cannot find duplicate matches. Consider a match discovered by starting exploration from an edge update. Given that the snapshot of the graph at the timestamp corresponding to the edge update cannot include any future edges, all other edge updates that are part of the same subgraph must, therefore, have a lower timestamp. Since each exploration is rooted at the edge update. It follows that any new match found during the exploration could not have been discovered previously from an update with a lower timestamp. Similarly, any removed match could not have been removed previously.

4.4.3 Snapshot-Based Exploration

We now describe how Tesseract improves performance by assigning multiple consecutive updates the same timestamp. This approach is particularly beneficial when updates are close to each other (localized) because it reduces repeated unsuccessful explorations.

Tesseract applies all updates with the same timestamp atomically to create a graph snapshot. Then, we explore the graph and emit matches at snapshot granularity, which allows skipping work for intermediate matches between snapshots. To do so, the input graph snapshot in the EXPLORE Algorithm 2 contains all the updates at timestamp ts, while the previous graph snapshot excludes all these updates.

For each snapshot, a worker still explores each update separately while ensuring that matches overlapping more than one update in the same graph snapshot are only found from one of the updates. This is achieved by imposing a strict total order on the graph's edges (in addition to the total order on update timestamps), thus ensuring that one starting edge in a match takes precedence over the others. The CAN_EXPAND function in Algorithm 2 checks whether an expansion edge is in the same snapshot by comparing the timestamps and returns false if that edge is lower than the starting (update) edge. The total order can be assigned in different ways, e.g., based on the vertex ids of the edges.

Example Consider triangle mining (Algorithm 1) and an input graph to which we add a triangle (1, 2, 3). We first examine the case where we add the three edges making up the triangle one at a time. We start by adding (1, 2), which cannot be expanded as there is no edge to vertex 3. We then add (1, 3) and expand to the subgraph (1, 2, 3). This subgraph does not pass `filter` because it is missing an edge between vertices 2 and 3. Finally, we add (2, 3) and expand again to the subgraph (1, 2, 3). This time, the subgraph passes the `filter` and is emitted. Next, we consider adding all three edges in the same snapshot. In this case, we assume that the total edge ordering is such that $(1,\ 2) < (1,\ 3) < (2,\ 3)$. Tesseract executes exploration from each edge, similar to when updates arrive one at a time. We start exploring from the edge (1, 2). The subgraph (1, 2, 3) passes the `filter` and is emitted. Exploring from (1, 3) cannot expand with vertex 2 since edge (1, 2) is lower. Similarly, exploring from (2, 3) cannot expand with vertex 1 since both edges (1, 2) and (1, 3) are lower. Comparing the two executions, we find that snapshot-based exploration invokes EXPAND and DETECT_CHANGES once while adding edges one at a time incurs this cost twice.

4.4.4 Subgraph Expansion Rules

Algorithm 3 implements the CAN_EXPAND function used in Algorithm 2 and eliminates duplicates using the techniques described above. Executing CAN_EXPAND before EXPAND avoids paying the cost of expanding duplicates.

Algorithm 3: CAN_EXPAND

```
input  : G data graph snapshot at timestamp ts
input  : s subgraph
input  : ts update timestamp
input  : v new vertex to expand s
1 foreach edge (v,u) in G with u ∈ s do
2     if TIMESTAMP(v,u) == ts and (v,u) < (s[0],s[1])
        then return false
3 found ← IS_NEIGHBOR(G, v, s[0]) or
    IS_NEIGHBOR(G, v, s[1])
4 foreach u in s[2:] do
      // s[2:] excludes the update endpoints
5     if not found and IS_NEIGHBOR(G, v, u) then
6         found ← true
7     else if found and u > v then return false
8 return true
```

The first loop rejects expansions along an edge that is part of the same snapshot as the initial edge (and therefore has the same timestamp) if that edge is lower according to the strict ordering (§4.4.3). The second loop implements the second update canonicality rule by checking if the vertex considered for addition has a larger identifier than the preceding expansion vertices (§4.4.1). This check is achieved by looking for the first neighbor of vertex v in the subgraph s (lines 3 and 5) and, when found, ensuring that no subsequent vertex in s has a larger identifier (line 7). The first update canonicality rule is ensured by always starting exploration from the updated edge. Finally, duplicate elimination for several updates involving the same match (§4.4.2) is guaranteed because this algorithm executes on a snapshot of the graph at the timestamp of the updated edge.

4.5 Parallel Exploration

So far, we have focused on exploration running on a single threaded worker. The techniques presented in the previous section allow parallel exploration across multiple distributed workers to execute with no other changes because Tesseract ensures that the exploration task (Algorithm 2) for each update is independent of others and can be executed in any order. This independence property is achieved by the combination of our mechanisms for change detection and duplicate elimination (via timestamps and the multiversioned store). This property enables multiple workers to execute EXPLORE on disjoint updates, even updates in the same snapshot, while getting the same behavior as if this algorithm was executed sequentially at a single worker. Therefore, in Tesseract, parallel exploration is achieved for free because each exploration task for a given graph update is completely independent from the others.

5 Implementation

Tesseract leverages Apache Spark Structured Streaming [71], a stream processing engine, to provision nodes and provide an execution environment for workers. Tesseract complements Spark with a graph processing engine designed specifically for graph pattern mining. Since Tesseract only relies on commonly available operators, it can be implemented on any streaming or dataflow engine.

Each Tesseract worker executes Algorithm 2 independently. An idle worker picks the next update in the work queue and processes it to output the corresponding changes in the match set. Workers access the graph store to obtain the necessary structure and labels to perform the exploration algorithm. Tesseract is implemented in about 8k lines of C++ library code for the graph mining engine and 1k lines of Scala code for distributed execution and interfacing to Spark.

In the rest of this section, we describe the implementation of the individual components of Tesseract as seen in Figure 2.

5.1 Ingress Node

The ingress node sanitizes incoming graph updates and applies them to the graph store in timestamp order. Timestamp assignment is user-customizable: each update can be timestamped with an increasing integer value or a window of several updates with the same timestamp can be created, either based on time intervals or number of updates. The ingress node also garbage collection of old deleted edges.

5.2 Graph Store

Tesseract's graph store is based on MongoDB [3], a popular key-value store, and uses an adjacency list format. Each vertex record maintains a list of outgoing edges, identified by the destination endpoint of the edge, and the edge timestamp and associated labels. Deleted edges are kept but marked with a special flag. We validate experimentally that MongoDB can ingest updates at a sufficient rate, so it is not a bottleneck.

Since workers operate on an in-memory graph representation, Tesseract could also be deployed over other distributed databases [4, 5, 11] or graph databases [6, 69, 74] by using their storage interface. However, graph stores designed for analytics that do not support transactions [40, 59] are unsuitable for our purposes.

5.3 Work Queue

We implement the work queue using Apache Kafka [37] to ensure durability of updates and exactly-once delivery to workers. The work queue offers first-in-first-out (FIFO) semantics and updates are ordered by timestamps, i.e., any pull operation from the queue is guaranteed to receive an update with a timestamp lower or equal to the timestamp of all other updates in the queue. The single work queue is not a bottleneck as the work per update is significant.

Dynamic Work Assignment In Tesseract, any worker can process any update because each worker can access any part of the input graph from the graph store. Therefore, it is unnecessary to partition the updates in the queue. Provided that there are sufficiently many updates, our scheme keeps all workers busy while ensuring that the load remains balanced across workers.

5.4 Output Processing

The output processing and aggregation API (Table 2) is implemented using Spark Structured Streaming. We leverage Kafka on the output side to store emitted matches and provide a durable publish-subscribe platform to users.

Output Ordering Since the work queue is ordered by timestamps, each worker also outputs graph updates in FIFO order. However, processing different graph updates may require different amounts of time and changes in the match set may be emitted out-of-order across workers. The publish-subscribe platform supports reordering of matches by timestamps. Reordering is used, for instance, in the FSM algorithm (§3.3) to maintain consistent support values across updates. Tesseract supports low watermarks to provide a synchronization point guaranteeing that all updates with a timestamp lower or equal to a target timestamp have been emitted.

5.5 Fault Tolerance

Fault tolerance is an essential concern for high-throughput evolving graph mining systems that may execute for long periods of time. Tesseract maintains state in the graph store, as well as soft state in the workers. The graph store is replicated and sharded on worker machines and can be recovered in case of failures. The graphs cached at workers can be lost without affecting correctness. Tesseract relies on Spark to handle worker crashes and to restart and redistribute work. We ensure exactly-once semantics for updates using Kafka.

5.6 Optimizations

Tesseract optimizes operations on subgraphs and their memory footprint by storing the edges connecting vertices in the subgraph in a bitset representing that subgraph's adjacency matrix. Our exploration uses a fixed-size bitset. Many operations on subgraphs such as counting the number of edges or computing the degree of a vertex are implemented using bitwise operators. Bitwise operations also make it cheap to expand subgraphs and backtrack row-by-row.

Tesseract implements its own optimized motif library to identify motifs (e.g., for motif counting), but also supports other implementations such as bliss [35].

6 Evaluation

We compare Tesseract to existing graph mining systems on various input graphs and algorithms. As Tesseract is the first distributed general graph mining system for evolving graphs (See Table 1), a direct comparison is not possible. Therefore, we compare it with three types of systems. First, we compare with recent static, distributed, general mining systems (Arabesque [64] and Fractal [24]) that require periodic full computation on the entire graph (§6.2). We show that Tesseract outperforms these systems on static graphs and provides orders-of-magnitude performance improvements compared to periodic full computation. Second, we compare with Delta-BigJoin [10], the closest system that supports evolving graphs for distributed subgraph queries (§6.3). We show that Tesseract is faster and more expressive than Delta-BigJoin. Third, we compare with Peregrine [34], a recent, single node, static mining system (§6.4) and demonstrate that Tesseract outperforms Peregrine's performance in a fair comparison when both systems materialize the output. Fourth, we show how Tesseract incrementally mines large graphs with trillions of matches. Finally, we perform a scalability and sensitivity analysis (§6.5).

6.1 Experimental Setup

We use the following four common mining applications:

k-clique enumeration (*k*-C): Find all cliques (fully connected subgraphs) of a given size k within a graph. We also consider an extended version of this algorithm in which all vertices in the clique must have distinct labels (***k*-CL**).

Graph keyword search (*k*-GKS-*n*): Enumerate all minimal subgraphs whose vertices contain all n labels of interest. We refer to GKS with subgraphs of size $\leq k$ as k-GKS-n.

Motif counting (*k*-MC): Count the number of times each motif appears in a graph. We refer to motif counting with motifs of size $\leq k$ as k-MC.

Frequent subgraph mining (*k*-FSM-*s*): Enumerate all frequent subgraphs in a graph. A frequent subgraph has a minimum image-based support [18] larger than a threshold s. We refer to FSM with subgraphs of size $\leq k$ as k-FSM-s.

k-C is implemented as per Algorithm 1. k-GKS-n is also implemented as per Algorithm 1, and an example of 5-GKS-3 is shown in Figure 1. k-MC and k-FSM-s are described in Section 3.3. k-FSM-s uses edge-induced subgraphs.

These algorithms cover a wide spectrum of workloads, including varying selectivity (k-C vs. k-CL), graph querying (k-GKS-n), counting (k-MC), and aggregation (k-FSM-s).

Datasets Table 3 lists the graph datasets we used in the experiments. We use graphs with various sizes and characteristics representing different real-world use cases. Many of these datasets are used by other graph mining systems to evaluate their performance [10, 24, 34, 46, 64, 67]. Note that given the exponential search nature of graph mining, even relatively small datasets can contain a large number of matches. For example, LiveJournal has ~148 billion 5-cliques.

Dataset	Vertices	Edges	Domain
LiveJournal (LJ) [1]	4.8M	68.9M	social network
UK-2007 (UK) [16, 17]	106M	3.7B	web hyperlinks
DC-2012 (DC) [7]	3.5B	128B	web hyperlinks

Table 3. Datasets.

For k-GKS-n, we assign labels to nodes randomly, uniformly across all k so that 1/8th of the nodes are labeled. We simulate a dynamic graph by loading and applying a shuffled

subset of the edges (and associated vertices) of a static graph iteratively until the entire graph is constructed. We simulate deletions in a similar way by deleting a shuffled subset of edges already present in the graph.

Hardware and Configuration We evaluate Tesseract using 8 16-core machines (2 Xeon E5-2630), each equipped with 128 GB of DDR3 ECC main memory and two 500GB SSDs. All components of Tesseract, including the graph store, ingress, work queue, and output processing, run on the same 8 machines. We do not run anything else on these machines. By default, we create snapshots with a window size of 100K updates. For reproducibility, we assign increasing timestamp values to snapshots.

6.2 Static, General, Distributed Graph Mining

We first evaluate the performance benefits of Tesseract's incremental computation approach by comparing it with periodic full computation by static systems. To this end, we consider two state-of-the-art general distributed graph mining systems for static graphs: Fractal [24] and Arabesque [64].

6.2.1 Distributed Static Graph Baseline

Table 4 shows the time taken by each system to fully process the LJ dataset for the 4-C, 4-MC, and 4-FSM-2K algorithms using 8 machines. We do not show the CL or GKS algorithms as they are not implemented in Arabesque and Fractal. Tesseract runs static graphs by loading all edges as edge addition updates into an initially empty graph.

Algorithm	Arabesque	Fractal	Tesseract
4-C	4.9h	310s	174s
4-MC	— †	12.3h	1.9h
4-FSM-2K	— †	23.7h	10.3h

Table 4. Arabesque, Fractal, and Tesseract runtime for 4-C, 4-MC, and 4-FSM-2K on the LJ dataset, using 8 machines. Arabesque runs out of memory on LJ for 4-MC and 4-FSM-2K (†).

Despite being designed specifically for evolving graphs, Tesseract is faster on static graphs than Arabesque and Fractal because its workers run explorations independently. Unlike Arabesque, Fractal uses a depth-first exploration strategy that avoids storing and shuffling large amounts of intermediate state. However, Fractal workers coordinate with each other via an application master, resulting in high network traffic and introducing a bottleneck on the master.

6.2.2 Benefits of Incremental Computation

We now demonstrate the benefits of Tesseract's incremental computation approach over static mining for evolving graphs. We select Fractal for our comparison since it is faster than Arabesque. We first create 90% of the LJ graph. Then, we simulate an evolving graph by adding the remaining edges to the graph in 0.1%, 1%, or 10% increments. A 1% increment corresponds to 689K updates. Figure 3 shows the time for each system to process a single increment on a log scale for 4-C and 4-FSM-2K on the LJ dataset. Since Fractal is a static mining system, it must fully compute the result on the entire graph from scratch after every increment.

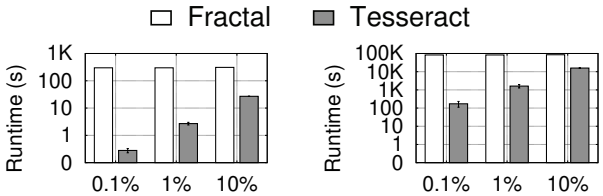

Figure 3. Fractal and Tesseract runtime for 4-C and 4-FSM-2K on the LJ dataset using 8 machines. We show the average time to process increments of 0.1%, 1%, and 10%. For Tesseract, we also plot the range over all increments (min to max).

These results demonstrate the benefits of Tesseract's incremental computation over periodic full computation. The time to process a single increment in Tesseract is low compared to full computation and does not vary significantly. For 4-C, incremental computation improves runtime by 11.5× for a single increment of 10%, by over 110× for a 1% increment, and by 1,067× for a 0.1% increment. For 4-FSM-2K, Tesseract requires 4.4h instead of 23.7h (5.3× faster) to process the remaining 10%, 27 minutes for 1% (51×), and 172s for 0.1% (483×). In the latter case, our incremental approach involves more work to maintain the matches due to certain motifs crossing above the support threshold and requiring that we mine and emit all corresponding matches. Mining a single motif can be done much more efficiently than mining all motifs. For example, enumerating all cliques of 4 vertices on the LJ graph takes ∼5 minutes.

6.3 Evolving, Distributed Subgraph Queries

We compare Tesseract with Delta-BigJoin [10], a state-of-the-art subgraph query system based on Timely Dataflow [49]. Unlike Tesseract where patterns are defined with arbitrary code, BigJoin is a fixed subgraph query system that matches a single isomorphic subgraph. For example, counting all 4-motifs in a graph requires six separate subgraph queries, one for each (undirected) motif shown in Figure 4. A query for motif A would be q_A `:= e(a,b), e(b,c), e(c,d)` while a query for motif F would be q_F `:= e(a,b), e(a,c), e(a,d), e(b,c), e(b,d), e(c,d)`, assuming `e(x,y)` defines an edge relation. Moreover, to support evolving graphs, each subgraph query is decomposed into multiple delta-queries, one for each edge relation. As a result, BigJoin requires 6 separate subgraph queries for 4-motif counting and executes 25 delta-queries (one for each edge in Figure 4) to support updates. The situation worsens for mining algorithms with labels. For instance, finding all matches of 5-GKS-3 requires 98 subgraph queries and 743 delta-queries.

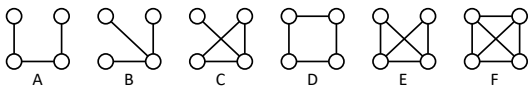

Figure 4. All 6 possible 4-motifs.

Figure 5 compares the runtimes of various algorithms for Delta-BigJoin and Tesseract on the LJ dataset using 8 machines. We use Delta-BigJoin's existing 4-C implementation, and implement optimized versions of 4-CL, 4-MC, and

5-GKS-3. BigJoin does not provide native support for FSM-style aggregation, so we do not consider the FSM algorithm in this comparison. We fully replicate the input graph on all machines for Delta-BigJoin. With BigJoin, we need to issue separate queries for each possible isomorphic combination (6 for 4-MC and 98 for 5-GKS-3). Although BigJoin currently does not support running multiple queries in parallel, the system could theoretically factor out some subqueries. Therefore, we opt to report the runtime of the slowest query (best case) for algorithms where multiple queries are needed.

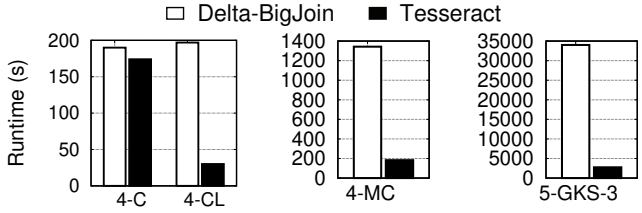

Figure 5. Delta-BigJoin and Tesseract runtime for different algorithms on the LJ dataset using 8 machines. For Delta-BigJoin in 4-MC and 5-GKS-3, we show the runtime of the slowest query when it needs to run 6 and 98 respectively.

Tesseract is faster than Delta-BigJoin for 4-C by 1.1× and 6.5× for 4-CL. Our performance gains on 4-CL come from the general programming model that reduces the search space significantly when mining algorithms are not looking for purely structural patterns. In 4-CL, when expanding subgraphs by candidate vertices, Tesseract checks immediately using `filter` whether the vertex label matches a label already in the subgraph, and, if so, backtracks. In contrast, Delta-BigJoin must materialize all matching subgraphs before checking in a second, post-processing step that the labels are all distinct. Tesseract fares even better than 4-C and 4-CL for larger k-C and k-CL (not shown). For 4-MC, Tesseract, when executing an algorithm equivalent to all 6 queries, is over 7× faster than Delta-BigJoin's slowest query and only 18% slower than the fastest query. When considering all 6 queries executed sequentially for 4-MC, Tesseract is 26× faster. For 5-GKS-3, Tesseract outperforms Delta-BigJoin, mining all 98 matches over 12× than Delta-BigJoin executes its slowest query and 2× faster for the fastest query.

Impact of Data Shuffle Tesseract workers operate without coordination, while Delta-BigJoin has significant communication requirements. We confirm this intuition by profiling Delta-BigJoin's network usage and observe that the system runs almost constantly at high network bandwidth. In the 4-C experiment on LJ, Delta-BigJoin workers exchange 280 GB of data across the network, and in the 5-GKS-3 experiment, over 15 TB of data are exchanged. Despite Delta-BigJoin's ability to overlap computation and communication, such large data shuffles result in significant overhead. In comparison, Tesseract workers only synchronize to pull graph updates from the work queue, requiring data exchanges in the order of the graph size (a few gigabytes). This data shuffle overhead is the reason why Delta-BigJoin is unable to finish executing larger graphs than LJ in our environment.

6.4 Static, Single-Node, General Graph Mining

We compare Tesseract running on a single node with Peregrine [34], a recent, high-performance, static, general graph mining system. Peregrine is currently the fastest single-node mining system whose source code is available, and it consistently outperforms AutoMine [46] (source code unavailable).

Table 5 shows the time taken by each system to process the entire static LJ dataset for 4-C and 4-MC on a single machine. Peregrine crashes on 4-FSM-2K. We also do not show CL or GKS as they are not implemented in Peregrine, and the comparison would likely be unfair to Peregrine due to its limited label support. We also compare with PeregrineMat, a modified version of Peregrine that materializes and outputs all matches. PeregrineMat offers a more apples-to-apples comparison with Tesseract as Peregrine only counts the number of matches for a given pattern, which is significantly faster.

Algorithm	Peregrine	PeregrineMat	Tesseract
4-C	473s	1855s	1015s
4-MC	2.6h	>24h	12.3h

Table 5. Peregrine, PeregrineMat, and Tesseract runtime for 4-C and 4-MC on the LJ dataset using a single machine.

Tesseract is only ∼2.1× and ∼4.7× slower than Peregrine for 4-C and 4-MC, respectively, despite materializing and outputting all matches, supporting evolving graphs, and distributed execution. In comparison, PeregrineMat is slower in both cases. We also report Tesseract's COST metric, defined as the number of threads at which a system outperforms an efficient single-threaded implementation [47]. When compared with PeregrineMat, Tesseract achieves a COST of 3 for 4-C and 5 for 4-MC.

6.5 Scalability and Sensitivity Analysis

This section evaluates the scalability, the overhead of supporting dynamic updates, the throughput-latency tradeoff, the ingress overhead, and the deletion performance.

6.5.1 Incrementally Mining Large Graphs

We evaluate Tesseract's ability to maintain the match set in large evolving graphs by only generating the changes in the match set for graph updates. We consider the UK and DC datasets that contain trillions or quadrillions of matches for simple patterns such as 3-cliques or 4-cliques. A complete enumeration of these matches could require days. Therefore, we do not aim to enumerate all the matches in such large graphs, but rather maintain the mining result under updates.

We first load all but 10M edges of the graph in the graph store, without computing the associated matches. Next, we apply the remaining edges as updates to the graph and produce the associated changes in the match set. Table 6 shows the average time to process 1M edge updates and the corresponding output rate on the UK and DC datasets for 4-C and 5-GKS-3. We perform these experiments both using a single machine and a distributed deployment with 8 machines.

#	Metric	UK		DC	
		4-C	5-GKS-3	4-C	5-GKS-3
1	Time	1,428s	2,905s	2.7h	8.5h
	Out. rate	726K/s	1.52M/s	486K/s	985K/s
8	Time	168s	372s	993s	1.5h
	Out. rate	5.61M/s	11.4M/s	4.73M/s	7.57M/s

Table 6. Average time and output rate when processing 1M updates on UK and DC for 4-C and 5-GKS-3 using 1 and 8 machines.

Tesseract outputs matches at a rate of millions per second with 8 machines and over half a million using a single machine. The output rates scale almost linearly (7.7× and 7.5×) with the number of machines for the UK dataset. We observe superlinear scaling (9.7× and 8.9×) for the DC dataset as workers collectively have more memory capacity, which reduces the number of times a worker fetches data from the graph store. The average processing time increases significantly for DC compared to UK as expected since each update involves significantly more matches and requires more processing. While the difference in size between both datasets is 34×, the average processing time per window only increases by 10-14×. Finally, we execute the 4-CL algorithm on the same datasets to evaluate the performance of an algorithm with higher selectivity. 4-CL runs 8.2× (UK) and 7.9× (DC) faster than 4-C while maintaining a comparable output rate. In conclusion, Tesseract provides a practical solution for maintaining mining results for large evolving graphs.

6.5.2 Scalability and Breakdown of Operations

We run Tesseract in our cluster using the LJ dataset and 4-C and 5-GKS-3 to show how the system scales as we increase the number of nodes from 1 to 2, 4, and 8. Figure 6 shows the runtime for each experiment, broken down into the different operations described in Algorithm 2.

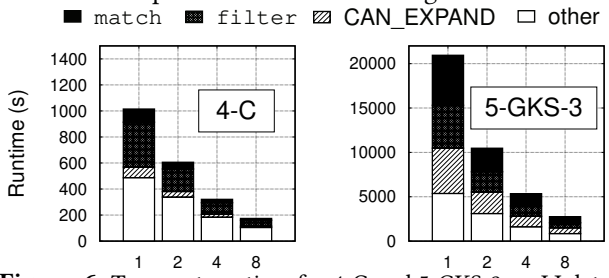

Figure 6. Tesseract runtime for 4-C and 5-GKS-3 on LJ dataset with increasing number of machines.

Tesseract processes 4-C 7.3× faster and 5-GKS-3 7.6× faster on 8 machines than on a single one. We conclude that Tesseract scales almost linearly with an increasing number of machines. The scalability is not perfectly linear due to overhead from accessing and constructing the set of neighbors for a subgraph, emitting matches, and dequeuing updates. This overhead is partially reflected in the breakdown per operation as the time spent executing other operations (*other*). While `match`, `filter`, and CAN_EXPAND are on average 7.5× faster for 4-C and 8.2× faster for 5-GKS-3 on 8 machines compared to one, the other operations are only 5.7× faster for 4-C and 6.3× faster for 5-GKS-3.

6.5.3 Overhead of Supporting Dynamic Updates

We measure the overhead to support dynamic updates by comparing Tesseract's performance for 4-C on LJ using a single machine with STesseract, an optimized version of Tesseract designed to mine static graphs. STesseract executes EXPLORE for each edge in the graph. The system also does not perform any differential processing or use snapshots, and it only includes lines 3 to 8 of the `can_expand` function (Algorithm 3). While the original Tesseract finishes in 1,015s, STesseract only requires 724s to execute the algorithm, a 29% slowdown. We expect the overhead of supporting evolving graphs to be between 25% and 50% for most algorithms.

6.5.4 Latency-Throughput Tradeoff

Tesseract supports snapshot-based exploration, which helps improve throughput by reducing repeated unsuccessful exploration of matches but at the cost of higher latency for emitting matches (§4.4.3). This approach also amortizes the cost of reading from the graph store as each worker can use the same snapshot to process multiple graph updates. Note that Tesseract allows the graph updates in a snapshot to be dynamically assigned to multiple workers, avoiding load imbalance across machines.

We compare the throughput and latency for 4-C on the LJ dataset using 8 machines. Tesseract processes 133 million matches per second with a window size of 10K updates, 142 million matches per second with 100K window size and 155 millions matches per second with 1M window size (a throughput improvement of 17%). The mean latency to process the updates in a window increases almost linearly with the window size. For instance, a 10K window takes roughly 311ms of processing time, a 100K window takes 2.91s, and a 1M window takes 26.9s. Tail latencies (P99) remain within 10% of the mean for window sizes larger than 1K since snapshot-based exploration amortizes processing latency for updates that impact many matches. This experiment also justifies our default window size (100K) as it provides a good compromise between throughput and latency.

6.5.5 Ingress Scalability

A possible concern with Tesseract's ingress node and work queue design is that it introduces a scalability bottleneck due to the linearization of updates. Intuitively, this is not the case because graph mining is CPU intensive, i.e., computing the matches for an update involves orders of magnitude more work than simply timestamping updates. The fastest graph mining algorithm in this paper is 4-CL, executing in 30s on 8 machines (Figure 5). In this experiment, since the LJ graph has 68.9 million edges (Table 3) the aggregate ingest rate for all workers is $68.9/30 = 2.3$ million updates per second.

In order to verify that our ingress node is not a bottleneck, we execute an empty algorithm that does not do any processing or matching of updates and measure the ingest rate. We

find that Tesseract workers can ingest ∼1.2 and ∼7.4 million updates per second, respectively on 1 and 8 machines in our environment with an empty algorithm. This ingest rate is, therefore, quite sufficient for most graph algorithms. Moreover, we point out that our ingress node design is simple and could benefit from many optimizations in a production environment, such as partitioning of the update stream.

6.5.6 Deletions

Tesseract supports both additions and deletions to the input graph. We verify that processing deletions takes the same time as processing additions by adding all edges of the LJ dataset into an initially empty graph before deleting all these edges. Tesseract takes 2,756s to process all additions with the 5-GKS-3 algorithm on 8 machines and another 2,510s to process all deletions when updates are applied in reverse order. We run the same experiment while deleting updates in a randomly selected order and find that processing all the deletions takes 3,014s. Randomly ordered deletions cause matches to be emitted in a different order, creating and deleting additional matches, resulting in a 20% slowdown.

7 Related Work

Graphs are used for representing and analyzing information and relationships in a wide variety of domains [58]. There are two broad classes of graph problems: graph analytics and graph mining. Graph analytics aim to compute various graph-wide properties, usually through iterative matrix-vector multiplication. Examples of such problems include PageRank [53], connected components, and community detection. A wide variety of graph processing frameworks [20, 27, 28, 41, 43–45, 52, 56, 57, 61, 63, 72, 73, 75], have been developed large scale graph analytics. Their design is based on think-like-a-vertex, making them unsuitable for mining algorithms that require enumerating subgraphs [64].

State-of-the-art general, graph mining systems such as Arabesque [64], Fractal [24], and RStream [67] use graph-wide exploration and enumerate *all* the matches in the graph at the same time. In Arabesque, this approach enables parallelism via BSP-style phased execution, with subgraphs being built incrementally in each phase, by adding one vertex or one edge at a time. In RStream, it enables storing and streaming subgraphs from disk in a sequential manner. Fractal [24] uses a depth-first search approach to enumerate embeddings, which reduces memory footprint and subgraph enumeration costs. Recent single-node approaches such as AutoMine [46], Pangolin [21], and Peregrine [34] have proposed compiler techniques or pattern-specific optimizations to improve performance over the state-of-the-art. All these systems focus on static graphs, whereas Tesseract targets evolving graphs.

Several graph query systems [10, 31, 39, 42] use relational methods for supporting subgraph queries by expressing patterns as a relation query over the graph edges, and generate matches by joining the edge table. These systems solve a subclass of graph mining problems where patterns are expressed as a fixed subgraph query. BigJoin [10] performs subgraph queries over static graphs using the GenericJoin algorithm [51] to provide worst-case optimal performance guarantees. BigJoin is implemented using Timely Dataflow [49], and is especially effective for purely structural queries that just involve joins, since the joins are run in parallel. However, this parallel execution makes it harder to filter embeddings efficiently, requiring the joined tuples to be generated before they can be filtered. Some of these distributed graph query systems [10] have recently been adapted to support evolving graphs by leveraging delta queries [15, 29]. There has been much work on improving subgraph queries performance over evolving graphs by storing information about query vertices in the vertices or edges of the graph [39, 62]. Some specialized systems can accommodate continuous queries [8, 38, 60, 68, 70]. ASAP [33] is a fast, approximate subgraph query system that estimates the number of pattern matches in a graph, and it provides an error profile that allows trading accuracy for query runtime. It has good performance due to sampling, but it cannot be used to enumerate subgraphs, and it has limited support for labels. While ASAP is primarily designed for static graphs, it can be extended to evolving graphs by rebuilding its error profile on updated graphs.

Tesseract's compute-storage disaggregation strategy is inspired by the Scatter architecture [12–14, 56]. Unlike in these applications, storage is never the bottleneck in graph mining. However, separating the graph store and keeping only soft state at each worker simplifies load balance and avoids partitioning.

8 Conclusions

We presented Tesseract, the first distributed and general graph pattern mining system designed for evolving graphs. Tesseract maintains the match set incrementally as the input graph receives updates. Tesseract performs change detection for a single update efficiently and assigns updates dynamically to a set of distributed workers that operate independently without exploring duplicates. We demonstrate that our incremental mining system provides orders-of-magnitude improvements over existing mining systems.

Acknowledgments

We would like to thank our anonymous reviewers, our shepherd, Marina Papatriantafilou, as well as Marco Serafini, Lijun Chang, Mario Bucev, Calin Iorgulescu, and Florin Dinu, for their feedback that improved this work. We also thank Anne-Marie Kermarrec for providing the cluster used for our experiments. This work was supported in part by the Swiss National Science Foundation NRP 75 Grant No. 167157, by an SNSF Early Postoc.Mobility Fellowship Grant No. P2ELP2_-195136, and by the Australian Research Council Discovery Project DP210101984.

References

[1] 2021. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[2] 2021. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[3] 2021. https://www.mongodb.com/.

[4] 2021. http://cassandra.apache.org/.

[5] 2021. https://www.aerospike.com/.

[6] 2021. https://dgraph.io/.

[7] 2021. http://webdatacommons.org/hyperlinkgraph/.

[8] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 61.

[9] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (2008), i241–i249.

[10] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018), 691–704.

[11] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: The Definitive Guide: Time to Relax.* " O'Reilly Media, Inc.".

[12] Laurent Bindschaedler. 2020. *An Architecture for Load Balance in Computer Cluster Applications*. Technical Report. EPFL.

[13] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. ACM, 301–316. https://doi.org/10.1145/3373376.3378504

[14] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock You Like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the 13th EuroSys Conference (EuroSys '18)*. ACM, Article 20, 20 pages.

[15] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. In *ACM SIGMOD Record*, Vol. 15. ACM, 61–71.

[16] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press.

[17] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[18] Björn Bringmann and Siegfried Nijssen. 2008. What is frequent in a single graph?. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 858–863.

[19] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.

[20] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 1.

[21] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* 13, 10 (April 2020), 1190–1205. https://doi.org/10.14778/3389133.3389137

[22] Young-Rae Cho and Aidong Zhang. 2010. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Transactions on information technology in biomedicine* 14, 1 (2010), 30–36.

[23] Michael Codish, Alice Miller, Patrick Prosser, and Peter James Stuckey. 2013. Breaking symmetries in graph representation. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

[24] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 1357–1374.

[25] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 155–169.

[26] Gary William Flake, Steve Lawrence, C Lee Giles, and Frans M Coetzee. 2002. Self-organization and identification of web communities. *Computer* 3 (2002), 66–71.

[27] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 17–30.

[28] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[29] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

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

[31] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 337–348.

[32] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *European conference on principles of data mining and knowledge discovery*. Springer, 13–23.

[33] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. {ASAP}: Fast, Approximate Graph Pattern Mining at Scale. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 745–761.

[34] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[35] Tommi Junttila and Petteri Kaski. 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 135–149.

[36] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. 2005. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 505–516.

[37] Apache Kafka. 2014. A high-throughput distributed messaging system. *URL: kafka. apache. org as of* 5, 1 (2014).

[38] Mehdi Kargar, Lukasz Golab, and Jaroslaw Szlichta. 2015. Effective keyword search in graphs. *arXiv preprint arXiv:1512.06395* (2015).

[39] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 411–426.

[40] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *USENIX Conference on*

File and Storage Technologies, (FAST).

[41] Aapo Kyrola and Guy Blelloch. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the Conference on Operating Systems Design and Implementation*. USENIX Association.

[42] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 10, 3 (2016), 217–228.

[43] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Daniel Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. https://arxiv.org/abs/1006.4990

[44] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud.. In *Proceedings of Very Large Data Bases (PVLDB)*. https://arxiv.org/abs/1204.6078

[45] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 631–643.

[46] Daniel Mawhirter and Bo Wu. 2019. AutoMine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

[47] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[48] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[49] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[50] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal—The International Journal on Very Large Data Bases* 19, 1 (2010), 91–113.

[51] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: new developments in the theory of join algorithms. *ACM SIGMOD Record* 42, 4 (2014), 5–16.

[52] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Symposium on Operating Systems Principles*. ACM, 456–471.

[53] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[54] Clifton Phua, Vincent Lee, Kate Smith, and Ross Gayler. 2010. A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119* (2010).

[55] Nataša Pržulj, Derek G Corneil, and Igor Jurisica. 2004. Modeling interactome: scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515.

[56] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 410–424.

[57] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the ACM symposium on Operating Systems Principles*. ACM, 472–488.

[58] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[59] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. [n.d.]. GraphIn: An Online High Performance Incremental Graph Processing Framework. ([n. d.]).

[60] Saeed Shahrivari and Saeed Jalili. 2015. Distributed discovery of frequent subgraphs of a network using MapReduce. *Computing* 97, 11 (2015), 1101–1120.

[61] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 135–146.

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

[63] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.

[64] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 425–440.

[65] Mikkel Thorup. 1999. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)* 46, 3 (1999), 362–394.

[66] Haixun Wang and Charu C Aggarwal. 2010. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*. Springer, 249–273.

[67] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 763–782.

[68] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and Concurrent {RDF} Queries using RDMA-assisted {GPU} Graph Exploration. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 651–664.

[69] Jim Webber. 2012. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 217–218.

[70] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. 2013. Scalable maximum clique computation using mapreduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 74–85.

[71] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[72] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 183–193.

[73] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*.

[74] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 (Mar 2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

[75] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.