SUBGRAPH SEARCH FOR DYNAMIC GRAPHS

by

SUTANAY CHOUDHURY

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
Department of Electrical Engineering and Computer Science

MAY 2014

© Copyright by SUTANAY CHOUDHURY, 2014
All Rights Reserved
To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of SUTANAY CHOUDHURY find it satisfactory and recommend that it be accepted.

__________________________
Lawrence B. Holder, Ph.D., Chair

__________________________
Ananth Kalyanaraman, Ph.D.

__________________________
John Feo, Ph.D.
Abstract

by Sutanay Choudhury, Ph.D.
Washington State University
May 2014

Chair: Lawrence B. Holder

Subgraph search is the problem of searching a data graph for the occurrences of another graph, typically referred to as the query or pattern graph. This thesis is dedicated to studying a specific class of subgraph search, named subgraph isomorphism for dynamic graphs, i.e., graphs that are evolving over time. Subgraph isomorphism is a well studied NP-complete problem in computer science. It requires finding a bijective mapping between the vertices in the query graph and the data graph so that if two vertices are neighbors in the query graph, their mapped counterparts are neighbors in the data graph as well. Our research on dynamic graphs is motivated by large-scale graph data sources such as social media and cyber-security where applications continuously produce prodigious amounts of data. This thesis makes three major contributions. We begin with proposing a new subgraph isomorphism algorithm for dynamic graphs and a novel data structure, namely the
Subgraph Join-Tree (SJ-Tree) to support the algorithm. Next, we demonstrate how the statistics of the graph stream can be leveraged to produce the best-performing SJ-Tree for a given query graph. We propose a metric for estimating the selectivity of a graph query and demonstrate its use to reason about the relative performance of different query execution strategies. Our observations are supported by experiments on multiple real-world data sources drawn from online news, social media and network traffic. Our experiments demonstrate speedups by 10-100 times over existing approaches. Finally, we propose a new algorithm and supporting data structures to implement the aforementioned dynamic graph search algorithm on a distributed system. As a secondary contribution, we demonstrate how the novel ideas from the graph search paradigm can be applied to discover patterns in dynamic graphs. The thesis concludes with presentation of real-world use cases from three different application domains.
# TABLE OF CONTENTS

ABSTRACT .................................................................................................................. iii

LIST OF TABLES .......................................................................................................... viii

LIST OF FIGURES ........................................................................................................ ix

1. INTRODUCTION ..................................................................................................... 1
   1.1 Motivation .......................................................................................................... 3
   1.2 Research Contributions ..................................................................................... 5

2. BACKGROUND AND RELATED WORK ................................................................. 7
   2.1 Definitions .......................................................................................................... 8
   2.2 Anatomy of a graph matching algorithm ......................................................... 10
   2.3 Indexing ............................................................................................................ 14
   2.4 Query Pre-Processing ....................................................................................... 25
   2.5 Graph Matching and Search ............................................................................ 29
   2.6 Continuous Queries ......................................................................................... 36
   2.7 Dynamic Graph Search .................................................................................... 38

3. DATA STRUCTURES AND ALGORITHMS FOR DYNAMIC GRAPH SEARCH .... 40
   3.1 Problem Statement ........................................................................................... 44
   3.2 Subgraph Join Tree (SJ-Tree) .......................................................................... 45
   3.3 Continuous Query Algorithm .......................................................................... 50
   3.4 Experimental Results ....................................................................................... 57
   3.5 Summary ........................................................................................................... 67
8. CONCLUSIONS AND FUTURE WORK ........................................ 186

8.1 Future Work .............................................................. 188
LIST OF TABLES

3.1 Summary of test datasets .................................................. 57
4.1 Summary of test datasets .................................................. 87
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Salient features of the streaming graph database</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Illustration of the search process for Subgraph Isomorphism</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>A graph query for monitoring emergencies in social media and news streams.</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustration of a naive incremental algorithm. Assume AB matches with PQ, and AC matches with both QR and QS.</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of query decomposition in SJ-Tree.</td>
<td>48</td>
</tr>
<tr>
<td>3.5</td>
<td>a) Example query b) Occurrence of a match in the search graph c) Combining two partial matches to form the complete match.</td>
<td>50</td>
</tr>
<tr>
<td>3.6</td>
<td>Example SJ-Tree for the query shown in Figure 3.5a.</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>The query template used to find temporal events. Experiments are performed using queries with 4 event vertices and 2 feature vertices. One feature vertex is labeled and all other vertices specify only types.</td>
<td>59</td>
</tr>
<tr>
<td>3.8</td>
<td>Results from queries finding four articles with a common keyword and location. Legends indicate degree of query label.</td>
<td>60</td>
</tr>
<tr>
<td>3.9</td>
<td>Comparison with IncIsoMatch. MQD refers to the Multi-Relational Query Decomposition algorithm from this paper.</td>
<td>61</td>
</tr>
<tr>
<td>3.10</td>
<td>Performance results for queries on the DBLP dataset. The spikes in the plot can be attributed to the bursty nature of scientific publishing where authors target the same group of conferences and journals every year.</td>
<td>63</td>
</tr>
<tr>
<td>3.11</td>
<td>Query processing time for the Tencent Weibo dataset for queries with varying selectivity.</td>
<td>65</td>
</tr>
<tr>
<td>3.12</td>
<td>Query processing time for the Tencent Weibo dataset with temporal match pruning applied on every 5 million edges.</td>
<td>66</td>
</tr>
<tr>
<td>4.1</td>
<td></td>
<td>84</td>
</tr>
</tbody>
</table>
4.2 Edge type distribution shown with the evolution of the dynamic graph. X-axis shows the number of cumulative edges in the graph as it is growing. The plotted distribution is not cumulative. The edge distribution is collected after fixed intervals. The interval is 10 thousand, 100 thousand and 1 million respectively. There are 4, 7, and 45 edge types in these datasets.

4.3 2-edge path distribution in each test data set. Each point on X-axis represents a unique 2-edge path and Y-axis shows its corresponding count.


4.5 Example queries used for testing New York Times data.

4.6 1 and 2-edge based decompositions of the network flow query graph shown in Figure 4.5a.

4.7 Query processing times using four different search strategies for New York Times data.

4.8 Degree distribution from New York Times data based on vertex types.

4.9 Analyzing relative performance between the top-3 search strategies for New York Times data.

4.10 Example path queries used for testing CAIDA data.

4.11 1 and 2-edge based decompositions of the network flow query graph shown in Figure 4.10a.

4.12 Runtime collected for diameter 4 queries at different intervals.

4.13 Memory usage collected for diameter 4 queries at different intervals.

4.14 Runtime collected for diameter 5 queries at different intervals.

4.15 Memory usage collected for diameter 4 queries at different intervals.

4.16 Summary of results from Netflow.

4.17 Example path queries used for testing LSBench data.
4.18 1 and 2-edge based decompositions of the network flow query graph shown in Figure 4.17d. ........................................ 109
4.19 Experimental results from LSBench. ................................. 110
5.1 Illustration of a BSP computation..................................... 115
5.2 Progression of a Pregel computation in a 4-vertex graph. Dotted lines indicated messages and shaded vertices are voted to halted state. ....... 118
5.3 The core API for a vertex program................................. 119
5.4 Overview of the GMT architecture. ................................. 127
5.5 Data structures for Distributed Streaming Graph............... 132
5.6 Dataflow inside the SJ-Tree............................................. 136
5.7 Scaling results from New York Times data......................... 144
5.8 Scaling results from CAIDA internet backbone traffic data........ 145
5.9 Scaling results from LSBench data................................... 145
6.1 Example of a frequent subgraph appearing in a social media data stream.. 149
6.2 An example of frequent subgraphs in network traffic data where a group of clients (gray nodes) use the same set of servers (red nodes). ...... 150
6.3 Directed graph showing dependencies between patterns. Solid and dotted rectangles are used to indicate frequent and infrequent subgraphs respectively. ............................................... 155
6.4 Pipeline for incremental pattern discovery.......................... 160
6.5 A sample pattern discovered from New York Times data.......... 166
6.6 Total runtime as a function of varying frequent subgraph support (NYT)........ 167
6.7 A sample pattern discovered from internet backbone traffic data where
an IP address (one on right) is flooded with requests from another IP
address using different source ports. .................................................. 167

6.8 Total runtime as a function of varying frequent subgraph support (net-
work traffic). .......................................................... 168

6.9 Total number of frequent and infrequent subgraph patterns tracked dur-
ding the course of the dynamic graph evolution. ......................... 170

6.10 Total number of frequent pattern instances tracked during the course of
the dynamic graph evolution. ................................................. 171

7.1 Cyberattack graph queries for a) Witty worm, b) Smurf DDoS, c) Fraggle
DDoS, and d) DNS Amplifications DDoS cyberattack. ............... 176

7.2 Example of a streaming query for a social stream with geo-location data. 180

7.3 Logical schema of the LS Bench stream data generator. ............... 181

7.4 Graph representation of SPARQL query 6 from LS Bench benchmark. .. 181

7.5 Graph representation of SPARQL query 7 from LS Bench benchmark. .. 182

7.6 Example of a graph query targeting multiple clouds. The colors on the
edges indicate the cloud database where the relation is stored. ........ 183

7.7 Edge based decomposition for query in Figure 7.6 ....................... 184

7.8 Decomposition using larger subgraphs for query in Figure 7.6 .......... 185
“I have a turn both for observation and for deduction. The theories which I have expressed there, and which appear to you to be so chimerical are really extremely practical - so practical that I depend upon them for my bread and cheese.” Sherlock Holmes, A Study in Scarlet (1887)

Data sources such as social media streams and network traffic can be modeled as multi-relational graphs. Such graphs are defined by a set of relations between a potentially diverse set of entities. For example, a social media data stream contains a diverse set of entity types such as person, movie, and image, and relations such as friendship or liking of a post or image. For cyber-security, a network traffic data source can be represented as a graph with IP address as nodes. The edge types for a network traffic graph are defined by the communication protocols.

All these applications generate a prodigious amount of data in a continuous fashion. Application domains such as computational journalism, emergency response, and national security put a premium on discovering critical events as soon as they emerge in the data stream. From a database perspective, real-time querying of streaming data is not a new problem; it has been exhaustively studied in the context of streaming databases and continuous queries. A continuous query system is a query that logically runs continuously
over time as opposed to being executed intermittently. Once registered into the system, the query is continuously evaluated against incoming data. Driven by real-time application needs, such queries are usually distinct from ad-hoc queries. They typically follow a pattern such as “notify the user when $X$ happens”, where $X$ describes the event of interest. As a simple example, one may consider registering a query into a brokerage system to receive notifications when the stock symbol “FB” reaches $50.00 with a certain number of trades happening in the past 5 days.

The emergence of large-scale graph databases from fields such as social media and cyber data makes it imperative to revisit this familiar field. Similar to the trading example presented above, one can envision the social networking websites introducing more advanced graph search features. Given that users are often overwhelmed from updates from their social network, they may choose to receive notifications once certain patterns are matched in their social stream. Examples of such queries are “tell me when someone from company $X$ is reachable by $k$ degrees of connections” to “tell me when two of my college friends check-in at a restaurant within $k$-miles”.

The stakes are higher in the fields of cyber-security and other national security applications. Detecting adversarial activities, prevention of theft of intellectual properties and customer data is a high priority for corporations and government agencies around the world. Users or analysts from such domains typically have a set of “interesting patterns” that they want to be continuously evaluated on the streaming input. As an example, one
may wish to detect botnets by searching for patterns of denial-of-service attacks in the network traffic. To cite another example, a system administrator will want to know whenever a user’s network activity shows unusual patterns of logins from a set of connected machines. A cyber defender will register these queries on a stream processor and wish to be notified as soon as the pattern is matched.

1.1 Motivation

The natural representation of both the data graph and the patterns motivate us to explore a graph pattern matching based solution to the above mentioned problems. The final outcome of this dissertation is a system for Streaming Graph Database (Figure 1.1). The system manages the dynamic graph as it is changing over time, and performs incremental query processing by leveraging on the stream statistics.

1.1.1 Motivating Goals

Social data streams such as Twitter generate about 500 million tweets everyday. This translates into about 5,700 tweets per second. Each social media post typically results in insertion of multiple edges in the dynamic graph. Considering the fact that a significant fraction of tweets can be filtered away as noise, this level of traffic requires processing
around a billion edges per day. Projecting these numbers to 5 years into the future and considering the exponential growth in data volume, building a system that can process 100 billion edges is an ambitious challenge. Processing 100 billion edges per day nearly equates a throughput of 1.15 million edges per second. Such throughput numbers are representative of the cyber-security domain as well.

How does the state of the art fare with these challenges of scale? The area of dynamic graph research is a nascent field. The field of graph search has grown immensely over the past ten years. However, (non-streaming) graph databases that support data with billions of nodes or tens of billions of edges are starting to emerge only now. The rise in
scale introduces unique challenges as well. For example, many of the successful graph search algorithms rely on indexing techniques that have a $O(N^2)$ complexity, where $N$ is the number of nodes in the graph. At a billion nodes, a $O(N^2)$ solution may not be an acceptable solution. Turning towards streaming, many of the proven graph search heuristics become irrelevant for dynamic graph search [1]. Twitter currently employs Storm, a stream computation system based on the dataflow model. None of the published use cases [2] represent expressive or sufficiently complex graph queries.

1.2 Research Contributions

The research is aimed at four major focus areas.

1. **Incremental Graph Search**: How do we efficiently execute queries on a dynamic graph? Development and implementation of incremental graph search algorithms for streaming data is the primary contribution of this research (chapter 3). Our experiments demonstrate speedups by 10-100 times over existing approaches.

2. **Query Optimization**: The statistical characteristics of the graph stream can have a profound impact on the query processing performance. Developing search strategies that measure and exploit statistics of the graph stream to select an efficient execution strategy is the next important contribution (chapter 4).
3. **Scalable Solutions:** The driver applications generate streaming updates at a prodigious rate, sometimes producing millions of records within a minute. Developing scalable implementations of the above algorithms that leverage parallel or distributed computing techniques is critical to address throughput challenges (chapter 5). We developed a distributed implementation of the dynamic graph search and demonstrated strong scaling on multiple large-scale datasets.

4. **Enabling Graph Mining:** We are often interested in the complementary task of *graph mining*, where we do not know the patterns to search for, or wish to discover interesting patterns from the data. Efficient algorithms for graph search and computing summary statistics from streaming graphs play an important role in steering the graph mining process. A new streaming graph mining algorithm is proposed in the thesis that leverages on the ideas from graph search (chapter 6).
CHAPTER 2. BACKGROUND AND RELATED WORK

“There comes a time when for every addition of knowledge you forget something that you knew before. It is of the highest importance, therefore, not to have useless facts elbowing out the useful ones.” Sherlock Holmes, A Study in Scarlet (1887)

The past decade has been an watershed for the field of graph theory and its applications. It saw the emergence of large-scale graph databases from social networks to biological networks, where graphs serve as a natural representation of the underlying data. Searching a database for patterns is a fundamental task. Querying a graph for patterns, or graph matching as it is generally called, has been studied for more than four decades [3]. The scale of current databases render many of the historical approaches ineffective. On the other end of the spectrum, relational databases solved the problem of querying large databases for decades by focusing on data organization, indexing and query optimization. Not surprisingly, the past decade has seen application of many of these ideas to graph databases. This chapter provides an overview of the representative work on graph search over the past decade. The survey exclusively focuses on works that deal with searching a database containing a large graph and not many small graphs.
2.1 Definitions

The research on graph search explored two primary areas. The first one, subgraph isomorphism, requires finding a subgraph in the query graph such that there is a bijective mapping between the edges of the query graph and a subgraph of the data graph. The other variants are relaxations of the constraints associated with subgraph isomorphism. For example, graph homomorphism relaxes the bijective mapping criteria between vertices of the query graph and a matching subgraph. The second area, subgraph pattern matching, allows mapping of an edge of the query graph to a path in the data graph.

We begin with some basic definitions.

**Definition Graph** We define a graph $G$ as an ordered-pair $G = (V, E)$ where $V$ is the set of vertices and the $E$ is the set of edges that connect the vertices.

An edge represents a pair of vertices, also known as end points. We use $V(G)$ and $E(G)$ to indicate the set of vertices and edges associated with a graph $G$. We use the term query graph to indicate a graph based representation of the pattern being searched for and data graph to indicate the graph that is being searched for a pattern.

**Definition Subgraph** Given a graph $G = (V, E)$, a subgraph denoted as $G' = (V', E')$ is a graph with $V' \subseteq V$, $E' \subseteq E$, such that $E' = \{\{v_i, v_k\} | v_i \in V', v_j \in V'\}$.

**Definition Directed Graph** A graph can be directed or undirected depending on the asymmetric or symmetric relation between vertices. For directed graphs, $E(G)$ is a col-
lection of ordered pairs. An edge whose end-points refer to the same vertex is a self-loop.

A simple graph is one that has no self-loops and does not allow multiple edges between two vertices.

**Definition Labeled Graph** A labeled graph is a six-tuple $G = (V, E, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$, where $\Sigma_V$ and $\Sigma_E$ are sets of distinct labels for vertices and edges. $\lambda_V$ and $\lambda_E$ are vertex and edge labeling functions, i.e., $\lambda_V : V \rightarrow \Sigma_V$ and $\lambda_E : E \rightarrow \Sigma_E$. Unless otherwise specified, we consider simple, undirected graphs with vertex labels.

The terms (sub) graph pattern matching, (sub)graph matching, (sub)graph query or (sub)graph search are often used interchangeably. This document defines subgraph query or subgraph search as pertaining to subgraph isomorphism. Subgraph pattern matching is referred to as subgraph matching in the rest of the document. Both terms are defined below to contrast the differences between the problem statements. The term subgraph query is used in this document in a rather abstract sense to encompass both subgraph search and subgraph pattern matching. It is used to allude to the high-level task of searching a database for a query without caring about processing constraints.

**Definition Subgraph Isomorphism** Given the query graph $Q$ and a subgraph $G'$ of $G$, an isomorphism between $Q$ and $G'$ involves finding a bijective function $f : V(Q) \rightarrow V(G')$ such that for any two vertices $u_1 \in V(Q)$ and $u_2 \in V(Q)$, $(u_1, u_2) \in E(Q) \Rightarrow (f(u_1), f(u_2)) \in E(G')$.

**Definition Subgraph Matching** Given the query graph $Q$ and a subgraph $G'$ of $G$,
a matching between $Q$ and $G'$ involves finding a set of pairs $M = \{(u, v) | u \in V(Q), v \in V(G')\}$ such that for any two pairs $(u_1, v_1) \in M$ and $(u_2, v_2) \in M$, $u_1 \neq u_2$ and $v_1 \neq v_2$.

Observe that subgraph matching is a relaxed version of subgraph isomorphism, without its adjacency requirements. Thus, subgraph matching will allow mapping an edge in $Q$ to a *path* in $G$, where *path* is defined as follows.

**Definition** Path A path between two vertices $u$ and $v$ in graph $G$, denoted by $d(G, u, v)$ is a sequence of non-repeating vertices such that each consecutive vertex pair in that sequence is an edge in $G$.

In view of its relaxed requirements, the number of answers to a subgraph matching query can be large. Therefore, it is useful to constrain the definition of subgraph matching with a distance bound $k$.

**Definition** Constrained Subgraph Matching Given the query graph $Q$ and a subgraph $G'$ of $G$, a matching between $Q$ and $G'$ involves finding a set of pairs $M = \{(u, v) | u \in V(Q), v \in V(G')\}$ such that for any two pairs $(u_1, v_1) \in M$ and $(u_2, v_2) \in M$, $u_1 \neq u_2$ and $v_1 \neq v_2$, and $d(G, v_1, v_2) < k$.

### 2.2 Anatomy of a graph matching algorithm

It is important to appreciate the combinatorial issues that make graph matching a hard problem. We begin with the description of a simple graph matching algorithm. Given the
query graph $G_q$, $\forall u \in V_q$ we generate the candidate set of vertices $C_u$ from the search graph $G_s$ such that $u$ can be matched with any element in $C_u$. Next, define a global candidate set $C$ which is the union of all individual candidate sets, $C = \{C_u|u \in V_q\}$. Observe that a vertex from $G_s$ can be potentially mapped to multiple vertices in $G_q$. However, each such mapping will lead to a different combination or match.

**Figure 2.1:** Illustration of the search process for Subgraph Isomorphism.

The vertex to vertex mapping is subjected to the constraints of subgraph isomorphism. Assume that an edge $e = \{u, v\}$ belongs to the graph $G_q$ and $u'$ and $v'$ are the corresponding vertices from the candidate set $C$. The constraint specifies that since $u$ and $v$ are adjacent in $G_q$, $u'$ and $v'$ should be adjacent in $G_s$. Typically a graph matching algorithm generates the candidate sets and then performs constraint checking to determine the matches. Let us say $N_c$ and $N_q$ are the cardinality of the vertex sets of $C$ and $G_q$. Therefore,
the number of initial candidate sets is $\binom{N_c}{N_q}$, which can be extremely large if $N_c \gg N_q$. This exponential factor is the root cause behind the hardness of subgraph isomorphism. Once the candidate sets are generated the second phase involves performing a search on this candidate space to compute the matches.

Figure 2.1 shows an example of how the search process develops. Assume that the vertex sets $V_q = \{v_0^q, v_1^q, \ldots\}$ and $C$ are ordered. We begin with selecting the first element from both, and add them to a match set if they satisfy all constraints. Then $v_1^q$ is selected and the remaining entries from $C$ are searched to find its matching candidate. Thinking in terms of a search tree, each such choice is equivalent to visiting one more level deeper in the tree. A complete match is produced when we reach a depth $|V_q|$ and have a matching candidate for all vertices in $V_q$. At that point, the search begins again from the root by selecting a different matching candidate for $v_0^q$ and it progresses in a depth-first fashion. Thus, the entire search tree represents the complete search space, or all combinations of vertices from $V_q$ and $C$. All internal nodes represent partial matches. A complete match is produced when we reach a leaf node successfully. During the search process or equivalently at any node in the search tree, if the last selected vertex pair are incompatible then the search space corresponding to the subtree of that internal node can be avoided. Such pruning is critical to the efficiency of the search. The candidate generation and search processes are two logical steps and can be completely overlapped for an efficient implementation. The novelty of any graph matching algorithm lies in how it addresses these two issues.
There is significant work reported in the literature that targets each of these phases with different objectives. These sub-problems, or different query processing components, are summarized below.

- **Indexing** Many of the operations involved in subgraph querying involve label equality check as a necessary step. However, generating matching candidates based on simple attribute checks is naive. Techniques for structural and semantic summarization are reviewed from the papers by Zhao [4] and Lian et al. [5].

- **Query Planning** Given a query graph $Q$, is it a good idea to move straight to the candidate generation and join/search? *Minimization* refers to the process of generating an equivalent graph that may be more amenable to optimization. Next, given a set of candidate vertex sets, what is the best way of performing joins across them? The query planning step is used to answer this question and the contributions from [6] and [5] are presented.

- **Matching/Search** What are the best implementation strategies for performing the state-space search? What are the different query classes? What are the time complexities for these different query classes? Contributions from [4, 7, 8] are summarized here.
2.3 Indexing

An overwhelming number of papers focus on indexing and different query algorithms that exploit various types of index structures. Works such as fg-Index [9] and GADDI [10] leverage on frequent graph pattern mining techniques to index frequent and discriminative substructures. We review GADDI in greater detail below. Indexing a database with many small graphs attracted a significant amount of work [11–13]. TALE and SAPPER [14, 15] focus on designing a hybrid indexing mechanism for graphs that are too large to fit into main memory. A common theme of such approaches is to index vertices based on their labels, degree and some signature of the neighborhood structure of every vertex.

TALE uses the concept of centrality and a graph matching algorithm to produce a set of candidate vertices that are used to initialize a combinatorial search. Similar matching based approaches are pursued by [16] and [17] who respectively use the spectral properties of graphs and center-piece subgraphs to drive the matching algorithm. Finally, there are applications where finding a collection of matching vertices with a minimal structural semantics is sufficient.

NESS [18] belongs to this category as it applies an information propagation approach to transform a network into a multi-dimensional vector space to take advantage of efficient similarity search techniques. Section 2.3.4 summarizes the work by Zhao and Han [4], which seems squarely applicable for both subgraph matching and search.
2.3.1 Indexing via Discriminative Substructures

GADDI [10] approaches the above problems by characterizing the neighborhood of every vertex by exploring vertices $k$ hops away. $k$ is typically set to 2. It preprocesses both the search and query graph to compute a neighborhood signature based index for every vertex. The neighborhood signature is based on frequent and discriminative substructures in the graph. The signatures coupled with the label of a vertex provides strong pruning capability during the candidate generation phase. The information from the candidate generation phase is leveraged during the search phase as well. The fundamental concepts behind the algorithm are defined first.

**Shortest Distance** Given vertices $u$ and $v$ in a connected graph $G$ the shortest distance is denoted as $d(G, u, v)$ and defined as the number of edges on the shortest path connecting $u$ and $v$.

**k-Neighborhood** Given a vertex $v$ in graph $G$ and an integer $k$, the $k$-Neighborhood of $v$ denoted as $N_k(G, v)$, is defined as the set of vertices in $G$ such that $\forall v' \in N_K(G, v), d(G, v, v') \leq k$.

**Induced Subgraph** Given a subset $V'$ of vertices of vertex set $V$ of graph $G$, an induced subgraph of $V'$ or $I_{sub}(G, V')$ is the subgraph $G' = (V', E')$ such that $E' = \{(u, v) \in E(G) | u \in V' \text{ and } v \in V'\}$. 
Neighborhood based distance

Given a vertex $v_q$ from $G_q$, it can be matched to a vertex $v_s$ from the search graph only if it has the same label. However, such a criteria will produce numerous candidates for every vertex. A more restrictive approach is to define newer properties for vertices in the query graph and require that these properties are satisfied in addition to the label equality criteria. The same intuition applies to the adjacency requirement between neighboring vertices which states that if $u$ and $v$ are adjacent in $G_q$ then their counterparts $u'$ and $v'$ should be adjacent as well. These ideas can be formalized as a function for a vertex pair from a graph, denoted as $f(G, u, v)$ that characterizes the local neighborhood of the vertices in the graph. Thus, $f(G_q, u, v) \Rightarrow f(G_s, u', v')$.

GADDI implements this neighborhood characterization by neighborhood discriminating substructure distance (NDS). Given a vertex pair $u$ and $v$ in $G$, this requires extracting a local subgraph around each vertex. Observe that the local subgraph is the result of the k-neighborhood function. Next, the common subgraph of the k-neighborhoods, $I_{sub}(G, N_k(G, u) \cap N_k(G, v))$ is extracted. Assume that we have a set of discriminative substructures $DS(G)$ which are the frequent subgraphs of $G$. If $P$ is a substructure present in $DS(G)$, then $d_{NDS}(G, u, v, P)$ is the number of occurrences of $P$ in $I_{sub}(G, N_k(G, u) \cap N_k(G, v))$. Finally, the NDS distance between $u$ and $v$ is the total occurrences of all discriminative substructures that appear in the common subgraph, i.e.,
\[ \forall P_k \in DS(G), d_{NDS}(G, u, v) = \sum_{P_k} d_{NDS}(G, u, v, P_k). \]

**Computation of discriminative substructures**

A *discriminating substructure* is defined in the context of the induced subgraph that results from the intersection of the k-neighborhood of two vertices. The authors consider a subgraph as a discriminating subgraph if it is frequent in the intersecting subgraph by occurring in more than 50% of the cases. The gSpan [19] algorithm is used for discovering the top-10 frequent subgraphs. Also, the frequent subgraph mining is performed using purely structural features and no labels. Next, only the most discriminative subgraphs are chosen that lead to large inter-class distance and small intra-class variance. Finally, only a small subset of the originally selected top-10 frequent patterns are selected for insertion in \( DS(G) \). The results presented in their paper are based on top-3 patterns.

**Index construction**

Once the set of discriminative substructures is learned, the next step is to compute the NDS distance between every vertex pair in the graph. If the k-neighborhood computation explores vertices at most \( L \) hops away and if \( \bar{d} \) is the average vertex degree, then there are approximately \( N_p = V \bar{d}^L / 2 \) vertex pairs in the graph. It is obvious that the multiple vertices will have significant overlap between their k-neighborhood sets and thus, the computation for \( N_p \) pairs can be performed efficiently by reusing partial results from subproblems. The index structure resembles an adjacency list representation of a graph.
The data structure is represented by a collection of arrays. A graph with \( N \) vertices has \( N \) arrays, each for one vertex. Each array stores the distance from a vertex to all the vertices in its \( k \)-neighborhood.

**Depth first matching**

This phase implements both the candidate generation and state-space search as described earlier. The candidate generation process uses the NDS distance to prune candidates. Assume \( v_q \) and \( v_s \) are two vertices from the query and search graph. If \( v_q \) is to be matched with \( v_s \), then for each \( v \) that is in a \( k \)-neighborhood of \( v_q \), we need to identify a \( v' \) in the search graph such that 1) \( \text{label}_q(v) = \text{label}_s(v') \), 2) \( d(G_q, v_q, v) \geq d(G_s, v_s, v') \), and 3) \( d_{NDS}(G_q, v_q, v) \leq d_{NDS}(G_s, v_s, v') \). For each vertex in the query graph a set of vertices from the search graph are retrieved satisfying these constraints. The depth-first-search is performed as described in section 2.2.

Constructing an index of discriminative substructures is not feasible in a dynamic graph setting. However, our approach toward graph search is inspired by searching for discriminative substructures. As we discuss in later chapter, we rank all single and 2-edge subgraphs by their frequency count and employ a search strategy that begins with searching for the most discriminative substructure.
2.3.2 **TALE: Hybrid index with centrality based matching**

TALE [14] implements an approximate subgraph matching algorithm. The algorithm allows the user to quantitatively specify the amount of mismatch that can be tolerated between the query graph and the matches found. The novelty of their implementation lies in two distinct levels. The first level implements a hybrid index structure. A B+Tree structure is used to index the degree, label and neighborhood connectivity of the vertices. The number of edges between the neighbors of a vertex is defined as neighborhood connectivity. Each leaf of the B+ tree stores points to a bitmap index which contains the vertex membership information for that unique index value. Given a query graph, the algorithm uses this hybrid index structure to retrieve matching search graph vertices. The maximum number of mismatched vertices are specified by a error tolerance parameter. The second level begins with the set of query graph vertices and the corresponding set of search graph vertices. Next, it ranks the query graph vertices in terms of their prominence in the query graph. The authors use the centrality score as a measure of prominence. Once ranked, the top-k vertices in the query graph are used as seeds or initialization candidates for the combinatorial search. A match is initiated with the top-k vertices in the query graph and their corresponding search graph counterparts. Given that each vertex may have multiple matching candidates, a maximum weighted bipartite matching algorithm is used to generate the initial match. The match is expanded by considering each matched vertex pair...
in isolation and trying to grow the match by recursively by matching their immediate and 1-hop distance neighbors.

TALE is not immediately relevant for our work. However, its use of centrality as a measure for ranking results is novel. We plan to investigate such approaches for developing approximate dynamic graph search algorithms in future.

2.3.3 **SAPPER: Space efficient indexing and edit distance based matching**

SAPPER [15] is another algorithm that incorporates a combination of indexing and approximation graph matching techniques. SAPPER contains four main steps - vertex matching, constructing random spanning trees of the query graph, generating a matching order of graphs, and final graph matching. For each vertex $v_q$ in the query graph, the vertex matching step consists of finding matching candidates $v_s$ that satisfy the following conditions: 1) the labels of $v_q$ and $v_s$ should be same, 2) the degree of $v_s$ should be greater than or equal to the degree of $v_q$, 3) the labels of vertices in $N_1(v_q, G_q)$ are a subset of those in $N_1(v_s, G_s)$, and 4) the labels of vertices in $N_2(v_q, G_q)$ are subset of those in $N_2(v_s, G_s)$.

Bloom filter, a space efficient probabilistic data structure, is used to retrieve the matching candidates for each query graph vertex. SAPPER uses the graph edit distance as a measure of approximate matching. If the threshold for this approximation error or maximum allowed edit distance is small, then all candidate matches will be heavily overlapped. A
set of random spanning trees of the query graph are generated and the following phases are
dedicated to finding matches for each of the generated random spanning trees. For each
spanning tree generated, a tree match algorithm is executed to generate sets of matching
candidate vertices from the search graph. The key insight here is that the matching of the
spanning tree does not solve the actual graph matching problem; however, it generates can-
didates for matching with a subset of edges in the query graph and exploits the efficient
properties of tree matching. Finally, the edges of the query graph are enumerated in a lexi-
cographic order and are matched with candidates from the search graph using a depth-first
search. Experimental results presented show that SAPPER clearly outperforms GADDI.

As an algorithm, SAPPER is not relevant for dynamic graph search. Conceptu-
ally, our query decomposition approach for dynamic graphs is similar to the spanning tree
based idea used by SAPPER. Its use of edit distance to detect closeness between candi-
date matches is an interesting approach, although computation of graph edit distance is
an expensive operation. As we discuss in the subsequent chapters, generation of a large
number of overlapping candidates (through high-degree vertices) is often the performance
bottleneck for search and mining techniques. Using alternate techniques such as locality
sensitive hashing [20] in place of graph edit distance, and pruning match candidates by
detecting overlaps can provide great speedups for both dynamic graph search and mining.
2.3.4 SPath: Label propagation based Neighborhood Signature Computation

SPath [4] extends the concept of neighborhood based graph indexing in a different direction. The above algorithms are based on a vertex based matching approach. They begin with a set of candidate vertices and explore various combinations of such matchings. Vertices are the smallest matching unit which is selected using label equality and often, a comparison of their neighborhood signature. SPath elevates this smallest matching unit to a path. SPath indexes each vertex in the graph using a k-neighborhood as in GADDI. However, their definition of neighborhood signature is different.

Given a graph \( G = (V, E, \Sigma_V, \Sigma_E, L) \), a vertex \( u \in V(G) \), a non-negative distance \( k \) and a vertex label \( l \in \Sigma_V \), the authors define a function \( S^l_k(u) \) that is the set of vertices \( k \) hops away from \( u \) and have the vertex label \( l \). Thus \( S^l_k(u) = \{ v \mid d(u, v) = k, L(v) = l, v \in V(G) \} \). The \( k \)-distance set of \( u \) is defined as \( S_k(u) = \{ S^l_k(u) \mid l \in \Sigma_V \} \). Finally, the neighborhood signature of \( u \) is defined as \( NS(u) = \{ S_k(u) \mid k \leq K \} \) where \( K \) is a user specified threshold. Therefore, given a search graph \( G_s \) and a query graph \( G_q \), finding a vertex-to-vertex matching \( f \) requires that \( \forall v \in V(G_q), NS(v) \subseteq NS(f(v)) \), where \( f(v) \in V(G_s) \).

Thus, each vertex is associated with a number of paths of varying length, and the set of labels on each path is stored. Therefore, given a query graph \( G_q \), each vertex \( v \in V(G_q) \) can be associated with a vertex \( v' \) in \( G_s \). Now, each such \( v' \) is associated with multiple
paths. If $P$ denotes the set of all such paths, then a subset of $P$ can be selected to produce a subgraph of $G_s$ that is isomorphic to $G_q$. This reduces to the set cover problem, and the authors use a greedy $\log(n)$-approximation algorithm to compute an optimal combination of paths that leads to a complete match.

Again, our approach is conceptually similar to SPath as both approach the graph search problem by combining results for smaller subgraph queries. However, $k$-neighborhood based indexing is not feasible in the dynamic setting, and it causes the final search strategies to be entirely different. SPath accumulates small paths obtained from vertex index lookups and repeatedly determines the optimal combination order, while we accumulate small partial matches that are collected in a pre-determined order using graph stream statistics.

### 2.3.5 Bit-Vector based Indexing

An alternate approach to the above data structures is to encode the label information from each node’s $k_0$-hop neighborhood as a collection of bit vectors [5]. For each vertex $v$ and for each $k \leq k_0$, a bit vector $BV_{\leq k}(v)$ will be maintained. Assuming all bit vectors have uniform size $B$ and a hash function $H(\cdot)$, each label will be mapped to a position in $BV_{\leq k}(v)$ and the $H(l(v))$-th bit will be set to 1. The bit-vectors are attractive for their space optimal characteristics, and the label information for both vertices and edges in a $k$-neighborhood can be encoded in this fashion. Such an implementation is attractive for the
compact representation and the fast bit-based computations. However, the authors recognize that for graphs with a skewed label distribution such bit-vectors will lose their pruning power. This is because multiple labels can be hashed to the same position in a bit vector. If both a frequent and infrequent label are hashed to the same index, the infrequent label will no longer be effective for pruning.

The best of both worlds may be obtained by using adaptive hash functions which are cognizant of label frequencies. Assume that $w(l_i)$ is the frequency of a label $l_i$, denote $W = \sum_{i=1}^{L} w(l_i)$. Then, for each label a random number $r_i$ is generated within the range $[0, w(l_i)]$. Thus, an adaptive hash function maps a label $l_i$ as follows,

$$H_{ada}(l_i) = \lfloor B \cdot \frac{\sum_{j=1}^{i-1} w(l_j) + r_i}{W} \rfloor$$

The goal of $H_{ada}(l_i)$ is to map label $l_i$ to a random position in the range $\left[ B \cdot \frac{\sum_{j=1}^{i-1} w(l_j)}{W}, B \cdot \frac{\sum_{j=1}^{i} w(l_j)}{W} \right]$ with a probability proportional to $1/w(l_i)$. This lowers the chance of collision between a frequent and infrequent label. More details on designing adaptive hash functions are available in [5].

Bit-vector based indexing is not feasible for dynamic graph setting as periodic recomputation of the index can be expensive. However, as discussed earlier at the end of section 2.3.3, locality-sensitive hashing algorithm is a promising area for future work; hence, using adaptive hash functions that address skews with label frequency will be extremely
2.4 Query Pre-Processing

2.4.1 Query Rewriting

Query rewriting is a popular term in databases where a query is transformed into its equivalent form that is more amenable to optimization. This problem can be approached via Graph Minimization in the context of subgraph pattern matching [21]. Graph minimization is defined as follows: given a query graph $Q$, find another query graph $Q_m$ such that (1) $Q \equiv Q_m$, (2) $|Q_m| \leq |Q|$ (i.e. the number of vertices in $Q$) and (3) $\not\exists Q'_m$ such that $|Q'_m| < |Q_m|$.

Note that $Q_1 \equiv Q_2$ indicates that $Q_1$ and $Q_2$ are equivalent. This is defined via the properties of containment and equivalence. Without getting into mathematical details, a graph $G_1$ is said to be contained in $G_2$, denoted as $G_1 \subseteq G_2$ if there is a renaming function that maps all edges of $G_1$ to $G_2$. Two graphs $G_1$ and $G_2$ are equivalent if $G_1 \subseteq G_2$ and $G_2 \subseteq G_1$. Fan et al. [21] show that given a query graph $Q$, it is possible to compute a minimum equivalent graph $Q_m$ in cubic time.

At the core of the minimization process is finding similarity between different vertices within a graph [22], transforming that information into an equivalent relation with no redundancy, creating a new graph structure with an appropriate number of copies of relevant for such an approach.
each equivalence substructure class, and finally, removing redundant edges from the original graph. Typically, the query graphs are small and if they were to run many times in a production environment, transforming such graphs using a cubic time algorithm will be extremely productive.

This idea is based on the intuition that a smaller query graph is better from a search perspective. There is no ample data-driven proof in the literature on its efficacy and remains a strong candidate for validation.

2.4.2 Query Plan Generation

As indicated earlier, the order in which various candidate vertex sets are combined is critical to the performance of the matching or search process. This section is dedicated to the computation of an optimal join-order.

Minimum Spanning Tree

Assume that two vertices $q_i$ and $q_j$ are selected from $V(Q)$. Next, we select two corresponding vertices from the data graph $v_i$ and $v_j$. Because we have a set of candidate vertices for every $q \in V(Q)$, given this matching $\{q_i, v_i\}$ and $\{q_j, v_j\}$ there will be many combinations to choose from. Understand that all these combinations are subject to satisfying the distance based constraint between $v_i$ and $v_j$, i.e., their shortest path has to be within a few hops. Given that $q_i$ and $q_j$ are any two pairs from the query graph $Q$, the
shortest path distance constraint will depend on the distance between $q_i$ and $q_j$. The further $q_i$ and $q_j$ are, more are the choices for $v_i$ and $v_j$. The greater the distance between $v_i$ and $v_j$, larger is the candidate space for matching the rest of the vertices \( \{V(Q) \setminus \{q_i, q_j\}\} \). Lian and Chen [5] solve this problem using a minimum spanning tree approach. They build a completely connected graph $K$ where each vertex corresponds to a vertex in $Q$ and the edge weights between two vertices is the shortest path. To obtain a minimum spanning tree $T$ from $K$, they start with an edge with the smallest distance weight and keep adding to $T$ by examining edges in non-descending order of weights such that it does not introduce a loop in $T$. They stop when all edges in $K$ have been examined. The order in which edges are added in $T$ is the subsequent matching order.

**Vector Space Approach**

While the above sounds conceptually simple, an implicit operation in the above algorithm is the shortest path computation between vertices. The literature that focuses on this query planning step is rather limited, and it is important to acknowledge that the distance computation can be expensive in a large graph. Zou et al. [6] present an extensive study of this topic. They recommend using a vector-space based distance computation approach in contrast to using Dijkstra’s algorithm as in [5]. [6] uses a graph embedding technique to map all vertices in $G$ to a vector space $\mathbb{R}^k$, where $k$ is the dimensionality of $\mathbb{R}^k$. They argue that the $L_\infty$ distance computation between two vertices $u_1$ and $u_2$ in $\mathbb{R}^k$ is the lower bound
of the shortest path $d_{sp}(u_1, u_2)$ and it is much cheaper to compute. While there is always a trade-off between accuracy and performance, the work in [6] is worth consideration for implementing a fast query planning algorithm.

**Greedy Vertex Cover**

This particular case is targeted at subgraph search where such distance computations will not arise. The SPath index (section 2.3.4) is assumed as the index structure supporting the graph query. Below is a description of the planning that happens following the index lookup and prior to beginning a search through the graph. It is worth reminding the reader that each vertex in $V(G)$ is associated with a number of paths of varying length, and the set of labels on each path is stored. Given a graph query $Q$ and $\forall u \in V(Q)$, this step begins with retrieving the paths corresponding to each $u$. Assume that each vertex $v \in V(Q)$ is associated with a vertex $v'$ in $G$ at some point in the search process. Now, each such $v'$ is associated with multiple paths. If $P$ denotes the set of all such paths, then a subset of $P$ can be selected to produce a subgraph of $G$ that is isomorphic to $G_q$. This reduces to the set cover problem and the authors use a greedy log($n$)-approximation algorithm to compute an optimal combination of paths that leads to a complete match.

Our approach to join-order or partial match combination order determination belongs to the same class of divide-and-conquer strategies that the minimum spanning tree or greedy vertex cover belong to. The differences arise in how the smaller matches are com-
bined into larger ones. The main difference between the static and dynamic graph search
is that under the dynamic setting, matches for different parts of the query graph arrive at
different points of time. Therefore, 1) the combination of matches, or match joins are
data-driven and 2) such joins need to be robust to out of order arrival.

On a different note, our current work is restricted to using a greedy-heuristic for join
order determination. This is motivated by our extensive survey of the literature on optimal
join order determination in relational databases [23–25]. A key conclusion of the survey
states that left-deep join plans (ones we adopt) is one of the best performing heuristics.
The above mentioned studies point to a large body of research using techniques such as
dynamic programming and genetic algorithms to find the optimal join order. Nonetheless,
finding the lowest cost join order or using a cost-driven join order determination remains
an interesting problem in graph databases, and the approaches based on minimum spanning
trees or approximate vertex cover can provide an initial path forward.

2.5 Graph Matching and Search

One of the earliest well-known subgraph isomorphism algorithms is proposed by
Ullman [26] which is based on a state-space search approach. Other notable algorithms are
the graph edit distance based approaches proposed by Messmer and Bunke [27]. As with
strings, given two graphs \( g_1 \) and \( g_2 \) a graph edit distance function returns the minimum
number of operations (insertion and deletion of vertices and edges) required to transform $g_1$ to $g_2$. If $g_1$ and $g_2$ are isomorphic, then their graph edit distance will be zero. However, graph edit distance computation [28, 29] has a very high computational complexity and it does not scale beyond graphs with tens of vertices.

VF2 [30] is a more recently proposed algorithm for subgraph isomorphism. It is better than the former counterparts because of the look-ahead and filtering-and-verification approaches it proposes. As an example, assume that it finds a query graph vertex $u$ that has one neighbor $v$ during the search process. Further assume that $v$ has another distinct neighbor $w$ with 10 neighbors. Under such circumstances, the VF2 algorithm will perform degree based checks two hops away before assigning any matches to $u$, which provides greater pruning ability. Other historical approaches that relied on indexing substructures within the search database and the frequent substructure mining tools do not scale for large networks [31]. However, for large networks where vertices often have a power-law degree distribution these algorithms are still vulnerable to combinatorial explosion. Most of the data from domains such as social and biological networks come with rich semantic information captured as labels on vertices and edges. Therefore, more effective approaches are ones that exploit the semantic features of the network (local or micro-level characteristics of network vertices as label distributions) and macro-level features (different approaches depending on the sparsity of the graph).

A bulk of the recent work on graph matching or search is focused on the efficient
candidate generation using the indexing techniques described earlier. There is tremendous diversity in this space. Some of the prominent design patterns include: (1) exploitation of domain specific characteristics for efficient indexing and retrieval ([32–34]), (2) exploiting domain specific characteristics to transform the graph search problem to string search techniques such as suffix-trees ([35, 36]), (3) other variants of searches such as similarity search or top-k queries [37–39], and (4) specific query classes as reachability queries [40], or queries on uncertain data [5, 41].

2.5.1 Spectral graph theoretic approach

A very promising area is the application of spectral graph theory to graph queries [16]. The authors propose a novel approach that computes a similarity matrix between the query graph and the data graph. This similarity measure accounts for both global and local properties of the graph. Given two graph $G_1$ and $G_2$, the similarity matrix $S$ is a $|V(G_1)| \times |V(G_2)|$ matrix where

$$S[u, v] = S_g[u, v] \times S_l[u, v].$$

$S_g$ and $S_l$ represent the global and local similarity matrices. The global similarity matrix compares the spectral properties of two graphs by computing the Laplacian matrix as follows, $L = D - A$ where $D$ is the diagonal matrix with $D_{i,i} = \text{degree}(v_i)$ and $A$ is the adjacency matrix. The next step is to compute the eigen-decomposition of the Lapla-
cian matrix $L = U\Sigma U^T$. When two graphs are of different size, only the eigenvalues of size $\min(|U_1|, |U_2|)$ are considered. The local similarity matrix is computed by i) approximating its Laplacian matrix and ii) then applying maximum bipartite graph matching to retrieve best matching candidates.

Computing the spectra of large matrices is a hard problem, and there is a large body of work in linear algebra dedicated to this problem. The spectral approach is rather orthogonal to our work. The goal here is to simply point the reader to this alternate approach which may be suitable for applications that do not require exact matching between query and data graphs.

2.5.2 Graph Homomorphism

Fan et al. [7] introduced the concept of P-homomorphism and 1-1 P-Homomorphism, which extends the concepts of graph homomorphism and subgraph isomorphism respectively. These notions are defined below.

**Definition P-Homomorphism** Graph $G_1$ is said to be p-homomorphic to graph $G_2$ with respect to a vertex-vertex similarity matrix $M$ and a similarity threshold $\zeta$, denoted by $G_1 \preceq G_2$, if there exists a mapping $\sigma$ from $V(G_1)$ to $V(G_2)$ such that for each vertex $v \in V(G_1)$, (1) if $\sigma(u) = v$, then $M(u, v) > \zeta$ and (2) for each edge $(v, v') \in E(G_1)$, there exists a nonempty path $u/..u'$ in $G_2$ such that $\sigma(u') = v'$. 
**Definition 1-1 P-Homomorphism** A graph \( G_1 \) is 1-1 p-homomorphic to \( G_2 \), denoted by \( G_1 \preceq^{1-1} G_2 \), if there exists a 1-1 (injective) p-homomorphic mapping \( \sigma : G_1 \rightarrow G_2 \).

Now, assume that it is not possible to map all vertices in \( G_1 \) to \( G_2 \). Let us say there is a p-homomorphic mapping \( \sigma : G'_1 \rightarrow G_2 \) where \( G'_1 \subseteq G_1 \). This motivates developing two metrics for measuring the quality of the mapping. **Maximum Cardinality**: Given a p-homomorphic mapping \( \sigma : G_1 \rightarrow G_2 \), this is obtained as the ratio of the number of vertices mapped in \( V(G_1) \) and \( |V(G_1)| \). The second concept is called **Overall Similarity**. If each vertex \( v \) in \( V(G_1) \) has a weight \( w(v) \) associated with it, then the metric is defined as
\[
\frac{\sum_{v \in V(G'_1)} w(v) M(v, \sigma(v))}{\sum_{v \in V(G_1)} w(v)}.
\]

The authors show that the problem of obtaining a p-homomorphic mapping that maximizes either the maximum cardinality or the maximum overall similarity is \( \text{NP} \)-complete. Additionally, they show that there is no polynomial time algorithm for finding 1-1 p-homomorphic mappings such that the quality of each mapping is guaranteed to be within \( O(1/n^{1-\epsilon}) \) of its optimal counterpart, where \( n \) is the number of vertices. They also show that it is possible to build a factor graph of size \( |V(G_1)||V(G_2)| \) and leverage on an approximate algorithm for computing maximum independent sets to develop an approximate algorithm for the max-cardinality and max-similarity problem. For a graph with \( n \) vertices and \( m \) edges, the approximate maximum independent set computation takes \( O(nm) \) and \( O(nm \log n) \) time. With \( n = |V(G_1)||V(G_2)| \) and \( m = n^2 \) in the worst case, an ap-
proximate max-cardinality has a time-complexity of \(O(|V(G_1)|^3|V(G_2)|^3)\). Similarly, an approximate solution to the max-similiarity problem can be obtained in \(O(|V(G_1)|^3|V(G_2)|^3 \log(|V(G_1)||V(G_2)|))\) time.

This thesis is focused on exact subgraph isomorphism. It is not possible to compare these approaches with our solution, primarily due to the differences in the setting in which they are applied. As observed earlier in this chapter, subgraph matching or homomorphism relaxes the requirements imposed by subgraph isomorphism. However, for datasets with a lot of noise the requirements for subgraph isomorphism can be overly restrictive, and subgraph matching or homomorphism promises to be a pragmatic choice.

### 2.5.3 Graph Simulation

In another work, Fan et al. [21] explore the problem of finding patterns defined via graph simulation. The problem of graph simulation is defined as follows.

**DEFINITION Graph Simulation** A graph \(G\) matches a pattern query \(Q\) via graph simulation if there exists a binary relation \(S \subseteq V(Q) \times V(G)\), such that (1) \(\forall\{u, v\} \in S\), \(label(u) = label(v)\), and (2) \(\forall u \in V(Q) \exists v \in V(G)\) such that (a) \((u, v) \in S\) and (b) \(\forall\{u, u'\} \in E(Q) \exists \{v, v'\} \in E(G)\) and \((u', v') \in S\).

Given a set of candidates for each vertex \(u\) in \(V(Q)\), their approach performs the following operations. (1) If \(Q\) is a directed acyclic graph (DAG), then it processes the
DAG in reversed topological order (bottom-up). For any vertex in the DAG, the algorithm combines the matches from its children, subject to the constraints imposed by the corresponding query edges. (2) If the query graph is not a DAG, a DAG representation is built by computing the strongly connected components (SCC) of the query graph. Therefore, each vertex in the produced DAG is a SCC of the query graph Q. Next, the algorithm follows the same process as in (1) to perform a bottom up aggregation of smaller matches until a match with size $|Q|$ is reached.

2.5.4 Strong Simulation

In a sequel to the previous work on graph simulation, Ma et al. [8] introduce a restrictive notion of graph simulation called Strong Simulation. It introduces the concept of Dual Simulation, which is the same as graph simulation with the additional requirement of preserving parent/child relationships between the mapped vertices from Q to G. If a vertex $u$ in Q matches a vertex $v$ in G, then each child (parent) of $u$ must match a child (parent) of $v$. It also enforces the constraints of locality, which suggests that the shortest distance between any two matching vertices from $V(G)$ should not be larger than the diameter of the query graph Q. Intuitively, the first requirement enforces the preservation of structural features in a match, and the second requirement ensures that the vertices in the matching subgraph are not too dispersed to be meaningless. The authors also claim that the number
of matches is bounded by the number of vertices in $G$. This is an attractive feature when contrasted with subgraph isomorphism, which can yield an exponential number of matches.

### 2.6 Continuous Queries

A continuous query can be described as computing a function $f$ over a stream $S$ continuously over time and notifying the user whenever the output of $f$ satisfies a user-defined constraint [42]. They are distinguished from ad-hoc query processing by their high selectivity (looking for unique events) and need to detect newer updates of interest as opposed to retrieving lots of past information. In this paradigm the primary objective is to notify a listener as soon as the query is matched. One may view conventional databases as passive repositories with large collections of data that work in a request-response model whereas continuous queries are data-driven or trigger oriented. These features coupled with real-time demands challenge many of the fundamental assumptions for conventional databases and establish continuous query processing on relational data streams as a major research area. The literature on database research from the past the two decades is abundant with work on continuous query systems [43, 44]. Babcock et al [45] provide an excellent overview of continuous query systems and their design challenges. A common theme across various solutions is the implementation of window-based operators that expose a data stream as a buffer or queue to apply traditional relational operators such as
selection, projection and joins [46]. A large body of work was dedicated to extending SQL and incorporation of continuous query semantics by adding various constraints [47].

In the remainder of this subsection, we begin with some basic definitions to establish the connection between continuous queries and dynamic graph queries and discuss our implementation of key continuous query features.

A continuous query system is defined in terms of streams and the functions that execute upon them.

**Definition Stream** A sequence $S$ is a list of elements $⟨s, τ⟩$ where $s$ is a tuple belonging to the schema of $S$ and $τ$ is the timestamp that belongs to time domain $T$. A stream is defined as an unbounded sequence.

Given this definition of stream one can define various functions $f(S)$ that accepts a stream $S$ as input and produces a sequence of tuples as an output. Further, one may classify them as non-blocking or blocking.

**Definition Non-Blocking Function** Assume that $f^k(S)$ is the cumulative output of the function $f(S)$ after processing a sequence upto and including step $k$. Next, assume that the function accepts a sequence of length $k$, i.e. $S = S^k$, and the corresponding function output is $f(S^k)$. A non-blocking function is one for which $∀k < |S|, f(S^k) = f^k(S)$. Thus when time advances from $τ_{k-1}$ to $τ_k$, the function or query produces a new output corresponding to $τ_k$. 

2.7 **Dynamic Graph Search**

We focus on applications where the input sequence is a list of edges in a multi-relational graph annotated with timestamps. Therefore, our objective is to implement a non-blocking function that operates over the edge sequence and reports whenever an incoming edge to $G_d$ produces a matching subgraph for $G_q$. The primary challenge in designing such a function is with managing the collection of partial matches. The memory requirement for maintaining these partial matches may be unbounded. We address this issue via the join-tree approach discussed in the next chapter. The join-tree exploits the characteristics of multi-relational queries to reduce the number of join outputs and implements temporal property based optimizations to further reduce the number of tracked matches.

Investigation of subgraph isomorphism for dynamic graphs did not receive much attention until recently. It introduces new algorithmic challenges because we can-not afford to index a dynamic graph frequently enough for applications with real-time constraints. In fact this is a problem with searches on large static graphs as well [48]. There are two alternatives in that direction. We can search for a pattern repeatedly or we can adopt an incremental approach. The work by Fan et al. [1] presents incremental algorithms for graph pattern matching. However, their solution to subgraph isomorphism is based on the repeated search strategy. Chen et al. [49] proposed a feature structure called the node-neighbor tree to search multiple graph streams using a vector space approach. They relax
the exact match requirement and require significant pre-processing on the graph stream. Our work is distinguished by its focus on temporal queries and handling of partial matches as they are tracked over time using a novel data structure. From a data-organization perspective, the SJ-Tree approach has similarities with the Closure-Tree [13]. However, the closure-tree approach assumes a database of independent graphs and the underlying data is not dynamic. There are strong parallels between our algorithm and the very recent work by Sun et al. [48], where they implement a query-decomposition based algorithm for searching a large static graph in a distributed environment. Here our work is distinguished by the focus on continuous queries that involves maintenance of partial matches as driven by the query decomposition structure, and optimizations for real-time query processing.
CHAPTER 3. DATA STRUCTURES AND ALGORITHMS FOR DYNAMIC GRAPH SEARCH

“Before turning to those moral and mental aspects of the matter which present the greatest difficulties, let the inquirer begin by mastering more elementary problems”. Sherlock Holmes, A Study in Scarlet (1887)

Graph search involves finding exact or approximate matches for a query subgraph in a larger graph. It has been studied extensively and is formally defined as the problem of subgraph isomorphism: given a pattern or query graph (henceforth described as query graph) $G_q$ and a larger input graph (henceforth described as the data graph) $G_d$, find all isomorphisms of $G_q$ in $G_d$. Following the definition of isomorphism, the matching involves finding a one-to-one correspondence between the vertices of a subgraph of $G_d$ and vertices of $G_q$ such that all vertex adjacencies are preserved. Now consider the challenges in applying traditional graph search techniques to this problem. Unless carefully adapted, a standard search function will search the entire data graph repeatedly and retrieve the same search results. Also, many of the best performing graph search algorithms rely on indexing the graph. Even with an interval as large as 5 minutes, rebuilding the index of a massive graph repeatedly is infeasible. This motivates exploration of incremental algorithms for continuous queries, although the general problem of incremental subgraph isomorphism is
proven to be NP-complete as well [1].

Figure 3.1: A graph query for monitoring emergencies in social media and news streams.

Queries like the one shown in Figure 3.1 share a number of common attributes. First, they all involve an implicit time window to suggest the timeliness aspect associated with the query. Clearly, the length of the time window varies depending on the application context. The average monitoring time window for a high volume social media stream may be in tens of minutes whereas the equivalent period for online news may be in hours or days. Second, all these queries aim to discover a number of temporal events that share the same context, such as a common set of keywords and location. Lastly, a multi-relational graph often takes the form of k-partite graphs [50, 51] where each partite set represents a group of entities of the same type. For queries like the ones shown in Figure 3.1, each event that is represented by an article or a tweet can be viewed as a k-partite subgraph.

We exploit these three features to implement a continuous query processing framework for multi-relational graphs. First, by utilizing a rolling time window we continuously prune partial search results that would otherwise need to be tracked and would contribute to the combinatorial growth in memory utilization. Second, the temporal property of the
vertices and edges representing events suggests that it is logical to search for distinct sub-
graphs where such “temporal” vertices or edges are ordered, thus significantly reducing the
search space. Finally, we take advantage of the multi-relational structure of the data and the
characteristics of temporal events to avoid expensive joins. Given a multi-relational query
graph we decompose it in a hierarchical fashion. We design a data structure called the
Subgraph Join Tree, or henceforth referred as the SJ-Tree to model the hierarchical decom-
position and store matches with various subgraphs of the query graph as represented in the
tree. This paper demonstrates the validity of this decomposition approach towards query
processing. We refer to the smallest units of the decomposed query graph as “search prim-
itives”, which almost always consists of more than one edge. As new edges arrive over
time, we continuously perform (a) “local searches” to look for matches with the search
primitives and (b) use the decomposition structure to “join” them into progressively larger
matches. This represents a middle ground between the periodic application of a graph
search algorithm on the data graph and the approach that would have been employed by a
traditional stream database. Stream databases have no alternative but to model each edge
in the query graph as a separate join operator. Our model can express this degenerate case
where an edge is represented as a search primitive in the SJ-Tree, but the performance is
extremely poor. By grouping subgraphs into search primitives, we can simplify the query
plan, significantly improve performance by multiple orders of magnitude, and perhaps most
importantly, reason about the trade-offs involved and explore a large space of possible op-
Our contributions from this research are summarized below.

1. We introduce a data structure called \textit{SJ-Tree} for query graph decomposition (section 3.2) and present a novel subgraph search algorithm (4.1) for continuous queries on dynamic multi-relational graphs.

2. We compare our performance with the incremental subgraph isomorphism algorithm developed by Fan et al. [1] and show that our approach provides improvements by multiple orders of magnitude (section 4.4).

3. We present a series of experiments on representative online news (New York Times), co-authorship networks (DBLP) and social media data sources (Tencent Weibo) modeled as multi-relational graphs. The scale of these datasets are orders of magnitude larger than previously reported research [1, 49] in the literature (section 4.4).

4. We present a theoretical model for complexity analysis of the search algorithm. We also provide an extensive experimental analysis of the algorithm’s performance as a function of the frequency distribution of vertex labels for verification of the theoretical model.
Figure 3.2: Examples of monitoring queries on multi-relational graphs. The query on the left can be used to discover events in a certain context. Set one of the keywords to “Oil” and run the query to discover various events that center around oil, such as price movements, discovery, accident etc. By setting the keyword to “buyout”, the right query can be used to detect when news surface about a merger between two companies.

3.1 Problem Statement

First, we define the problem. We assume every edge in a dynamic graph has a timestamp associated with it and therefore, for any subgraph \( g \) of a dynamic graph we can define a time duration \( \tau(g) \) which is equal to the duration between the earliest and latest edge belonging to \( g \). Given a dynamic multi-relational graph \( G_d \), a query graph \( G_q \) and a time window \( t_W \), we report whenever a subgraph \( g_d \) that is isomorphic to \( G_q \) appears in \( G_d \) such that \( \tau(g_d) < t_W \). The isomorphic subgraphs are also referred to as matches in the subsequent discussions. If \( M(G^k_d) \) is the cumulative set of all matches discovered until time step \( k \) and \( E_{k+1} \) is the set of edges that arrive at time step \( k+1 \), we present an algorithm to compute a function \( f(G_d, G_q, E_{k+1}) \) which returns the incremental set of matches that
result from updating $G_d$ with $E_{k+1}$ and is equal to $M(G_{k+1}^d) - M(G_k^d)$. We assume that the graph only receives edge inserts and no deletions.

3.2 Subgraph Join Tree (SJ-Tree)

3.2.1 Naive Approach

We begin with a simplistic solution to motivate an incremental approach for continuous query processing. For every new edge that is added to $G_d$, we detect if the edge matches any edge in the query graph. This check can be performed minimally by examining 1) if there are edges in the query graph with the same type as the new edge and 2) if the endpoint vertices of the new edge match with the corresponding edges in the query graph based on their types and attributes. Once an edge is considered as a matching candidate, the next step is to consider different combinations of matches it can participate in.

A simple illustration of this matching process is shown in Figure 3.3. While intuitively simple, this approach falls prey to combinatorial explosion very quickly. It finds the match with the query graph at the cost of creating many partial matches. Assume that the $G_d$ receives a large number of edges that match with the query graph edge between A and B (Figure 3.3a). Let us denote this edge as $e_{AB}$. Therefore, a large number of partial matches will be created with mapping information for $e_{AB}$. Subsequently, every future
Figure 3.3: Illustration of a naive incremental algorithm. Assume AB matches with PQ, and AC matches with both QR and QS.

edge that matches with $e_{AC}$ will need to be matched or checked against all the existing partial matches for augmenting into a larger match. While the subgraph matching problem has an inherent exponential nature associated with it, a better algorithm will restrict the growth of the number of partial matches to track and still produce the correct result.

3.2.2 Our Approach

Our approach guides the search process to look for specific subgraphs of the query graph and follow specific transitions from small to larger matches. Following are the main intuitions that drive this approach,

1. Instead of looking for a match with the entire graph or just any edge of the query
1. Partition the query graph into smaller subgraphs and search for them.

2. Track the matches with individual subgraphs and combine them to produce progressively larger matches.

3. Define a *join order* in which the individual matching subgraphs will be combined.

   Do not look for every possible way to combine the matching subgraphs.

Although the current work is completely focused on temporal queries, the graph decomposition approach is suited for a broader class of applications and queries. The key aspect here is to search for substructures without incurring too much cost. Even if some subgraphs of the query graph are matched in the data, we will not attempt to assemble the matches together without following the join order. Thus, if there are substructures that are too frequent, joining them and producing larger partial matches will be too expensive without a stronger guarantee of finding a complete match. On the other hand, if there is a substructure in the query that is rare or indicates high selectivity, we should start assembling the partial matches together only after that substructure is matched.

### 3.2.3 Subgraph Join Tree (SJ-Tree) Properties

We introduce a tree structure called *Subgraph Join Tree (SJ-Tree)* that supports the above intuitions for implementing a join order based on selectivity of substructures of the
Figure 3.4: Illustration of query decomposition in SJ-Tree.

query graph.

**Definition 1** A SJ-Tree $T$ is defined as a binary tree comprised of the node set $N_T$. Each $n \in N_T$ corresponds to a subgraph of the query graph $G_q$. Let us assume $V_{SG}$ is the set of corresponding subgraphs and $|V_{SG}| = |N_T|$. Additional properties of the SJ-Tree are defined below.

**Property 1.** The subgraph corresponding to the root of the SJ-Tree is isomorphic to the query graph. Thus, for $n_r = \text{root}\{T\}$, $V_{SG}\{n_r\} \equiv G_q$.

**Property 2.** The subgraph corresponding to any internal node of $T$ is isomorphic to the output of the join operation between the subgraphs corresponding to its children. If $n_l$ and $n_r$ are the left and right child of $n$, then $V_{SG}\{n\} = V_{SG}\{n_l\} \bowtie V_{SG}\{n_r\}$. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the join operation is defined as $G_3 = G_1 \bowtie G_2$, where
such that \( G_3 = (V_3, E_3) \) where \( V_3 = V_1 \cup V_2 \) and \( E_3 = E_1 \cup E_2 \).

**Property 3.** Each node in the SJ-Tree maintains a set of matching subgraphs. We define a function \( \text{matches}(n) \) that for any node \( n \in N_T \), returns a set of subgraphs of the data graph. If \( M = \text{matches}(n) \), then \( \forall G_m \in M, G_m \equiv V_{SG}\{n\} \).

**Property 4.** Each internal node \( n \) in the SJ-Tree maintains a subgraph, \( \text{CUT-SUBGRAPH}(n) \) that equals the intersection of the query subgraphs of its child nodes.

For any internal node \( n \in N_T \) such that \( \text{CUT-SUBGRAPH}(n) \neq \emptyset \), we also define a projection operator \( \Pi \). Assume that \( G_1 \) and \( G_2 \) are isomorphic, \( G_1 \equiv G_2 \). Also define \( \Phi_V \) and \( \Phi_E \) as functions that define the bijective mapping between the vertices and edges of \( G_1 \) and \( G_2 \). Consider \( g_1 \), a subgraph of \( G_1 \): \( g_1 \subseteq G_1 \). Then \( g_2 = \Pi(G_2, g_1) \) is a subgraph of \( G_2 \) such that \( V(g_2) = \Phi_V(V(g_1)) \) and \( E(g_2) = \Phi_E(E(g_1)) \).

Conceptually, the SJ-Tree is an index structure where keys track the occurrence of matching subgraphs in the data graph. Our decision to use a binary tree as opposed to an n-ary tree is influenced by the simplicity and lowering the combinatorial cost of joining matches from multiple children. Analyzing the trade-offs between different tree models (e.g., left-deep vs bushy) is part of future work. Automated construction of the SJ-Tree for a given query is described in chapter 4.
3.3 Continuous Query Algorithm

We present a subgraph search algorithm (Algo. 1 and 2) that utilizes the SJ-Tree structure (referred to as $T$). The search process is illustrated in Figure 3.5. The SJ-Tree corresponding to this query is shown in Figure 3.6. The input to PROCESS-CONT-QUERY is the dynamic graph $G_d$, the SJ-Tree ($T$) corresponding to the query graph and the set of incoming edges. Each leaf of the SJ-Tree represents an unique subgraph of the query graph. Lines 4-6 in Algorithm 1 describe the search for each of these unique subgraphs around every incoming edge. Any discovered match is added to the SJ-Tree (line 9).
**Algorithm 1** \textsc{Process-Cont-Query}(\(G_d, T, \text{edges}\))

\textbf{Input:} \(G_d: \) Dynamic graph being queried.

\textbf{Input:} \(T: \) SJ-Tree.

\textbf{Input:} \(\text{edges}: \) New edges to be added to \(G_d\).

1: \(\text{leaf-nodes} = \text{Get-Leaf-Nodes}(T)\)

2: \textbf{for all} \(e_s \in \text{edges} \textbf{do}\)

3: \quad \text{UPDATE-GRAPH}(G_d, e_s)

4: \textbf{for all} \(n \in \text{leaf-nodes} \textbf{do}\)

5: \quad \(g^q_{sub} = \text{Get-Query-Subgraph}(T, n)\)

6: \quad \text{matches} = \text{Local-Search}(G_d, g^q_{sub}, e)

7: \quad \textbf{if} \text{matches} \neq \emptyset \textbf{then}\n
8: \quad \quad \textbf{for all} \(m \in \text{matches} \textbf{do}\)

9: \quad \quad \quad \text{T.UPDATE-SJ-TREE}(n, m)
3.3.1 Local Search

The LOCAL-SEARCH function performs a subgraph isomorphism check around the neighborhood of every incoming edge $e$. The query decomposition often reduces the local search to performing star queries where the center of the query is the vertex representing a temporal event. The peripheral vertices of the star query are the other entities that represent various attributes of the event. Further, in the context of real-time search, if the current time is $t$ and the query specifies a time window of length $t_W$ then all edges that have a timestamp older than $(t - t_W)$ are ignored from the search. In addition to filtering search candidates, we also periodically prune the SJ-Tree to remove partial matches that are older than $t_W$ from the current time. The pruning is not an expensive process. The pruning involves iterating through all the partial matches stored in the SJ-Tree. So its cost is proportional to the number of the matches stored in the SJ-Tree. The frequency of the pruning is expected to be application dependent, and for now we recommend pruning the matches at every $T W$ interval. From an efficiency perspective, we will want to prune the matches when the total number of matches or equivalently, the memory usage reaches a limit. But this limit can be hit quickly by the arrival of a high-degree vertex in the data. For such scenarios, pruning the matches can lead to missing a complete match, or in other words the search becomes a best-effort approach. Determining the ideal pruning frequency is reserved for future work.
Algorithm 2 UPDATE-SJ-TREE(node, m)
1: sibling = sibling[node]
2: parent = parent[node]
3: k = GET-JOIN-KEY(CUT-SUBGRAPH[parent], m)
4: $H_s = \text{match-tables}[\text{sibling}]
5: M^k_s = \text{GET}(H_s, k)$
6: for all $m_s \in M^k_s$ do
7: \hspace{1em} $m_{sup} = \text{JOIN}(m_s, m)$
8: \hspace{1em} if parent = root then
9: \hspace{2em} PRINT(‘MATCH FOUND : ’, $m_{sup}$)
10: \hspace{1em} else
11: \hspace{2em} UPDATE-SJ-TREE(parent, $m_{sup}$)
12: ADD(matches[node], m)
13: ADD(match-tables[node], k, m)
Algorithm 3 GET-JOIN-KEY($g_p, m$)

**Input:** $g_p$: subgraph used to project.

**Input:** $m$: subgraph match being projected.

**Output:** $key$: hashed representation of projected subgraph.

1: $s = \text{string}()$

2: for all $v \in V(g_p)$ do

3: append($s$, GET-MATCH($m, v$))

4: append($s$, “-”)

5: for all $e \in E(g_p)$ do

6: append($s$, GET-MATCH($m, e$))

7: append($s$, “-”)

8: return Hash($s$)
3.3.2 Partial Match Join and Aggregation

This subsection describes the process outlined in UPDATE-SJ-TREE. The SJ-Tree data structure maintains the sibling and parent information for every node as distinct arrays (Algo. 2, line 1-2). Each node in the SJ-Tree maintains a hash table that supports storing multiple objects with the same key. This collection of tables is denoted by the match-tables property of the SJ-Tree (Algo 2., line 4). GET() and ADD() provide lookup and update operations on the hash tables. Whenever a new matching subgraph \( g \) (available as a property of the partial match \( m \)) is added to a node in the SJ-Tree, we compute a key using its projection \( (\Pi(g)) \) and insert the key and the matching subgraph into the hash table. Algorithm 3 describes the steps for computing the key. Observe that we do not need to compute a canonical string representation of the subgraph. All matches that belong to two sibling nodes in the SJ-Tree, are first projected using the “cut-subgraph” stored at their parent node and subsequently hashed. Every match that belongs to two sibling nodes are projected using the same \( g_p \), so any ordering of vertices and edges for a given \( g_p \) is sufficient. “-” is used as a separator between numeric vertex ids.

When a new match is inserted into a leaf node we check to see if it can be combined (referred as JOIN()) with any matches that are contained in the collection maintained at its sibling node. A successful combination of matching subgraphs between the leaf and its sibling node leads to the insertion of a larger match at the parent node. This process is
repeated as long as larger matching subgraphs can be produced by moving up in the SJ-
Tree. A complete match is found when two matches belonging to the children of the root
node are combined successfully.

The JOIN operation between partial matches is critical to the overall query processing
performance. Suppose we have a query that finds a sequence of two events with a common
set of attributes. Assume that two matching events \( \text{event}_1 \) and \( \text{event}_2 \) are found with
timestamps \( \tau_1 \) and \( \tau_2 \) respectively, with \( \tau_1 < \tau_2 \). Some graph queries have a temporal order
over its edges. Therefore, when it is safe, we can report the sequence \( \{\text{event}_1, \text{event}_2\} \) and
ignore the out of order combination. Therefore, given two partial matches \( M_1 \) and \( M_2 \) with
edge sets \( \{E_i, E_j\} \) and \( \{E_m, E_n\} \) respectively, the join algorithm rejects all combinations
of these two sets that do not represent a monotonic order based on timestamps. This is
accomplished by computing a range of timestamps for each partial match. If \( t_{\text{low}}[M_i] \)
and \( t_{\text{high}}[M_i] \) are the lowest and highest timestamp for match \( M_i \), then we require that
\( t_{\text{high}}[M_1] < t_{\text{low}}[M_2] \) for joining \( M_1 \) and \( M_2 \). Observe that this is a heuristic and the
software implementation allows enabling or disabling the heuristic using a user-specified
input.
Table 3.1: Summary of test datasets

<table>
<thead>
<tr>
<th>Graph dataset</th>
<th>vertices</th>
<th>edges</th>
<th>vertex types</th>
<th>edge types</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York Times</td>
<td>64,639</td>
<td>157,019</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DBLP</td>
<td>3.158M</td>
<td>3.26M</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Tencent Weibo</td>
<td>2.5M</td>
<td>89.6M</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

3.4 Experimental Results

This section is dedicated to answering two questions: 1) How does our continuous graph query algorithm compare with the state of the art? To answer this, we compare our algorithm’s performance with the IncIsoMatch algorithm presented in [1]. 2) How does our query algorithm perform on real-world datasets? We provide the answers from exhaustive experimentation on three real-world datasets (summarized in Table 3.1) through systematic query selection.

Our metric is the time to process increments of a fixed number of edges (1k or 100k) whichever is closer to 1% of the test dataset size. The times reported only include the time spent in the query processing section and exclude graph management operations such as
adding edges and dynamic re-allocation of adjacency lists.

In this chapter we exclusively focus on the multi-relational graphs (New York Times, DBLP, Tencent Weibo), and use a template shown in Figure 3.7. It can be seen that all the query graphs in Figure 3.1-3.2 have a template-like structure that is repeated. Each template represents a temporal event and we refer to it as the event-template. Using the news events from Figure 3.2, the overall query can be viewed as a description of a sequence of temporal events sharing a common context. Each event-template includes the temporal event vertex and the other vertices that describe the event context. From exhaustive user case studies we believe that such query structures are powerful to represent a wide variety of events. As an example, one may develop a set of query graphs symbolizing different concepts and execute the queries on a news stream or an individual’s microblogging stream to reason about the evolution of news events or the person’s on-line behavior.

We also develop a performance model in terms of the label distribution in this chapter. Performance modeling in terms of the query structure is discussed in depth in the next chapter. We sample the degree distribution of every vertex type and divide the range of the degree distributions into ten intervals. For each interval, five closest candidate vertices are selected for testing purposes. Selection of multiple vertices around each bin allows us to systematically observe the impact of increasing the degree of the labeled vertex in the query graph.
Figure 3.7: The query template used to find temporal events. Experiments are performed using queries with 4 event vertices and 2 feature vertices. One feature vertex is labeled and all other vertices specify only types.

3.4.1 Experimental Setup

The results were obtained by using a single core on a 48-core shared memory system comprising 2.3 GHz Opteron 6176 SE processors and 256 GB RAM. The processor cache size is 512KB and each system node has 32 GB RAM. The code was compiled with g++ 4.1.2-52 with -O3 optimization flag on Linux 2.6.18.
Figure 3.8: Results from queries finding four articles with a common keyword and location. Legends indicate degree of query label.
3.4.2 New York Times

We use a news dataset from New York Times collected over Aug-Oct 2011 \(^1\). Each article in the dataset contains a number of facets that belong to four type of entities. Each of the articles and facets are represented as vertices in the graph. Each edge that connects an article with a facet carries a timestamp that is the publication time of the article. Following the template shown in Figure 3.7, we run a query that finds four articles where all the articles have a common keyword and location. For the location vertex we specify the

\(^1\)http://data.nytimes.com
labels shown in the top diagram in Figure 3.8 and observe the performance. As the figures
indicate, selecting labels that correspond to vertices with increasing degrees increases the
running time of the query.

Next, we compare our approach with the IncIsoMatch algorithm described by Fan
et al. [1]. The VF2 algorithm [30] is adapted to implement the graph search functionality
as outlined in IncIsoMatch. We specify a label on the feature marked with † and select
a label with one of the highest degrees for that vertex type. The queries are as follows:
1) Find four articles with a common keyword and a common organization†, 2) Find four
articles with a common entity and a common keyword † and 3) Find four articles with a
common entity and a common location †. Figure 3.9 shows a performance improvement
from our algorithm by several orders of magnitude. The multiple orders of improvement
in performance is attributed to the strictly ordered aggregation of partial matches in the
SJ-Tree and the temporal property based optimizations. The performance gap between the
processing time of the two algorithms increases as the graph grows larger. We attribute this
to the nature of the IncIsoMatch where it performs a search around every new edge in the
graph. The search spans all vertices around the endpoints of the new edge as long as they
are within $k$-hops, where $k$ is the diameter of the query graph. As the data graph grows
denser, even for a query graph with small or modest size, the $k$-hop subgraph accumulates
a large number of edges and the search becomes increasingly expensive.
Figure 3.10: Performance results for queries on the DBLP dataset. The spikes in the plot can be attributed to the bursty nature of scientific publishing where authors target the same group of conferences and journals every year.

3.4.3 DBLP Co-Authorship Network

We build a multi-relational graph representation of the DBLP citation network \(^1\) with two types of entities: authors and articles. The author name and the title of the article are stored as labels of respective vertices. We run a query to find an author (author 1) who has co-authored four papers with a specified author (author 2). Following the previously shown

\(^1\)dblp.uni-trier.de/xml
query template, our query graph has four article vertices and two author vertices. Only
one author vertex is labeled. We observe the degree distribution of the “author” vertices
and select names with progressively increasing degrees. The results are shown in Figure
3.10. It can be seen that the performance of the algorithm is quite stable for a modestly
large network with nearly 3M+ edges. Additionally, the results show that even though
vertex degree is a good indicator of the query performance, there are other factors at play.
The graph describes author-article relationship; therefore, the degree of an author vertex
provides the number of authored articles. It does not provide the information about the
number of co-authors of a person. Searching for a person who publishes a given number of
articles with fewer co-authors will lead to more partial matches and increase per-edge query
processing time. The consistent high processing times for the query containing "Hsinchun-
Chen" despite smaller degree in the graph is a result of this aspect.

3.4.4 Social Media

Finally, we present our results on a data set collected from Tencent Weibo, a Chi-
inese microblogging social network\textsuperscript{2}. The data set provides a temporal history of item
recommendations to registered users of the social network. We build a graph with 4 vertex

\textsuperscript{2}www.kddcup2012.org/c/kddcup2012-track1
Figure 3.11: Query processing time for the Tencent Weibo dataset for queries with varying selectivity.

Our test query is to detect a series of item acceptances by a group of users described by a common keyword. Following the previously shown query template, our query graph has four user vertices and one item and keyword vertex. We specify a label on the item and seek to discover the keyword that characterize the users accepting or rejecting that item. The results are shown in Figure 3.11. The figure suggests a clear trend. It shows that as the graph grows large the query processing time eventually rises sharply. It also shows that the rise happens earlier for low-selectivity queries where the specified label has higher degree in the graph. This is because the number of partial matches grows rapidly in the event of
a successful search around a high degree vertex. Every partial match from the past can potentially be merged with the latest partial match, and the partial match collection grows combinatorially over time.

This brings us to implementing the temporal window based pruning. We select the query with the highest degree label \(\text{degree(item)} = 299199\), Figure 3.11) for which the rise in the processing time was sharpest. We set the time window \(t_W\) to 1 day and prune the SJ-Tree after processing every 5 million edges. The results from the windowing enabled search is shown in Figure 3.12. Observe that the peaks in the processing time are smaller than ones observed in Figure 3.11 by an order of magnitude.

This is an extremely promising result for practical applications. Figure 3.12 suggests
that it would take 10 seconds on average to process 100k edges for a query with very low selectivity. This translates into a throughput of 0.01 million edges/second or 864 million edges per day. At the time of this writing, high volume data streams such as Twitter receive nearly 300-400 million posts every day. Considering that every user action translates into multiple edges in a graph, one may expect around billions of edges everyday. The throughput can be expected to be much higher for a query with moderate selectivity. Thus, we believe this level of throughput on a very low-selectivity query gets us close to executing real-time graph queries on such high volume data streams.

3.5 Summary

We present a novel graph decomposition based approach for continuous queries on multi-relational graphs. We introduce the SJ-Tree structure, whose nodes represent the hierarchical decomposition of the query graph. The SJ-Tree systematically tracks the evolving matches in the data graph as they transition from smaller to larger matches based on the query graph decomposition. We present experimental analysis on several real-world datasets such as New York Times, DBLP and Tencent Weibo and show that our SJ-Tree based algorithm coupled with temporal optimizations clearly outperforms the state of the art [1] by multiple orders of magnitude. Our experiments demonstrate that it is possible to execute complex multi-relational graph queries in a real-time setting. To our knowl-
edge, the results presented in this chapter are the best reported performance for such queries. These initial results are highly promising in that they suggest possible ways of auto-selecting optimal values for query processing parameters based on the data distribution.
“It is of the highest importance in the art of detection to be able to recognize, out of a number of facts, which are incidental and which vital. Otherwise your energy and attention must be dissipated instead of being concentrated.” Sherlock Holmes, The Reigate Puzzle (1894)

We presented the SJ-Tree data structure in the previous chapter and described a dynamic graph search algorithm based on the SJ-Tree structure. The SJ-Tree data structure is query specific, i.e., each query graph needs to be translated into an equivalent SJ-Tree data structure. This chapter addresses the topic of automatic generation of SJ-Trees. Given a graph stream and a query graph, can we utilize the statistics of the graph stream to suggest the most efficient search strategy? In other words, this equates the goodness of the query decomposition with the query processing efficiency. This chapter is dedicated to techniques for sampling the graph stream, utilizing stream statistics to generate SJ-Trees and finally, use quantitative metrics computed from the SJ-Tree and stream statistics to reason about different query execution strategies.

In addition to automated query decomposition, we also introduce a variant of the dynamic graph search algorithm that exploits the statistical information collected from the graph stream. We introduce a “Lazy Search” algorithm where the search strategy is decided
on a vertex-to-vertex basis depending on the likelihood of a match in the vertex neighborhood.

The following describes the key idea behind the “Lazy Search” algorithm. We approach the problem from an incremental processing perspective where search happens locally on every edge arrival. We do not search for the entire query graph around every new edge. Given a query graph, we decompose it into smaller subgraphs as ordered by their selectivity. The selectivity information is obtained using the single-edge level and 2-edge path distribution obtained from the graph stream. We store the resulting decomposition into the corresponding SJ-Tree. For a new edge in the graph, we always search for the most selective subgraph of the query graph. For other subgraphs of the query graph, a search is triggered if and only if a match for the previous subgraph in the selectivity order was obtained in the neighborhood of the new edge. We introduce an Relative Selectivity Metric that captures the hardness of a given query. Finally, we demonstrate how this metric can be used to reason about the speed-memory tradeoffs from different decompositions and select the best performing strategy.

4.1 Analysis of Continuous Query Algorithm

As a first step to understand the speed-memory tradeoff, we begin with the complexity analysis of the dynamic graph search described in Algorithm 1 and 2. Algorithm 1
describes the process of subgraph isomorphism around every new edge in the graph. Earlier we had stated our preference for small graphs as search primitives. In this section, we present the complexity analysis in terms of 1-3 edge subgraphs as candidates for search primitives.

The SUBGRAPH-ISO function performs a subgraph isomorphism check around the neighborhood of every incoming edge $e$. Given that we will perform this subgraph isomorphism on every incoming edge, our goal is to restrict the work to searching for small subgraphs preferably with 1-2 edges. The following discussion specifically focuses on searching for such small subgraphs.

**Single Edge Subgraphs** Checking if $e$ matches the query edge require comparing the types and potentially other attributes of the edges. Depending on the query constraint, we may need to look up the node label to perform a string comparison or evaluate a regular expression. We assume that a single-edge query can be matched in $O(1)$ time.

**Triads** Assume that the query graph is a triad with three vertices $v_1, v_2$ and $v_3$ and edges ordered as $e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_3, v_1)$. For any edge $e$ in the data graph, we can detect a match with $e_1$ in constant time. If $e$ is matched, we search the neighborhood of the vertex that matches with $v_2$ to search for $e_2$. Denoting this vertex as $M(v_2)$, the cost of this second level of search is $O(\text{degree}(M(v_2)))$. In case of a 3-edge subgraph, each of the successful second level searches proceed to find a match for the third edge. Thus, the cost of a 2-edge subgraph is $O(\text{degree}(M(v_2)))$ and a 3-edge subgraph is
\( O(\text{degree}(M(v_2)) \times \text{degree}(M(v_3))) \). We can refine these estimates to obtain an average cost of the search as \( O(\bar{d}_2) \) for a 2-edge subgraph and \( O(\bar{d}_2 \bar{d}_3) \) for a 3-edge subgraph, where \( \bar{d}_2 \) and \( \bar{d}_3 \) are the average degree of the vertices in the graph for the types of \( v_2 \) and \( v_3 \).

The next step is to estimate a cost for the SJ-Tree update operation (Algorithm 2). We begin with the hash-join operation (Algorithm 2, line 7). Assume the frequency of a graph \( g^i_q \) is \( n_i \), where the frequency of a subgraph is defined as the count of its instances over an edge stream of length \( N \). Therefore, over \( N \) edges, we can expect \( O(n_1) \) matches for \( g^1_q \) and \( O(n_2) \) matches for \( g^2_q \). Therefore, \( H_2 \) will be probed for a match \( O(n_1) \) times over \( N \) edges and \( H_1 \) will be probed \( O(n_2) \) times within the same period. If we knew the frequency of \( G^q \), then we can also estimate the number of new subgraphs that will be produced as the result of the hash-joins. Given that the frequency of the larger subgraph can not exceed that of the more selective component we can approximate \( O(n(G^q)) \approx \min(O(n_1), O(n_2)) \). Therefore, the average work for every incoming edge in the graph can be expressed as,

\[
\left( f_S(g^1_q) + f_S(g^2_q) + O(n_1) + O(n_2) + \min(O(n_1), O(n_2)) \right) / N.
\]

The Hash-Join combined with leaf level searches provides the simplest example of a SJ-Tree, a binary tree with height 1. In this section, we analyze the time complexity of the query processing as it happens in a multi-level SJ-Tree. Given any non-leaf node \( n \), we can obtain the expression for average work by adapting the asymptotic complexity expression shown above. Note that if a child of \( n \) denoted by \( n_c \), is not a leaf level node but an internal
node, then the term corresponding to the search cost \( f_S(g) \) disappears. Additionally, we can replace the search cost with the cost corresponding to the average work incurred by the subtree rooted by \( n_c \). Therefore, given a SJ-Tree \( (T_{sj}) \) the average work \( (C(T_{sj})) \) can be obtained by recursive computation from the root. \( C(T_{sj}) = C(root(T_{sj})) \)

4.2 Lazy Search

Let’s revisit our example. It is reasonable to assume that the “friend” relation is highly frequent in the data. If we decomposed the query graph all the way to single edges then we will be tracking all edges that match “friend”. Clearly, this is wasteful. One may suggest decomposing the query to larger subgraphs. However, it will also increase the average time incurred in performing subgraph isomorphism. A detailed study on space-time trade-offs is later presented in the experimental analysis section. Instead, we adapt a different approach.

Assume the query graph \( G_q \) is partitioned into two subgraphs \( g_1 \) and \( G^1_q \). If the probability of finding a match for \( g_1 \) is less than the probability of finding a match for \( G^1_q \), then it is always desirable to search for \( g_1 \) and look for \( G^1_q \) only where an occurrence of \( g_1 \) is found. Therefore, we select \( g_1 \) to be the most selective edge or 2-edge subgraph in the query graph and always search for \( g_1 \) around every new edge in the graph. Once we detect subgraphs in \( G_d \) that match with \( g_1 \), we follow the same approach to search for \( G_q \) in their
neighborhood. We partition $G^1_q$ further into two subgraphs: $g_2$ and $G^2_q$, where $g_2$ is another 1-edge or 2-edge subgraph.

**DATA STRUCTURES** With the SJ-Tree, the partitioning of $G_q$ is done upfront at the query compile time with $g_1$, $g_2$ etc becoming the leaves of the tree. The main difference between Lazy Search and that of Algorithm 2 is that we will be searching for $g_2$ only around the the edges in $G_d$ where a match with $g_1$ is found. Therefore, for every vertex in $u G_d$, we need to keep track of the $g_i$-s such that $u$ is present in the matching subgraph for $g_i$. We use a bitmap structure $M$ to maintain this information. Each row in the bitmap refers to a vertex in $G_d$ and the $i$-th column refers to $g_i$, or the $i$-th leaf in the SJ-Tree. If the search for subgraph $g_j$ is enabled for vertex $u$ in $G_d$, then $M[u][j] = 1$ and zero otherwise. Whenever a matching subgraph $g'$ for $g_i$ is discovered, we turn on the search for $g_{i+1}$ for all vertices in $V(g')$. This is accomplished by setting $M[v][i + 1] = 1$ where $v \in V(g')$.

**ROBUSTNESS WITH SUBGRAPH ARRIVAL ORDER** Consider a SJ-Tree with just two leaves representing query subgraphs $g_1$ and $g_2$, with $g_1$ representing the more selective left leaf. The above strategy is not robust to the arrival order of matches. Assume $G_1$ and $G_2$ are subgraphs of $G_d$ that are isomorphic to $g_1$ and $g_2$ respectively. Together, $G_1 \times G_2$ is isomorphic to the query graph $G_q$. Because we are searching for $g_1$ on every incoming edge, $G_1$ will be detected as soon as it appears in the data graph. However, we will detect $G_2$ only if appears in $G_d$ after $G_1$. If $G_2$ appeared in $G_d$ before $G_1$ we won’t find it because we are not searching for $g_2$ all the time.
We introduce a small change to address this temporal ordering issue. Whenever we enable the search on a node in the data graph, we also perform a subgraph search around the node to find any match that have occurred earlier. Thus, when we find $g_1$ and enable the search for $g_r$ on every subsequent edge arrival, we also perform a search in $G_d$ looking for $g_l$. This ensures that we will find $g_2$ even if it appeared before $g_1$.

Algorithm 4 summarizes the entire process.

### 4.3 SJ-Tree Generation

Here we address the topic of automatic generation of SJ-Tree from a specified query graph. We begin with introducing key definitions, followed by the decomposition algorithm.

**Definition Subgraph Selectivity** Given a large typed, directed graph $G$, the selectivity of a typed, directed subgraph $g$ with $k$-edges (denoted as $S(g)$) is the ratio of the number of occurrences of $g$ and the total number of all $k$-edge subgraphs in $G$.

**Definition Selectivity Distribution** The selectivity distribution of a set of subgraphs $G_k$ is a vector containing the selectivity for every subgraph in $G_k$. The subgraphs are ordered by their frequencies in ascending order.

Algorithm 5 is a greedy algorithm for decomposing a query graph into its subgraphs and producing the SJ-Tree. Its inputs are the query graph $G_d$ and an ordered set of prim-
Algorithm 4 LAZY-SEARCH($G_d$, $T$, edges)

1: \textit{leaf-nodes} = GET-LEAF-NODES($T$)

2: \textbf{for all} $e_s \in$ edges \textbf{do}

3: \hspace{1em} UPDATE-GRAPH($G_d$, $e_s$)

4: \hspace{1em} \textbf{for all} $n \in$ leaf-nodes \textbf{do}

5: \hspace{2em} $u = \text{src}(e_s)$

6: \hspace{2em} $v = \text{dst}(e_s)$

7: \hspace{2em} \textbf{if} DISABLED($u$, $n$) \text{ AND DISABLED($v$, $n$)} \textbf{then}

8: \hspace{3em} \textbf{continue}

9: \hspace{2em} $g^\text{q}_{\text{sub}} = $ GET-QUERY-SUBGRAPH($T$, $n$)

10: \hspace{2em} matches = LOCAL-SEARCH($G_d$, $g^\text{q}_{\text{sub}}$, $e$)

11: \hspace{2em} \textbf{if} matches $\neq \emptyset$ \textbf{then}

12: \hspace{3em} \textbf{for all} $m \in$ matches \textbf{do}

13: \hspace{4em} \textbf{if} $n = 0$ \textbf{then}

14: \hspace{5em} ENABLE-SEARCH-SIBLING($n$, $m$)

15: \hspace{4em} \textbf{else}

16: \hspace{5em} $M_j = $ QUERY-SIBLING-AND-JOIN($n$, $m$)

17: \hspace{5em} $p = \text{PARENT}(n)$

18: \hspace{5em} \textbf{for all} $m_j \in M_j$ \textbf{do}

19: \hspace{6em} UPDATE($p$, $m_j$)

20: \hspace{6em} ENABLE-SEARCH-SIBLING($p$, $m$)
itive \( M \). Our goal is to decompose \( G_q \) into a collection of (possibly repeated) subgraphs chosen from \( M \). Entries of \( M \) are sorted in ascending order of their subgraph selectivity.

Given a query graph \( G_q \), the algorithm begins with finding the subgraph with the lowest selectivity in \( M \). This subgraph is next removed from the query graph and the nodes of the removed subgraph are pushed into a “frontier” set. We proceed by searching for the next selective subgraph that includes at least one node from the frontier set. We continue this process until the query graph is empty. GRAPH-SEARCH performs a subgraph isomorphism operation to find an instance of \( g_M \) in \( G_q \). Algorithm 5 uses two versions of GRAPH-SEARCH. The first version uses three arguments, where the second argument is a vertex id \( v \). This version of GRAPH-SEARCH searches \( G_q \) for instances of \( g_M \) by only searching in the neighborhood of \( v \). The other version accepting two arguments searches entire \( G_q \) for an instance of \( g_M \). REMOVE-SUBGRAPH accepts two graphs as argument, where the second argument (\( g_{sub} \)) is a subgraph of the first graph (\( G_q \)). It removes all edges in \( G_q \) that belong to \( g_{sub} \). A vertex is removed from \( G_q \) only when the edge removal results in a disconnected vertex.

### 4.3.1 Selectivity Estimation of Primitives

We propose computing the selectivity distribution of primitives by processing an initial set of edges from the graph stream. For experimentation purposes we assume that the
Algorithm 5 BUILD-SJ-TREE($G_q, M$)

1: $\text{frontier} = \emptyset$

2: while $|V(G_q)| > 0$ do

3: $g_{\text{sub}} = \emptyset$

4: for all $g_M \in M$ do

5: \hspace{1em} if $\text{frontier} \neq \emptyset$ then

6: \hspace{2em} for all $v \in \text{frontier}$ do

7: \hspace{3em} $g_{\text{sub}} = \text{GRAPH-SEARCH}(G_q, v, g_M)$

8: \hspace{2em} break

9: \hspace{1em} else

10: \hspace{2em} $g_{\text{sub}} = \text{GRAPH-SEARCH}(G_q, g_M)$

11: \hspace{1em} if $g_{\text{sub}} \neq \emptyset$ then

12: \hspace{2em} $\text{frontier} = \text{frontier} \cup V(g_{\text{sub}})$

13: \hspace{1em} $G_q = \text{REMOVE-SUBGRAPH}(G_q, g_{\text{sub}})$
selectivity order remains the same for the dynamic graph when we perform the query processing. This work does not focus on modeling the accuracy of this estimation. Modeling the impact on performance when the actual selectivity order deviates from the estimated selectivity order is an area of ongoing work.

Which subgraphs are good candidates as entries of $M$? Following are two desirable properties for entries in $M$: 1) the cost for subgraph isomorphism should be low. 2) Selectivity estimation of these subgraphs should be efficient as we will need to periodically recompute the estimates from a graph stream. Based on these two criteria, we select single edge subgraphs and 2-edge paths as primitives in this study. Computing the selectivity distribution for single-edge subgraphs resolves to computing a histogram of various edge types. The selectivity distribution for 2-edge paths on a graph with $V$ nodes, $E$ vertices and $k$ unique edge types can be done in $O(V(E + k^2))$ time (Algorithm 6). In our experiments, computing the path statistics for a network traffic dataset with 800K nodes and nearly 130 million edges takes about 50 seconds without any code optimization.

Counting the frequency for larger subgraphs is important. Given a query graph with $M$ edges, ideally we would like to know the frequency of all subgraphs with size 1, 2, ..., $M - 1$. In this work, we limit our frequency collection to 1 and 2-edge subgraphs only. Collecting the frequency of larger subgraphs, specifically triangles have received a significant attention in the database and data mining community. Exhaustive enumeration of all the triangles can be expensive, specially in the presence of high degree vertices in
the data. As an example, a simple implementation of a sub-quadratic triad census algorithm on the aforementioned dataset took nearly 55 minutes to finish. Triangle counting has been extensively studied in the recent years. This includes efficient implementation of exact triangle counting algorithms for shared memory [52, 53] and distributed frameworks such as MapReduce [54, 55]. Approximate triangle counting via sampling has been studied by [56, 57], followed by extensions for semi-streaming [58] and streaming [59] scenarios. Searching for triangles incur the same I/O cost as searching for 2-edge subgraphs, as both require fetching the same amount of adjacency list information. Therefore, for queries with triangles, incorporating fast, streaming triangle counting algorithms will provide enable better query planning abilities.

Algorithms for computing k-hop paths or more complex structures such as triangles are well known but we choose to restrict our selectivity distribution estimation to 2-edge paths by considering the factors mentioned in the above paragraph.

4.3.2 Query Decomposition Strategies

Algorithm 5 shows that we can generate multiple SJ-Trees for the same $G_q$ by selecting different primitive sets for $M$. We can initiate $M$ with only 1-edge subgraphs, only 2-edge subgraphs or a mix of both. Usually it becomes a choice between the first and the last option. As an example, for a 4-edge query graph, the removal of the first 2-edge sub-
Algorithm 6 COUNT-2-EDGE-PATHS($G_d$)

1: $P = \text{Counter}()$

2: for all $v \in V(G_d)$ do

3: \hspace{1em} $C_v = \text{Counter}()$

4: \hspace{1em} for all $e \in \text{Neighbors}(G_d, v)$ do

5: \hspace{2em} $e_t = \text{Map}(e)$

6: \hspace{2em} $\text{Update}(C_v, e_t, 1)$

7: \hspace{1em} $E_t = \text{Keys}(C_v)$

8: \hspace{1em} for all $e_1 \in E_t$ do

9: \hspace{2em} $n_1 = \text{Count}(C_v, e_1)$

10: \hspace{2em} $\text{key} = (e_1, e_1)$

11: \hspace{2em} $\text{Update}(P, \text{key}, n_1(n_2 - 1)/2)$

12: \hspace{1em} for all $e_2 \in \text{LEXICALLY-GREATER}(E_t, e_1)$ do

13: \hspace{2em} $n_2 = \text{Count}(C_v, e_2)$

14: \hspace{2em} $\text{key} = (e_1, e_2)$

15: \hspace{2em} $\text{Update}(P, \text{key}, n_1n_2)$
graphs can leave us with 2 isolated edges in $G_q$. At that stage, we create two leaf nodes in the SJ-Tree with 1-edge subgraphs. For brevity we refer to both the second and third choice as 2-edge decomposition in the remaining discussions. Clearly, these 1 or 2-edge based decomposition strategies offer a choice for the space-time trade off. Searching for 1-edge subgraphs is extremely fast. However, we stand to pay the price with memory usage if these 1-edge subgraphs are highly frequent. On the contrary, we expect 2-edge subgraphs to be more discriminative. Thus, we will trade off lowering the memory usage by spending more time searching for larger, discriminative subgraphs on every incoming edge.

Considering both, an optimal decomposition is one that satisfies the query processing throughput requirement while consuming a limited amount of memory. If we perform the experiments on a single node with 16 GB memory, the memory threshold may be set to 14 GB. So we will need to pick a strategy that consumes less than 14 GB memory to store both the graph and the SJ-Tree. Given that the 1-edge decomposition is always faster than the 2-edge decomposition, we will pick the 1-edge decomposition only if the memory usage falls below the threshold.

For a SJ-Tree we can derive a lower-bound on its space requirement by using the subgraph selectivity information of its leaves. Each leaf node in the SJ-Tree stores a hash-table. The keys of the hash-table are string representations of common subgraphs between the partial matches. The value is a list of partial matches that corresponds to the same key. The subgraph selectivity distribution provides an estimate for the total number of partial
matches stored in the hash-table.

**DEFINITION Query Selectivity** We introduce a metric called *Query Selectivity*, denoted as $\hat{S}(T_k)$. Given a query-decomposition $T_k$, the query selectivity is defined as the product of the selectivities of the leaf-level query subgraphs.

$leaves(T_k)$ returns the set of leaves in a SJ-Tree $T_k$. Given a node $n$, $V_{SG}(T, n)$ returns the subgraph corresponding to node $n$ in SJ-Tree $T$. Finally, $S(g)$ is the selectivity of the subgraph $g$ as defined earlier.

$$S(\hat{T}_k) = \prod_{n \in leaves(T_k)} S(V_{SG}(T_k, n))$$ (4.1)

**DEFINITION Relative Selectivity** We introduce a metric called *Relative Selectivity*, denoted as $\xi(T_k, T_1)$. Given a 1-edge decomposition $T_1$ and another decomposition $T_k$, we define $\xi(T_k, T_1)$ as follows.

$$\xi(T_k, T_1) = \frac{S(\hat{T}_k)}{S(T_1)}$$ (4.2)

**THEOREM 1** Given the data graph $G_d$ at any time $t$, assume that the query graph $G_q$ is not guaranteed to be present in $G_d$. Then initiating the search for $G_q$ by searching for $g_{rare}$ where $g_{rare} \subset G_q$ and $\forall g \subset G_q |E(g)| = |E(g_{rare})|$, $frequency(g) > frequency(g_{rare})$ is in optimal strategy.

**PROOF** The time complexity for searching for a $O(1)$ for a 1-edge subgraph and $O(d_v)$ for a 2-edge subgraph. Therefore, the runtime cost to search for $g_{rare}$ is same as any other
subgraph of $G_q$ with the same number of edges. However, searching for $g_{rare}$ will require minimum space because it has the minimum frequency amidst all subgraphs with same size. Therefore, searching for $g_{rare}$ is an optimal strategy.

**Theorem 2** Given a set of identical size subgraphs $\{g_k\}$ such that $\bigcup_k g_k = G_q$, a SJ-Tree with ordered leaves $g_k \prec g_{k+1} \prec g_{k+2}$ requires minimal space when $frequency(g_k \ltimes g_{k+1}) < frequency(g_{k+2})$.

![Figure 4.1](image)

**Proof** By induction. Assume a SJ-Tree with three leaves as shown in Figure 4.1. Following the definitions of SJ-Tree, this is a left-deep binary tree with 3 leaves. Therefore, the $frequency(c) = \min(frequency(a), frequency(b))$. Substituting for the frequency of $c$, space requirement for this tree $S(T) = f(a) + f(b) + f(d) + \min(f(a), f(b))$. Thus, the space requirement for this tree is minimum if $f(a) < f(b) < f(c)$.

Now we can consider any arbitrary tree where $T_n$ refers to a tree with a left subtree $T_{n_1}$ and a right child $l_{n+2}$. Above shows that $T_1$ constructed as above will have minimum space requirement, and so will $T_2$ if $f(a) < f(b) < f(c) < f(d)$.
Observation 3 Given $g_k$, a subgraph of query graph $G_q$, it is efficient to decompose $g_k$ if there is a subgraph $g \subset g_k$, such that $\text{frequency}(g) > \left( \frac{\text{frequency}(g_k)}{\bar{d}|V(g_k)|} \right)$, where $\bar{d}$ is the average vertex degree of the data graph and $|V(g_k)|$ is the number of vertices in $g_k$.

Proof Given a graph $g$, the average cost for searching for another graph that is larger by a single edge is $\bar{d}$ multiplied by the number of vertices in $g_k$, and the proof follows.

4.4 Experimental Studies

We present experimental analysis on two real-world datasets (New York Times $^1$ and Internet Backbone Traffic data $^1$), and a synthetic streaming RDF benchmark. The experiments are performed to answer questions in the following categories.

1. Studying Selectivity Distribution What does the selectivity distribution of 2-edge subgraphs look like in real world datasets? What is the duration of time for which the selectivity distribution or selectivity order of 2-edge subgraphs remain static?

2. Comparison Between Search Strategies In the previous sections, we intro-
duced two different choices for query decomposition (1-edge vs 2-edge path based) and two different choices for query execution (lazy vs non-lazy). How do the strategies compare?

3. **Automated Strategy Selection** Given a dynamic graph and a query graph, can we choose an effective strategy using their statistics?

### 4.4.1 Experimental setup

The experiments were performed on a 32-core Linux system with 2.1 GHz AMD Opteron processors, and with 64 GB memory. The code was compiled with g++ 4.7.2 compiler with -O3 optimization.

Given a pair of data graph and query graph, we perform either of two tasks: 1) query decomposition and 2) query processing.

**Query decomposition:** Query decomposition involves loading the data graph, collecting 2-edge subgraph statistics and performing query decomposition using the selectivity distribution of the subgraphs. The SJ-Tree generated by the query decomposition algorithm is stored as an ASCII file on disk.

**Query processing:** The query processing step begins with loading the query graph in memory, followed by initialization of the SJ-Tree structure from the corresponding file generated in the query decomposition step. We initialize the data graph in memory with
zero edges. Next, edges parsed from the raw data file are streamed into the data graph. The continuous query algorithm is invoked after each AddEdge() call to the data graph.

4.4.2 Data source and Query description

Summaries of various datasets used in the experiments are provided in Table 4.1. We tested each dataset with a set of randomly generated queries. The following describes the individual datasets and test query generation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Type</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York Times</td>
<td>Online News</td>
<td>64,639</td>
<td>157,019</td>
</tr>
<tr>
<td>Internet Backbone Traffic</td>
<td>Network traffic</td>
<td>2,491,915</td>
<td>19,550,863</td>
</tr>
<tr>
<td>LSBench/CSPARQL Benchmark</td>
<td>RDF Stream</td>
<td>5,210,099</td>
<td>23,320,426</td>
</tr>
</tbody>
</table>

**New York Times**: The New York Times dataset contains articles collected from 2013 July-September time period using Version 3 of its data collection API available at data.nytimes.com. Each article in the dataset contains a number of facets that belong to four type of entities: person, geo-location, organization and topic. Each of the articles and facets are represented as vertices in the graph. Each edge that connects an article with
a facet carries a timestamp that is the publication time of the article. The New York Times dataset was tested with a set of 10 randomly generated k-partite graphs.

**Network Traffic** The second dataset is an internet backbone traffic dataset obtained from www.caida.org. CAIDA (Cooperative Association for Internet Data Analysis) is a collaborative program that provides a wide collection of network traffic data. We used the “CAIDA Internet Anonymized Traces 2013 Dataset” for experimentation. The dataset contains 22 million network traffic flow records collected over a one minute period. We excluded the traffic to/from IP addresses matching patterns 10.x.x.x or 192.168.x.x. These address spaces refer to private subnets and a communication from a given IP address from these spaces can actually refer to multiple physical hosts in the real word. As an example, every internet service provider configures the routers or machines inside a home network with IPs selected from the private IP address range. Therefore, if we see a request from 192.168.1.1 to google.com, there is no way to determine the exact origin of this communication. From a graph perspective, allowing private IP address and the subsequent aggregation of communication will result in the creation of vertices with giant neighbor lists, which will surely impact the search performance. A detailed list of use cases describing subgraph queries for cyber traffic monitoring are described in [60]. The Internet Backbone traffic dataset was tested with a set of 100 randomly generated path queries with diameter 4. We randomly vary edge directions on the path as well as the edge types.

**Social Media Stream** Our final test dataset is a RDF social media stream available
from the Linked Stream Benchmark [61]. We generated the dataset using the sibgenerator utility with 1 million users specified as the input parameter. The generated graph has a static and a streaming component. The static component refers to the social network with user profiles, and social network relationships. The streaming component includes 3 streams. The GPS stream includes user checkins at various locations. The Post and Comments stream includes posts and comments by the users, subscription by users to forums, and a stream of “likes” and “tags”. Finally, the photo stream includes information about photos uploaded by users, and tags and likes as applied to photos.

4.4.3 Selectivity Distribution

We show the distribution of edge types and unique 2-edge subgraphs in Figure 4.2 and 4.3. Figure 4.2 shows the edge distribution plotted over time. Each vertical bar in the plot shows the distribution of different edge types collected over a fixed interval of edges. The distribution is not cumulative. Therefore, each vertex bar shows the distribution of edges obtained from last $N$ edges, where $N$ is the fixed interval for collecting the distribution. The key observation is that the relative order of different types of edges stays similar even as the graph evolves.

Figure 4.3 shows that all the 2-edge path distributions are heavily skewed. A small number of 2-edge subgraphs are found to dominate the distribution across all the datasets.
Figure 4.2: Edge type distribution shown with the evolution of the dynamic graph. X-axis shows the number of cumulative edges in the graph as it is growing. The plotted distribution is not cumulative. The edge distribution is collected after fixed intervals. The interval is 10 thousand, 100 thousand and 1 million respectively. There are 4, 7, and 45 edge types in these datasets.
Figure 4.3: 2-edge path distribution in each test data set. Each point on X-axis represents a unique 2-edge path and Y-axis shows its corresponding count.
The skew is heaviest for the LSBench dataset, which is expected given the higher number of unique edge types and the larger size of the dataset.

The goal of this analysis is to observe the variability in the selectivity distribution over time. The selectivity distribution is expected to vary over time. However, it is the relative order of the unique single edge or 2-edge subgraphs that matters from the query decomposition perspective. For each of the test datasets, we took multiple snapshots of the selectivity order and found it to be stable, except with fluctuations for the very low frequency components (data points on the left end of the distributions in Fig. 4.3). Significant changes in the selectivity order can adversely impact the performance of the query. Estimating the duration over which the selectivity ordering stays stable for a given data stream, quantification of errors based on shift in the distribution, and adapting the query algorithm to handle such shifts is reserved for future work.

4.4.4 Query Performance Analysis

This section presents query performance results obtained through query sweeps on each of the three datasets. Each dataset was tested with the set of randomly generated queries as described above. For each query, we collect performance from 4 different query execution strategies obtained by 1-edge or 2-edge decomposition of a query graph and the lazy vs. track everything approach adapted by the query algorithm. The following tags are
used to describe the plots in the remainder of the paper: a) “Single”: 1-edge decomposition, search tracks all matching subgraphs in SJ-tree, b) “SingleLazy”: 1-edge based query decomposition, use “Lazy” approach to search, c) “Path”: 2-edge decomposition, search tracks all matching subgraphs in SJ-Tree, and d) “PathLazy”: 2-edge decomposition with “Lazy” search.

We analyze the query performance data with the following goals in mind.

1. Which decomposition and query execution strategy scales best? How does runtime, number of matches and memory usage increase with the evolution of the dynamic graph?

2. Is there variation in performance across the range of queries and if so, can we explain that through the statistics from data and query graphs?

3. Given a query graph, can we use our knowledge of data and query graph statistics to automatically select a good execution strategy?

**New York Times**

We begin with our analysis on New York Times (NYT) data. Ten query graphs were generated using the template as shown in Figure 4.4. Some example queries are shown in Figure 4.5. Figure 4.6 shows two decompositions of an example query. Type of vertices 1 and 2 were kept fixed as “articles”, whereas the type of vertices 3 and 4 were permuted between author, location, organization and topic. The edge types were also changed to
keep them consistent with the types of end-point vertices. For example, setting vertex 3 as “author” from “location” required setting the edge (1, 3) type to “has-author” (from “has-location”).

![Diagram](image)

**Figure 4.4:** Query template for New York Times.

Figure 4.7 shows the total run time for processing 100,000 edges using each of these strategies. Evidently, the "SingleLazy" strategy that combines lazy-search with 1-edge based decomposition is the best performing strategy. Next, we investigate the relative performance of these strategies in more detail by studying the individual decompositions and degree distribution. As we discussed in the time complexity analysis (section 4.1), the subgraph isomorphism cost for single edges is $O(1)$ and $O(d)$ for 2-edge subgraphs, where $\bar{d}$ is the average degree of the target vertex $v_2$ in the query subgraph. To verify this, we plotted the ratio between the “Path” and “Single” strategies (Figure 4.9). For the 2-edge or path based decomposition, 8 out of 10 queries were searching for a 2-edge subgraph that has an article vertex and other the two vertices chosen from the following combinations: topic and topic, organization and organization, person and person, geo-location and geo-location,
Figure 4.5: Example queries used for testing New York Times data.
**Figure 4.6**: 1 and 2-edge based decompositions of the network flow query graph shown in Figure 4.5a.

**Figure 4.7**: Query processing times using four different search strategies for New York Times data.
**Figure 4.8:** Degree distribution from New York Times data based on vertex types.

**Figure 4.9:** Analyzing relative performance between the top-3 search strategies for New York Times data.
topic and organization, topic and person, topic and geo-location, organization and person. Two of the remaining decompositions resulted in searching for a 2-edge subgraph that has a person connected to two distinct articles, or a geo-location connected to two distinct articles. The degree distribution for each vertex type is shown in Figure 4.8, and the average degree for vertices of type “article” is 4.03. As Figure 4.9 shows, the speedup from “Path” to “Single” closely approximates the average degree of article vertices, as predicted in the complexity analysis.

We were surprised by the “PathLazy” approach taking a disproportionate amount of time. We discovered that for most of these queries, the “PathLazy” approach resorted to searching for a “twig” graph (example: an article connected to a person and a topic) and once it was found, it spawned off a search for another twig graph from all the matching vertices. As the degree distribution shows, there are many high-degree vertices in the non-article (geo-location, organization, person, topic) category. Therefore, any time the search finds a match around a high degree vertex such as “geo-location:New York” or “topic:Politics and Government”, it performs a second search. The high runtime of “PathLazy” results from a large number of occurrences of this event.

Next, we observe the impact of adopting the Lazy approach for the “Single” strategy. The lower plot in Figure 4.9 shows the speedup from Lazy Search as a function of edge selectivity. The speedup is seen to be higher with higher edge selectivity. The edge selectivity is higher if the probability of individual edges appearing in the graph stream is high.
However, the probability of an edge’s appearance in the graph stream does not provide us with any information about the presence of the query match. Given the lazy approach guarantees the savings in search time, and consequently, reduces the number of partial matches being tracked, the savings are higher when the likelihood of the appearance of individual edges in the graph stream is high.

**Network Traffic (CAIDA)**

Here we present the results from the Netflow and LSBench dataset. For the netflow data, we collect performance results from 100 random queries. Some example queries are shown in Figure 4.10. Figure 4.11 shows two decompositions of an example query. The query window is still kept fixed at 100,000 edges and we report the average runtime by running the query at 0, 5, 10 and 15 million edge offsets in the stream. We plot the runtime and memory utilization as a function of edge selectivity. Figure 4.16 summarizes the results from this analysis. We find the “SingleLazy” and “PathLazy” are the best search approaches in terms of runtime. We attribute the superior performance of the “PathLazy” approach (as compared to the New York Times data) to the higher selectivity of queries as indicated by larger diameter. While it is still likely that the search can run into a high-degree vertex, the probability of searching for a subgraph 2 or 4 hops away from the first partial match is quite less.

The plots collected at various intervals of the graph stream are shown in Figure 4.12
- 4.15. Figure 4.16 provides a summary of the performance results. The “SingleLazy” clearly outperformed the “PathLazy” approach when the graph was small, but as it grew larger their runtimes became comparable for queries with lower probabilities of occurrence. This is desirable because we would like to have either strategy work well for queries with high selectivity. For queries with lower selectivity (right end of the plot), the “SingleLazy” approach clearly wins in terms of runtime. However, the space utilization can be seen to rise by two orders of magnitude to provide an order of magnitude in the speedup.

We also profiled different components of the query processing such as the time spent in performing subgraph isomorphism and the time spent in updating the SJ-Tree. The latter is largely composed of the time spent in looking up the hash tables in various nodes of the SJ-Tree, performing joins between partial matches and inserting new entries. We also find that the subgraph isomorphism operation (for 1 or 2-edge subgraphs) dominates the processing time. Considering both classes of queries with diameter 4 and 5, the subgraph isomorphism operation consumes more than 95% of the total query processing time.
Figure 4.10: Example path queries used for testing CAIDA data.

Figure 4.11: 1 and 2-edge based decompositions of the network flow query graph shown in Figure 4.10a.
Figure 4.12: Runtime collected for diameter 4 queries at different intervals.

(a) search-offset=5M.

(b) search-offset=10M.

(c) search-offset=15M.
Figure 4.13: Memory usage collected for diameter 4 queries at different intervals.
Figure 4.14: Runtime collected for diameter 5 queries at different intervals.
Figure 4.15: Memory usage collected for diameter 4 queries at different intervals.
Figure 4.16: Summary of results from Netflow.
LSBench

Figure 4.19 shows plots of query processing statistics collected from randomly generated path queries with diameter 4 on the LSBench dataset. Some example queries are shown in Figure 4.17. Figure 4.18 shows two decompositions of an example query. The plots show runtime and number of matches in the SJ-Tree in terms of the relative selectivity of the queries. Also, the performance statistics were collected at two different points in the stream: at the beginning and at the offset of 5M triples. The size of the query processing window was fixed at 1M triples. The results agree with the findings from the experiments on the netflow data; “SingleLazy” was the fastest strategy in terms of speed, while “Path-Lazy” consumed the least amount of memory.

4.5 Summary

This chapter was a logical continuation of the previous chapter on data structures and algorithms. We developed query planning algorithms to generate a SJ-Tree for any query graph by exploiting its structural and semantic characteristics. The novelty of our approach lies in its use of the statistics collected from the graph stream. We went further to introduce a “Lazy” variant of the dynamic graph search algorithm that exploits the varying selectivity between different parts of a query graph. We also extensively studied the speed and memory trade-offs between different search strategies, performing experiments on three different
Figure 4.17: Example path queries used for testing LSBench data.
datasets and multiple query classes. Our studies indicate that for queries with high and moderate selectivity, “SingleLazy”, a strategy that combines 1-edge SJ-Tree decomposition with Lazy Search is the best approach. For queries with low selectivity, “PathLazy”, a strategy combining 2-edge based SJ-Tree decomposition and Lazy Search appears to be the best approach.
Figure 4.19: Experimental results from LSBench.
CHAPTER 5. PARALLEL AND DISTRIBUTED APPROACHES

“Data! Data! Data!” he cried impatiently. “I can’t make bricks without clay.”

*Sherlock Holmes, The Adventure of the Copper Beeches (1892)*

We focus on the Pregel [62] and PowerGraph [63] frameworks. Section 2 focuses on the Bulk-Synchronous Processing Model since it is the main concept underlying both Pregel and PowerGraph. Section 3 focuses on Pregel and its open source implementation, Apache Giraph. Section 4 describes the PowerGraph abstraction. Section 5 focuses on implementing Subgraph Isomorphism in the Pregel framework.

The ability to manage and analyze large graph datasets is a necessary problem for practitioners in the knowledge discovery and data mining community. Graph datasets with the number of nodes ranging in the billions are becoming commonplace for domains such as social networks and cyber-security. Say we have a data graph with $n$ billion vertices and $m$ billion edges, and we want to store this using an adjacency list data structure. Assume each adjacency list just stores the id of the neighboring nodes, and we store both incoming and outgoing connections in each adjacency list. If an id is stored as an 8-byte integer, this will require $16m$ GB of memory. Given that we also need to store pointers (using 64 bit addressing) to each adjacency list, the total required memory is $(8n + 16m)$ GBs. Therefore, a graph with 100 million nodes and 1 billion edges will require 16.1 GBs of
memory, a graph with 1 billion nodes and 1 billion edges will require 24 GBs of memory and a graph with 1 billion nodes and 10 billion edges will require 168 GBs of memory. Many of the datasets of interest today are multi-relational in nature and consequently, each of the nodes and edges can have attributes that are important for various graph mining tasks. Thus, it is easy to see how the memory requirement for storing the graph can explode very quickly.

The above numbers point to the problem of scale. Today a standard workstation is equipped with 4-8 GBs of memory and commodity servers are equipped with 32-64 GBs of memory. Clearly, even such high-end servers are not be capable of performing in-memory computation on graphs at scales described above. The challenge of in-memory versus out-of-core computation is not new and has been addressed by the high-performance computing (HPC) community for decades. However, most of the applications targeted by the HPC community benefits from linear memory access or high locality. In simple terms, this means algorithms processing a single-dimensional array or high-dimensional matrices typically end up following sequential memory access patterns. If memory location \( i \) is accessed, then memory location \( i + 1 \) is fetched with high probability and this is heavily exploited by cache enabled processors. On the other hand, graphs typically follow an irregular access pattern; a depth-first traversal may require accessing the adjacency list of nodes \( i, j, k \) and none of the memory locations are adjacent. Given that the cost (or latency) of accessing the memory (DRAM) is higher than fetching the data from the cache, this naturally leads
to lower performance. The performance is worse if the requested data needs to be fetched from a memory unit attached to another processor on the same computing node and worst when the data has to be fetched from another machine across the network.

The parallel and distributed computing community has traditionally approached this problem in two ways. A) First, hiding latency by resorting to massive scale parallelism or multi-threading. Here the objective is to continually keep a large number of memory requests in-flight so that the processor always gets to work on an incoming response to a past memory request. B) The second approach is to optimize the memory access for intra and inter-node access. Improving intra-node access refers to multi-processor systems that may have 2 quad-core processors and each processing core may continuously access the DRAM unit attached to the socket of the other processor. Inter-node communication requires optimizing the data transfer across the network by implementing a high-speed network interconnect to connect the systems, changing the network topology to reduce worst case memory access latencies and collective communication, where many small packets between the same source and target host can be optimally delivered by coalescing into a large message.

(A) is expensive as it requires custom hardware that may not be suitable for a wider class of applications. (B) typically applies to a broad swath of applications both in the HPC community and in cloud computing applications. Loosely speaking, these are also the distinctive approaches of the parallel and distributed computing communities. The success
of the Map-Reduce framework in scaling on big data problems using commodity hardware is a major attraction for embracing the distributed computing approach.

5.1 Bulk Synchronous Processing

The Bulk Synchronous Processing (BSP) model was proposed by Leslie Valiant in 1990 [64]. The BSP model is defined as a combination of three attributes,

1. A number of components, each of which execute programs or perform memory operations. The program execution happens on data local to each component. Thus, the computations assigned to each component are independent.

2. A router to deliver messages from one component to another.

3. Facility to synchronize all or a subset of components at regular intervals of $L$ time units where $L$ is the periodicity parameter.

A computation consists of a sequence of supersteps (Figure 5.1). In each superstep, each component carries out some local computation based on an assigned task and sends (receives) messages from other components. After each $L$ units of time, a global check is made if the superstep is finished by all components. The next superstep begins if all components have finished the superstep. Else, the next period of $L$ units is allocated to the unfinished superstep. The horizontal bars in Figure 5.1 represent multiple tasks that are
executing within each super step. As can be seen, superstep 1 begins only after all the tasks in superstep 0 are finished and a synchronization was performed across tasks.

![Diagram](diagram.png)

**Figure 5.1:** Illustration of a BSP computation.

The BSP model assumes that programs are written for \( v \) virtual processors to run on \( p \) physical processors, where \( v \) is much larger than \( p \) (ideally \( p \approx \log(v) \)). This strategy of over-decomposing the problem and over-subscribing the processors is key to load balancing. Also, the model distinguishes the role of processing components and the router to emphasize the distinct tasks of computation and communication. It does not ignore the costs of communication and synchronization operations.

### 5.1.1 Pregel

**Background**

Before we delve into the internals of Pregel and PowerGraph, it is worth understanding why they came into being. The Parallel Boost Graph Library (BGL) [65] had developed the idea of storing a graph across multiple machines since 2005. However, the Map-Reduce
framework [66] is the de-facto standard for large-scale data processing today. Its success has championed two major concepts, cloud computing and fault tolerance. None of these are first-order design goals for Parallel BGL. While MapReduce demonstrated the ability to scale on massive datasets, it was also realized that the key-value pair based abstraction, which is a central tenet of the Map-Reduce framework, does not serve graph computations well. Suri and Vassilvitskii [55] provide an excellent discussion of this problem.

**Vertex State Machine and Message Passing**

The Pregel system was developed at Google to design a system that excels at problems with irregular memory access and yet preserves the positive aspects of the MapReduce framework: the functional programming interface, fault-tolerance and scaling on cloud. The input to a Pregel computation is a directed graph where each vertex stores the outgoing edges from that vertex. Each vertex and edge in the graph contains a mutable value of user-defined type. It implements the bulk-synchronous processing model described above, where the framework invokes an user-specified function on every vertex during a superstep. The user-specified function describes a local computation or a ”vertex program” for a single vertex \( v \) in the graph during a single superstep. At superstep \( S \), it can read messages that were sent to vertex \( v \) in superstep \( S - 1 \) by other vertices. It can also modify the state of the vertex \( v \) and its outgoing edges in superstep \( S \) and send messages to other vertices in the graph which will be received in superstep \( S + 1 \). Observe that the recipient of a
message is not required to be a neighbor of \( v \). There is no guaranteed order of delivery for messages. This description of localized computation at a vertex is the closest extent of similarity with the MapReduce framework. Just as the MapReduce framework advocates describing any algorithm in terms of a map and reduce function, Pregel requires a “Think like a vertex” philosophy for algorithms to be implemented in this framework.

In addition to the vertex program execution, Pregel also introduces the concept of a vertex state machine. In this state machine, every vertex can be either in an active state or a halted state. At the beginning of a computation every vertex in the graph is set in an active state. A vertex can vote itself into a halted state by implementing the suitable logic into the vertex program. A halted vertex can be reactivated by messages from other vertices. Unless a maximum number of supersteps are specified, the sequence of supersteps continue as long as there is a single vertex in the active state. Figure 5.2 shows an illustration of a vertex program executing over multiple supersteps, with the number of active vertices reducing with multiple iterations.

**Topology Mutation**

A vertex is allowed to request addition or deletion of vertices from the graph. This is a critical feature for algorithms that mutate the graph such as shrinking the graph by collapsing a set of nodes into supernodes. Similar to messages, if a set of mutations were requested in superstep \( S \), all changes becomes visible from all remaining vertices in super-
Figure 5.2: Progression of a Pregel computation in a 4-vertex graph. Dotted lines indicated messages and shaded vertices are voted to halted state.

step $S + 1$. Mutation operations targeted at the same entity (node or edge) in the graph are handled via a pre-defined partial ordering imposed on operations as edge removal, vertex removal and additions.

Implementation Details

Figure 5.3 shows the core API that a vertex program needs to implement. Quite evidently, the Compute() function implements the vertex-centric algorithm and SendMessage() and VoteToHalt() are used to change the state of a vertex.

In addition to implementing this core API, the user program can influence the execution of the overall computation in other ways as well. As described earlier each vertex data structure stores the values of the vertex, its id and a list of all outgoing edges. The entire graph is split into multiple partitions where each vertex is assigned to a partition based on
a partitioning function. By default this function is hash(vertex-id) mod \( N \), where \( N \) is the number of partitions. If an algorithm has an idea about the locality of message passing then it may supply a custom partitioning function to increase the likelihood of communicating vertices being stored in the same partition. This can significantly improve performance by reducing the communication over the network.

```cpp
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
  public:
    virtual void Compute(MessageIterator* msgs) = 0;

    const string& vertex_id() const;
    int64 superstep() const;

    const VertexValue& GetValue();
    VertexValue& MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
                        const MessageValue& message);
    void VoteToHalt();
};
```

**Figure 5.3:** The core API for a vertex program.

### Comparison with MapReduce

With the Pregel framework defined, let us look at the implementation of the PageRank algorithm to see the difference in execution from a MapReduce based implementation. A MapReduce implementation will go through the following steps.

1. Read in key-value pairs from disk, which are \{node-id, node-value\}. Node-value is
defined as \{\text{pagerank}, [\text{neighbor-list}]\}.

2. Map emits the key-value pair shown above.

3. Data is sorted to disk and sent to reducers where it hits the disk again.

4. Reducer sums the pagerank from all neighbors, joins with the Node-Value structure and emits to disk for another iteration.

5. The processes is repeated until the pageranks converge.

Apart from the load balancing issues that result from the power-law distribution of most real-world graphs and the usual inter-cluster communication issues, we have two additional costs here: sorting and data hitting disk too many times. This problem associated with the MapReduce framework is quite well known and has been addressed by work such as HaLoop [67]. On the other hand, Pregel begins with loading the graph from disk and assigns partitions to each worker. Each worker executes the Compute() function and carries out inter-vertex message passing. Checkpoint happens after the messages are delivered to workers for the next superstep and this iteration continues until convergence. Quite clearly, the extra cost of sorting and disk-based computation is avoided here.
5.1.2 Apache Giraph

Apache Giraph [68] is an open-source implementation of the Pregel framework. It is designed to run on the Java based Hadoop infrastructure. As the following example shows, its design is highly modular. A typical Giraph program can be executed as follows:

```
    hadoop jar giraph.jar < arg1 > −inputFormat < arg2 > −inputPath < arg3 > −outputFormat < arg4 > −outputPath < arg5 > −workers < arg6 >
```

The first argument, or arg1 is the name of the VertexProgram to execute. Typically, this program contains the algorithm an user may implement or it could be an algorithm such as PageRank that is pre-complied into the giraph.jar library. arg2 and arg4 requires specifying an input and output format for the input graph. Again, the user can choose from pre-compiled options for a number of popular input formats. Once the input and output formats are known, the program needs to know the location of input and output directories. Finally, the user needs to specify the number of workers to indicate the level of parallelism.

5.1.3 PowerGraph

PowerGraph has its origins in GraphLab [69], an asynchronous, distributed shared-memory abstraction where vertex programs have access to a distributed graph with data stored on each vertex and edge. Unlike a Pregel program, a GraphLab vertex program
proceeds by directly modifying state information on the current vertex, adjacent edges and adjacent vertices irrespective of edge direction. The framework ensures serializability by ensuring two neighboring vertices are not executing the same vertex program at the same time.

The initial incarnations of the GraphLab framework were strongly focused on statistical machine-learning examples. Although it subscribed to the philosophy of executing vertex programs, the framework was not designed with massive-scale graphs and graph-analytics algorithms as major drivers. Very recently, the GraphLab group released a new framework called PowerGraph [63] where the authors address various challenges that arise in processing power law graphs. The authors claim that PowerGraph exploits the structure of Vertex Programs and tries to factor computation over edges instead of vertices. Given that power-law graphs have a small set of vertices connecting to a majority of vertices, treating edges as the units of computation leads to a greater amount of parallelism and improved load balancing. A vertex program operating on vertex $v$ is split into three phases to accomplish this.

1. Accumulation: In this phase $v$ accumulates the changes to its own state by calling a gather() function in parallel on all or a subset of its edges. The gather function has access to the state of all adjacent edges and adjacent vertices.

2. Apply: The accumulated change is applied to the state value of vertex $v$. The updated
value is applied to the underlying graph as an atomic update.

3. Scatter: In this phase a scatter() function is invoked in parallel on all or a subset of edges adjacent to \( v \). This typically updates the values of the adjacent edges, and the updated values are pulled in by the neighboring vertex when it invokes the gather function on the same edge.

Observe that the gather and scatter functions can be invoked on a subset of \( v \)'s neighbors. For such cases the vertex program needs to specify the subset to invoke for each phase.

The key takeaway message here is that by scheduling the gather and scatter operations on a large number of edges in the entire graph (which is much larger than the number of vertices in the graph) the PowerGraph model is able to extract a larger amount of parallelism as compared to a strictly vertex centric model of computation.

5.1.4 Implementing Subgraph Isomorphism

Algorithm 7 shows the pseudocode for an implementation of subgraph isomorphism as a vertex program. The edges in the query graph are ordered prior to starting the BSP supersteps. If \( E_q \) is the list of ordered query graph edges then in superstep \( s \) every vertex tries to match its neighboring edges with the edge \( E_q[s] \). Therefore, \( M \) edges in the query graph implies that there will be at most \( M \) supersteps. Lines 3-9 describe the logic for
Algorithm 7 SubgraphIsomorphismVertexProgram($v, messages$)

1: $s = \text{GET-SUPERSTEP}()$

2: $e_q = \text{GET-QUERY-EDGE}(s)$

3: if $messages = \emptyset$ then

4: $u = \text{GET-QUERY-VERTEX}(s)$

5: if $\text{matches}(v, u)$ then

6: for all $e_s \in \text{EDGES}(v)$ do

7: if $\text{matches}(e_q, e_s)$ then

8: $m = \text{INIT-MATCH}(e_q, e_s)$

9: $\text{sendMsg}(\text{target}(e_s), m)$

10: else

11: for all $m \in messages$ do

12: for all $e_s \in \text{EDGES}(v)$ do

13: if $\text{matches}(e_q, e_s)$ then

14: $m_{new} = \text{AUGMENT-MATCH}(m, e_q, e_s)$

15: if $s = M$ then

16: $\text{OUTPUT-FULL-MATCH}(m_{new})$

17: else

18: $\text{sendMsg}(\text{target}(e_s), m_{new})$

19: VoteToHalt()
the first superstep, when every vertex in the graph is checked to see if it matches with the
the source vertex of the first query edge. If there is a match, then it sends a message with
the partial match information to the the neighboring vertex. The neighboring vertex will
process that message in the following superstep and try to see if that partial match can be
augmented. Every vertex votes itself to the halted state after every superstep. If there are
any messages from neighbors in the next superstep, it gets reactivated.

5.2 Software Multithreading with Partitioned Global Address Space

We implement the dynamic graph search algorithm on the SGEM/GMT software
framework. GMT (Global Memory and Threading) [70] is a runtime system library that
couples software multithreading and message aggregation together with a Partitioned Global
Address Space (PGAS) data model to enable better performance and scaling of irregular
applications on multi-node systems. SGEM [71] is a software framework that provides a
set of distributed data structures and algorithms developed atop GMT.

5.2.1 System Level Architecture

GMT integrates a PGAS locality-aware global data model together with lightweight
software multithreading and message aggregation. The Partitioned Global Address Space
(PGAS) programming model [72] is a promising solution to develop applications with a shared memory abstraction on distributed memory clusters, without neglecting locality principles. The PGAS data model enables the allocation and access of difficult to partition datasets in the global, aggregate memory of a cluster. Typically, PGAS remote data access primitives have been designed and optimized for more regular applications and data sets. Multithread-ing and message aggregation may enable better support for irregular work-loads. Multithreading can be effectively used to hide the latencies of memory or network data accesses. For instance, the Cray XMT [73] implements a multi-threaded processor to tolerate memory and network latency.

In GMT parallelism is identified through parallel loop constructs. The parallel loop construct enables creation of new tasks from iterations of loops over independent individual structure elements (e.g., parallel loops over all vertices or edges of a graph). Concurrency, through fine-grained software multithreading, allows tolerating the added latency for aggregating communication operations. We use the term task to identify a function pointer and an execution context inside GMT, while we use the term specialized thread (or, simply, thread) to identify either a worker, a helper or the communication server. Figure 5.4 shows a high-level overview of the GMT architecture. Each node executes an instance of GMT and the various instances communicate through commands. Different types of commands exist for GMT operations, such as global data read/write, synchronization and thread management.
Figure 5.4: Overview of the GMT architecture.
**WORKER:** executes the application code, partitioned in tasks, and generates requests, in the form of commands, directed towards both the local node and the remote nodes.

**HELPER:** manages global address space and synchronization, handles incoming requests and generates the related outgoing replies, in the form of commands.

**COMMUNICATION SERVER:** communication endpoint on the network, manages incoming and outgoing communication at the node level. Workers and helpers send commands to the communication server, which forwards them to the remote nodes.

A GMT node includes multiple workers and helpers, but only a single communication server. Each worker executes a set of GMT tasks. The worker switches among tasks (or contexts) every time it generates a blocking command that requires a remote memory operation.

### 5.2.2 Distributed memory implementation of Graph Search

This section describes our distributed dynamic graph data structure implemented on top of GMT. The literature is rich with research on storing and querying massive graphs on parallel or distributed systems. There are two primary approaches for storing a graph in a distributed memory. The first approach partitions the graph into multiple nodes on the cluster. Stanton et al. [74] provide an exhaustive overview of various approaches for partitioning a streaming graph on a distributed memory system. The partitioning-based
approaches rely on a master or a co-ordinator process that knows the mapping of a vertex in the graph (and its adjacency list) to a physical node in the cluster. Sun et al. [75] describe a distributed memory cloud based implementation of a graph database using the vertex partitioning approach. However, determining optimal partitioning of a graph is a NP-complete problem and even the sophisticated partitioning approaches are best described as heuristics. A second approach is use a distributed system such as a cluster of machines to provide an uniform memory space, i.e. a shared memory view. Graph databases such as GMT/SGEM [76] represent this approach.

Even then, there is scarce work on managing streaming or dynamic graphs in a large-scale system. STINGER [77] describes a data structure for representing dynamic complex networks in a shared memory space. Mondal and Deshpande [78] propose an in-memory, distributed graph data management system aimed at managing a large-scale dynamically changing graph, and supporting low-latency query processing over it. They propose aggressive replication of the nodes in the graph for supporting low-latency querying. Specifically, they develop a hybrid replication policy that monitors node read-write frequencies to dynamically decide what data to replicate, and whether to do eager or lazy replication. They also propose a clustering-based approach to amortize the costs of making these replication decisions and use a fairness criterion to dictate how replication decisions should be made.

Our distributed dynamic graph data structure is inspired by the idea of implementing Skip Graphs [79] on a circular buffer memory. Skip Graphs are in turn inspired by Skip
Lists. Skip lists [80] is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists. Figure 5.5 shows the data structures used in implementing the distributed streaming graph.

1. EDGE POOL The edge pool is an pre-allocated array of edges stored in distributed memory. The size of the streaming graph is not known in advance, but it is reasonable to expect that a window size (such as 1 billion edges) can be pre-determined using domain knowledge. The edge pool is divided into fixed size slots. Initially every vertex is assigned a slot to store its neighbors. As a neighbor list outgrows a slot, the next available slot in the memory pool is assigned to the corresponding vertex. The slot size is set to 16 by default. Intuitively the edge pool can be viewed as a circular buffer containing the slots. GMT supports multiple memory allocation policies. By default, the edge pool is allocated with GMT_ALLOC_PARTITION option, which allocates the array evenly among nodes starting from node 0.

Given that we do not know the size of the graph, or the size of adjacency lists in advance, dividing the edge pool into small slots and assigning them on the fly is a heuristic at best. Most of the real world graph datasets we expect to work with are scale-free, and hence it is hard to estimate a good size for the slots. Selecting large slots will be wasteful as we expect to find a large number of vertices with very small adjacency lists. On the other hand, using small slot size will cause large adjacency
lists to be fragmented. However, we prefer to conserve memory and compensate for
the fragmentation of adjacency lists by processing them in parallel.

2. **Multi-Level Adjacency Lists** Retrieving the list of neighbors for a vertex is
a core operation for any graph data structure. As mentioned above, we expect the
medium and large adjacency lists to be fragmented into multiple slots. To handle
such cases, we can “chain” multiple slots together. As an example, the last element
of a slot can be reserved to store the offset of the next used slot in the GMT array.
But this is not a perfect solution either. Successive slots store the neighbors of a
vertex can be stored on different nodes in the cluster. Any time we reach the end
of a slot, we will need to make a request to fetch the next slot across the network.
Therefore, sequentially accessing the neighbors by traversing the chain of slots will
be slow. Drawing inspiration from *Skip Graphs*, we introduce a multi-level adjacency
list structure to address this problem. Using the similar concepts of slots and circular
buffer, an intermediate layer is introduced that stores the offsets of the slots that
corresponds to each vertex. This is accomplished by allocating another array in the
GMT memory. The slot offsets for any particular vertex are stored in an interleaved
fashion on this array using a linked list approach.
**Figure 5.5:** Data structures for Distributed Streaming Graph.
5.3 Algorithm for Distributed Dynamic Graph Search

As described in the preceding chapters, performing the dynamic graph search requires maintaining the dynamic graph and the SJ-Tree. At the minimum, the SJ-Tree serves as a container for partial matches. The serial algorithm processes every incoming edge to the graph by following the steps described below.

1. Add the new edge to the graph.

2. Perform searches for all appropriate query subgraphs around the edge. Note, for “Lazy Search” we do not need to search for every query subgraph for each incoming edge.

3. Update the SJ-Tree by adding all the partial matches produced by above search step. Join small partial matches to produce larger partial matches whenever possible.

As is obvious, sequential processing of edges does expose a significant amount of parallelism to be exploited by multi-core or multi-node systems. This motivates us to process a large set of edges in parallel. In a streaming context, we can accumulate the set of incoming edges to the graph until it becomes substantial in size and then perform each of the above steps in parallel to take advantage of multiple processing units. The following subsections discuss the implementation of each of these three steps in a distributed system (GMT).
5.3.1 Parallel update of Distributed Graph

We begin with walking through the steps of a serial implementation for updating a graph with new edges. Assume that each vertex is assigned a list, or a dynamic array structure to store its adjacency list information. To add one more entry in that list, we first check to see if the size of the list has reached its allocated size. If the list has reached its allocated size, we first reallocate another list with twice its size, copy the members of the previous list to the newly allocated one, and finally release the memory for the original list. In the parallel context, multiple threads or tasks will attempt to append new entries to the list. Therefore, we need to implement this function in a thread-safe manner.

First, we need to maintain a counter for each vertex that stores the size of its current adjacency list. Each thread or task needs to update the counter using an atomic increment function call. When the list reaches its allocated size, one of the threads need to obtain a lock to gain exclusive access to the list and perform the operations described above. If multiple threads had attempted to update a full neighbor list, only one should succeed in performing the reallocation. It is critical to ensure that once the lock is released, all other threads use the updated value of the counters (storing the latest allocation size and number of neighbors) to proceed.
5.3.2 Parallel graph search

For this step, each thread receives an edge and performs a subgraph isomorphism for a query subgraph around it. Remember that each leaf in the SJ-Tree corresponds to a different query subgraph, and hence we need to search for each unique query subgraph around every new edge. From the perspective of a thread, any partial matches obtained from the subgraph isomorphism operation needs to be inserted into the appropriate partial match collection in the SJ-Tree.

The choice of the implementation platform determines the data structures used to implement a partial match. Conceptually, a partial match is implement as a set of two maps. The first map stores the correspondence between vertices in the query graph and their counterpart in the data graph. The second map stores the mapping of the query graph edges and data graph edges. Given that the parallel search is going to generate thousands or perhaps hundreds of thousands of partial matches, efficient implementation of the partial match data structure is a critical factor in the overall performance. It is always better to avoid C++ map data structures due to their high overhead of initialization. Also the query subgraphs are small and hence, we will be requesting a lot of small memory fragments from the runtime system. Therefore, using small fixed-size arrays are the best choice and the performance can be further improved by using a memory pool.

In the GMT implementation, each of the nodes in the SJ-Tree maintains a table of
these fixed size matches in global memory stored across multiple nodes in the cluster. Therefore, any partial match that is found by a search thread needs to be copied from the local memory to the global memory. It is critical to ensure that the insertion of the partial match into the tables performs a “deep copy” of the match contents. In other words, it should not be copying a reference to the local memory (as may be obtained by an allocated pointer), but use a function such as \texttt{memcpy} that copies the entire partial match from the local stack or heap to the GMT memory.

![Diagram](image_url)

\textbf{Figure 5.6:} Dataflow inside the SJ-Tree.

5.3.3 \textit{Parallel SJ-Tree update}

Joining partial matches was a core operation in the serial implementation. Every time we inserted a partial match into a leaf-level (in the SJ-Tree) partial match collection, we also computed a hash-value using the set of join vertices. Next, we searched for join
candidates by checking if any partial matches stored in the sibling node in the SJ-Tree had the same hash value. If a set of join candidates were discovered, we performed a subgraph join between each of the candidates and the newly inserted partial match. In a parallel implementation, we expect to find a large number of partial matches from the partial graph search phase, and all the SJ-Tree updates need to happen concurrently.

To solve this problem, we propose a parallel algorithm for updating the SJ-Tree (Algorithm 8). Also, we propose a parallel streaming hash-join algorithm for efficient join of two partial match collections. We will use Figure 5.6 to explain the steps of the algorithm. If there are $N_q$ leaves in the SJ-Tree, then we need to perform $N_q - 1$ joins that correspond to each of the internal nodes in the SJ-Tree. This is same as in the serial version of the algorithm, except that in the serial version we would insert a single match into one of the leaf nodes in the SJ-Tree and perform joins in progressively higher levels in the SJ-Tree until there are no candidates to join. In the parallel version, as Figure 5.6 shows, we are dealing with a set of new partial matches, denoted as $\delta_k$ for node $k$.

At the end of the parallel search phase, we have a $\delta_k$ for each of the leaves in the SJ-Tree. Then for each internal node $n$ in the tree, we will have two sets of new partial matches $\delta_{n-1}$ and $\delta_{n+1}$, corresponding to its left and right child. If $n$ is the parent of the left deep-most node in the SJ-Tree, then $\delta_{n-1}$ is the set of partial matches obtained from graph search, else it is a set of partial matches generated by parallel joins between partial matches corresponding to the children of $n - 1$. For either of these new partial match
sets ($\delta_{n-1}$ and $\delta_{n+1}$), we perform a join with the partial matches stored in the sibling node in the SJ-Tree. This process is described in line 8-17 in Algorithm 8. The join between partial match sets are implemented as parallel hash joins and is discussed in the following paragraph. Following Figure 5.6, observe that we join the new matches ($\delta_n$) for a node $n$ with the cumulative set of partial matches in its sibling node, obtained as a set union between partial matches from previous batches ($M_{n+2}$) and new matches from the current batch ($\delta_{n+2}$). The process is symmetrically repeated for node $n + 2$.

The parallel hash-join is implemented following the map-reduce based join algorithm presented by Afrati and Ulman in [81, 82]. The input and intermediate data are stored in key-value pairs and the computation proceeds in rounds. Each round is split into three consecutive phases: map, shuffle and reduce. In the map phase the input is processed one tuple at a time. This allows different tuples to be processed by different machines and creates an opportunity for massive parallelization. Each machine performing the map operation, also known as a mapper, emits a sequence of key-value pairs which are then passed on to the shuffle phase. This is the synchronization step. In this phase, the MapReduce infrastructure collects all of the tuples emitted by the mappers, aggregates the tuples with the same key together and sends them to the same physical machine. Finally each key, along with all the values associated with it, are processed together during the reduce phase. Here too, the operations on data with one key are independent of data with a different key and can be processed in parallel by different machines.
Algorithm 8 PARALLEL-SJ-TREE-UPDATE($T_{sj}, \Delta$)

1: $N_q =$NUM-LEAVES($T_{sj}$)

2: for all $i \in \text{range}(1, N_q)$ do

3: \hspace{1em} $n_{left} = 2(i - 1)$

4: \hspace{1em} $n_{right} = 2i$

5: \hspace{1em} if $n_{left} =$ROOT($T_{sj}$) then

6: \hspace{2em} REPORT-FULL-MATCH($\delta_{left}$)

7: \hspace{2em} return

8: \hspace{1em} $M_{left} =$GET-MATCH-TABLE($T_{sj}, n_{left}$)

9: \hspace{1em} $M_{right} =$GET-MATCH-TABLE($T_{sj}, n_{right}$)

10: if $i = 1$ then

11: \hspace{2em} $\delta_{left} = \Delta[0]$

12: \hspace{2em} $\delta_{right} = \Delta[i]$

13: \hspace{2em} $M'_{left} = M_{left} \cup \delta_{left}$

14: \hspace{2em} $M'_{right} = M_{right} \cup \delta_{right}$

15: \hspace{2em} SET-MATCH-TABLE($T_{sj}, n_{left}, M'_{left}$)

16: \hspace{2em} SET-MATCH-TABLE($T_{sj}, n_{left}, M'_{right}$)

17: \hspace{2em} $\delta_{left} = (M'_{left} \bowtie \delta_{right}) \cup \delta_{left} \bowtie M'_{right}$
Algorithm 9 PARALLEL-HASH-JOIN($M_1, M_2, M_{out}$)

1: $H = \text{INIT-KEY-VAL-COLLECTION()}$

2: for all $m \in M_1$ parallel do

3: \hspace{1em} $\text{INSERT}(H, \text{hash(join-pred}(m)), \text{TUPLE}(m, \text{true}))$

4: for all $m \in M_2$ parallel do

5: \hspace{1em} $\text{INSERT}(H, \text{hash(join-pred}(m)), \text{TUPLE}(m, \text{false}))$

6: $\text{GROUP-BY}(H, \text{groups})$

7: for all $g \in \text{groups}$ do

8: \hspace{1em} tuples = $\text{GET-RANGE}(H, g)$

9: \hspace{2em} $L_1 = \emptyset$

10: \hspace{2em} $L_2 = \emptyset$

11: for all $t \in \text{tuples}$ do

12: \hspace{3em} if $t(1) = \text{true}$ then

13: \hspace{4em} append($L_1, t$)

14: \hspace{3em} else

15: \hspace{4em} append($L_2, t$)

16: for all $m_l \in L_1$ do

17: \hspace{2em} for all $m_r \in L_2$ do

18: \hspace{3em} success = $\text{JOIN}(m_l, m_r, m_{out})$

19: \hspace{3em} if success = true then

20: \hspace{4em} $\text{INSERT}(M_{out}, m_{out})$
5.4 Complexity Analysis

We begin this section with a discussion of the time complexity of the parallel algorithm. Assume that we buffer $N_s$ edges from the stream over a unit of time. Each of the edges in the buffer are added to the graph and then searched in parallel by $P$ processors. The GMT runtime uses tasks as the unit of work, where each task in pinned to a core by default. In the subsequent discussion we use the terms processors and tasks interchangeably.

Following the complexity analysis of the sequential algorithms, the time complexity of searching for a single edge is $O(1)$, and the cost for searching a 2-edge subgraph is $O(d_{max})$, where $d_{max}$ is the maximum degree of a vertex in the data graph. Given that each processor receives $N_s/P$ edges, the time complexity of search by each processor can be expressed as,

$$T_{\text{search}} = \begin{cases} \frac{N_s}{P}O(1), & \text{for single-edge search} \\ \frac{N_s}{P}O(d_{max}), & \text{for 2-edge search} \end{cases}$$

Each processor produces a set of partial matches in its local memory at the end of the search phase. Next, all the partial matches are collected in a single global table, which is created by uniformly partitioning across nodes in the cluster. Now, each processor does not produce the same amount of matches. In fact, we expect the distribution of partial matches
to be rather skewed with tasks processing edges associated with high-degree vertices producing more matches. If $B$ is the intra-cluster bandwidth, then the time to build the search table can be expressed as follows. Note that $N_c$ is the number of nodes in the cluster, and a factor of 2 is added for the 2-edge search to reflect the higher size of partial matches.

$$T_{table} = \begin{cases} \frac{N_s}{N_c B} O(1), & \text{for single-edge search} \\ \frac{2N_s}{N_c B} O(d_{\text{max}}), & \text{for 2-edge search} \end{cases}$$

Consider $N_l$ as the number of leaves in the SJ-Tree, then the total cost for subgraph isomorphism and subsequent temporary match table construction is multiplied by a factor of $N_l$. In a binary SJ-Tree with $N_l$ leaves, we perform $(N_l - 1)$ joins for each of the internal nodes in the SJ-Tree. Thus, the total processing time can be expressed as,

$$T_p = T_{search} + T_{table} + T_{join} \quad (5.1)$$

Next, we consider the intermediate steps of the Hash-Join algorithm to determine the cost of a single join. Given two partial match collections of size $N_1$ and $N_2$, hashing in lines 2-5 take $O(N_h/P)$ time, where $N_h = N_1 + N_2$. GROUP-BY is internally implemented using a parallel merge sort following by a parallel scan of the data. For an array with size $n$, the cost of sequential merge sort is $n \log(n)$. We use a simple parallel implementation of merge sort [83] with complexity $O(n)$. Thus, the cost of the GROUP-BY operation is $O(N_h + N_h/P)$. Following the GROUP-BY, we again process each group in parallel. It is
likely that the size of the groups will be unbalanced. However, load balancing problems for joins caused by uneven distribution of group sizes is a well known problem in parallel and distributed computing [55, 81], and we reserve further optimization of the join algorithm for future work.

Each processor handles $N_g/P$ groups where $N_g$ is the number of groups generated. Also, each group is processed sequentially by the processor. The processing of each group is dominated by the loops in line 16-17. Therefore, the total cost for processing the groups is $\frac{N_g}{P} O(G^2_{\text{max}})$ where $G_{\text{max}}$ is the size of the largest group.

The type of the query decomposition can offer us some insights into estimating $N_g$. If two subgraphs are being joined by a single common vertex, then $N_g$ is equal to the number of vertices in the data graph. In general, for joining subgraphs using $k$-common vertices, $N_g = O(\binom{N}{k})$, where $N$ is the number of vertices in the data graph.

### 5.5 Experimental Results

Figure 5.7 - 5.9 shows the scaling results for three different datasets. The tests were run with 15 worker and 15 helper processes per node. As described earlier, the worker processes are responsible for any computation such as the subgraph search, and joins etc. and the helper processes are responsible for internode communication. Given that each process is associated with a core, a 16-node cluster configuration employs 240 cores (16 x
Figure 5.7: Scaling results from New York Times data.

15) for computation and 240 cores for communication. The runtime can be seen to improve with the increase of nodes. For New York Times, the slow improvement can be attributed to the relatively small size of the dataset. For the larger datasets (CAIDA and LSBench) good scalability is observed up to 16 nodes.

5.6 Summary

This chapter presents a distributed algorithm for dynamic graph search. The algorithm is implemented on a runtime system based on the Partitioned Global Address Space data model. The implementation was evaluated with three datasets on a 16 node cluster with 32 cores per node. Our experiments demonstrate good scaling up to 16 nodes for all
Figure 5.8: Scaling results from CAIDA internet backbone traffic data.

Figure 5.9: Scaling results from LSBench data.
the datasets. We also developed new data structures to manage a streaming graph and the
SJ-Tree in a distributed system. Exploring different partitioning strategies to minimize the
memory footprint of the distributed graph and SJ-Tree with small impact to query process-
ing time, and scaling up to streaming graphs with billion edges are primary goals for future
work.
CHAPTER 6. INCREMENTAL PATTERN DISCOVERY

“In solving a problem of this sort, the grand thing is to be able to reason backwards.”

*Sherlock Holmes, The Country of the Saints* (1887)

6.1 Motivation

A wide range of data sources from social media, online news and cyber-security are naturally modeled as graphs. However, many of these datasets are inherently heterogeneous. For example, they might represent interactions between similar or different types of entities. Thus, finding emerging patterns in heterogeneous datasets, and responding to them is critical for many domains. Discovering frequent subgraphs is a well established problem in graph mining [19, 31, 84]. Discovering frequent patterns in these graph streams on a continuous basis is an important capability for detecting emerging trends.

Different events have unique temporal patterns or signatures that emerge over a period of time. For example, consider a major sports event, or breaking of a major news event. Such an event emerges quickly in volume. After an initial spike in activity, the event typically lingers for a while before fading away. On the contrary, activities before a major festival or holiday season emerge distinctively but slowly. Several domains have examples
where certain patterns can be associated with either transient or slowly evolving behaviors. Such behavior has been researched exhaustively in the context of mining frequent patterns from streaming data [85].

Frequent subgraph mining poses several computational challenges. Several successful graph-mining algorithms operate in a transactional setting [19], where the updates to a graph are viewed as a set of transactions. There the goal of the graph miner is to detect frequently occurring subgraphs from a series of transactions.

However, tracking activities in a social network, online news, and network traffic, can be modeled better as a single instance of a large graph that is dynamic in nature. It is expensive to recompute frequent patterns periodically on databases that typically grow at the scales of hundreds of gigabytes everyday. In particular, the mining algorithms need to remember a synopsis of the frequent and infrequent subgraphs. Researchers have explored the batched approach towards mining frequent subgraphs where the updates to the large data graph are treated as batches. Each batch contains a set of new updates to the graph such as new edges. A frequent subgraph-mining algorithm is then run on the batch to update the set of frequent and infrequent patterns. Such approaches work well when the patterns in the data graph emerge closely over time (temporal locality). A principal contribution of our work is an algorithm that incrementally maintains the knowledge of frequent and infrequent subgraphs, and is robust to the time horizon over which the patterns emerge.

We are broadly interested in discovering emerging trends in the graph stream. Several
Figure 6.1: Example of a frequent subgraph appearing in a social media data stream.

interesting questions emerge on identifying and defining meaningful trends. As Figure 6.1 shows, there were millions of tweets posted following the tornado in Oklahoma, USA. In a graph-based model of the data (top portion of the figure), identifying a meaningful emerging pattern is a hard problem. While every article or tweet may contain a varying set of tags, there will be a small set of important words that define the context of a trend. As an example, the tags ‘tornado’ and ‘moore’ are sufficient to provide a basic understanding of a tornado and its location. Assuming that the data is represented as a $k$-partite graph where users, articles and tags are the independent sets of nodes, such a trend will emerge when the
graph becomes dense with 'tornado' and 'moore' gaining many common neighbors with type 'article'. This approach can be called a *strength-based* pattern discovery where we define the emergence of a pattern by the volume of activity surrounding a few nodes.

![Network diagram](image)

**Figure 6.2:** An example of frequent subgraphs in network traffic data where a group of clients (gray nodes) use the same set of servers (red nodes).

Another example relevant in this context is the emergence of patterns in computer network traffic. The illustration in Figure 6.2 shows the emergence of patterns in the network flow data from an enterprise network. While the nodes in grey represent consumers, the nodes in red represent service providers. An edge represents an exchange between two nodes and the notion of producer or consumer is derived from the context of an exchange. Different nodes accessing the services of a producer form patterns over a period of time, and reveal important information about the kind of activities that can be valuable to administrators protecting a network.
Social media and cyber-security are two major big data applications today. Deeper investigations into a wider range of case studies strongly suggest that a broad class of interesting graph patterns manifest as dense subgraphs in a large heterogeneous graph. Over time, a core set of nodes that represents the context of an event or a slow trend typically have their degree rising strongly. Enabling traditional frequent graph mining methods to be density-aware is an important problem for graph mining [86]. Typically the frequent subgraph mining activity is guided by a frequency support parameter. Introduction of a degree based threshold to extract denser, more specific subgraph patterns is another contribution of our work.

6.1.1 Problem Statement

Following is the definition of the frequent pattern mining problem. Given a graph $G$, find a set of subgraph patterns $G_P$ whose occurrences exceed a user specified threshold $\sigma$. For a dynamic graph both $G$ and $G_P$ evolve over time. Thus, we refine the problem definition as follows:

*Given a dynamic graph $G(t)$, the set of frequent patterns $P(t)$ at time $t$ and a set of edge additions to the graph over time $\Delta_t$, return $P(t + \Delta_t)$. \*
6.1.2 Contributions

1. We present an indexing framework, FS-MAP, that manages all the information related to frequent graph patterns and their instances in a dynamic graph.
2. We propose a bottom-up incremental algorithm for processing newly arriving edges in the graph and updating the frequent pattern set.
3. We introduce a degree-based filtering technique to avoid discovering “infrastructural patterns” in the data and find meaningful patterns from dense subgraphs.
4. We present experimental studies on real world datasets from diverse application domains such as online news (New York Times) and cyber-security (Internet Backbone Traffic) and demonstrate the effectiveness of our algorithm in discovering emerging patterns in the data.

6.2 FS-Map: A Graph Indexing Structure

We begin this section by introducing some key definitions. We use the term pattern to refer to labeled graphs (defined in chapter 2.1). Typically, patterns refer to small graphs that provide a general description of an interesting event. The size of the pattern graphs are of the same order as of query graphs discussed earlier in the thesis.

**Definition 1.** Frequent Pattern: A pattern is frequent if it occurs (including overlaps) at least $\sigma$ fraction in the dataset, where $\sigma$ is user defined. A frequent pattern can
also be defined in terms of absolute frequency, where it needs to occur at least $k$ times in the data in order to be deemed frequent. The frequency of the pattern is also referred to as support.

**DEFINITION 2. CLOSED FREQUENT PATTERN:** A closed frequent pattern is defined as a frequent pattern that has no immediate super-graph with the same support.

### 6.2.1 Motivation for Dynamic Graphs

The typical approach to frequent subgraph discovery is as follows. We begin with finding frequent small subgraphs (such as single-edge subgraphs). Given a set of initial frequent patterns, we try to expand it by trying to discover frequent subgraphs with one additional edge. This process keeps repeating until we can not expand the candidate set any further. Two prominent algorithms for mining large graphs, SUBDUE [84] and HSiGraM [31] adapt this approach. Internally, the process described above resolves into three main computational tasks: Counting patterns, generation of candidates, and tracking the patterns.

In the following discussion we describe the unique challenges introduced by a dynamic graph to each of these tasks.

1. **COUNT PATTERNS** Given a pattern $P$, knowing if it is frequent requires counting its occurrences. Given a data graph $G$ and a query graph $P$, we need to enumerate all the subgraphs of $G$ that are isomorphic with $P$. The inherent hardness of subgraph
isomorphism is a primary challenge in the frequent subgraph mining process.

For a dynamic graph, the support for each pattern potentially changes over time with the evolution of the graph. However, searching the entire graph periodically to enumerate the instances of a pattern is not an efficient option. Rather, we take a different approach. Given a new edge in the graph, we try to discover candidate patterns using algorithms described later. Given any pattern, we compute its canonical string-based representation and use a hash-table to track the counts of each pattern.

2. **Candidate Generation** Given a pattern $P$ and one of its instances $I$, a subgraph of $G_d$, we can generate a new candidate pattern $P'$ by adding a neighboring edge $e$ from $G_d$ to $I$. The term *neighboring* refers to the requirement that one of the endpoint vertices of $e$ is present in $I$. Given the new candidate pattern $P'$, we need to count its occurrences and determine if it is frequent.

We address this in the dynamic setting as follows: given a new edge $e$ into the data graph $G_d$, we try to find an instance of a frequent pattern $(g_1)$ that includes $e$. If successful, we use the join-index (introduced later) to discover “nearby” instances of other frequent patterns that can be joined to produce candidates for larger patterns.

3. **Track Patterns** Given a pattern $P$, we also need to track all its occurrences in the data graph $G_d$. Tracking all instances of a pattern can be expensive. This is especially true for small subgraphs. If $P''$, a super-graph of $P$ is found to be frequent, then we
can discard the instances of $P$ by leveraging the property of closed patterns.

**Figure 6.3:** Directed graph showing dependencies between patterns. Solid and dotted rectangles are used to indicate frequent and infrequent subgraphs respectively.

This section describes FS-MAP, the indexing structure used to maintain the information about frequent subgraphs as the underlying graph changes over time. We describe the FS-MAP as a 3-tuple: $(G_{pd}, I_{sub}, I_{join})$, and describe each component below. These three components are also shown in Figure 6.3.

1. **Pattern Dependency Graph** ($G_{pd}$): We track all patterns that are discovered in the dynamic graph. The dependencies emerge as a hierarchical structure in which larger patterns are formed from smaller patterns. For every pattern we also track a set of attributes such as maintaining its support and frequent or infrequent status.

2. **Subgraph Index** ($I_{sub}$): For each pattern we track the occurrence of all the instances in the data graph $G_d$. The set of patterns can grow very large and, therefore,
we only track the instances of closed frequent subgraphs for efficiency considerations.

3. JOIN INDEX ($I_{\text{join}}$): Finally, for every instance subgraph in $G_d$ we also need to maintain a set of join candidates. Given a subgraph $g_d$, the join candidates are the subset of its nodes that it wishes to “publish” for joining with other subgraphs.

### 6.2.2 Pattern Dependency Graph ($G_{pd}$)

The pattern dependency graph is implemented as a directed graph. Assume $\psi$ is the set of all patterns. Thus, a function $f_{pd}: V(G_{pd}) \rightarrow \psi$ maps each node of $G_{pd}$ to a pattern graph in $\psi$. Also, edges of $G_{pd}$ indicate the formation of larger patterns from smaller pattern graphs.

**PROPERTY** For any edge $e = (s, t)$ in $G_{pd}$, if $g_s = f_{pd}(G_{pd}, e)$ and $g_t = f_{pd}(G_{pd}, t)$ are two pattern graphs then $g_s \subset g_t$. We use a hashtable to implement the function $f(G_{pd}, v)$.

We also track a set of attributes for each node in $G_{pd}$. They include:

1. Support information
2. Frequent/infrequent status
3. Pair of parent nodes from which it was created
4. Set of children nodes representing its supergraphs
The reader may observe that $G_{pd}$ is essentially a forest of trees and could be possibly disconnected. Leaves of the forest represent smallest frequent subgraphs and roots of the trees in the forest represent largest frequent subgraphs.

### 6.2.3 Subgraph Index ($I_{sub}$)

The subgraph index tracks the occurrence of the instances of patterns in the actual data graph $G_d$. $I_{sub}$ is an associative collection which maps every pattern graph in $\psi$ or equivalently each node of $G_{pd}$ to a set of isomorphic subgraphs of $G_d$.

**PROPERTY** Given a pattern graph $g_p \in \psi$ (the set of all patterns), $I_{sub}$ provides a mapping $f_{sub} : g_p \rightarrow G'_d$ such that, $(\forall g \in G'_d | g \subset G_d \land isomorphic(g, g_p))$.

Each subgraph of $G_d$, which is represented as a list of edges, in assigned a unique integer id. This is done by computing a string based signature for every subgraph, and hashing the string to an integer value. Given that $f_{pd}$ maps every pattern to an unique id, the implementation of $f_{sub}$ is reduced to mapping an integer to a set of integers.

### 6.2.4 Join Index ($I_{join}$)

Before proceeding further we would like to remind the reader about using subgraph join as a fundamental operation in our algorithm. Our approach for discovering frequent
subgraphs is based on joining the instances of smaller and frequent subgraphs to discover larger and frequent subgraphs. Remember from definition 2 that we need to specify a predicate function in order to join two subgraphs. This brings up the issue of identifying the join predicate, the basis on which two subgraphs can be combined to produce a larger subgraph.

We adopt the simplest option of mapping a subgraph to the set of corresponding nodes. Two subgraphs can be joined if they share a common node. Next, we look at how this simple predicate function is computed. Given that the underlying graph is dynamic, we may discover two subgraphs $g_1$ and $g_2$ at different points of time. Assume $g_1$ arrived before $g_2$, i.e., all edges in the subgraph $g_1$ arrived before all the edges in subgraph $g_2$. Next, we need to determine if a join can be performed between $g_1$ and $g_2$. In other words, we need to find if $g_1$ and $g_2$ share any common set of nodes. We need to provide an indexing capability similar to that of inverted index in information retrieval [87] to support such queries. Thus, given any node in the data graph, we maintain a list of the identifiers of all subgraphs containing that node.

When we discover a subgraph $g_d$ to be an instance of $g_e$, a pattern graph, the procedure for updating $G_{pd}$ and $I_G$ is described later in the UPDATE-FS-MAP().
6.3 Incremental Processing Algorithm

Following the definition of the data structures, we next turn our attention to describing the incremental processing algorithm. However, before proceeding further we introduce the concept of a primitive pattern.

PRIMITIVE PATTERN We define a “primitive pattern” as a $k$-edge pattern graph. We use the term “primitive” to refer to the fact that these patterns serve as building blocks in our pattern discovery process. Any pattern we report will be composed of one or multiple primitives. We constantly search the graph stream for primitives. Therefore, a primitive pattern should be fast to search for, such as single edge subgraphs or dyads. Fast computation of the distribution of primitive patterns is another strongly desirable property, although we do not explore its applications here. All results presented here are based on single edge primitive patterns.

The incremental algorithm performs the following tasks as shown in Figure 6.4.

1. Search each edge in the graph stream to discover instances of primitives patterns and generate candidate patterns.

2. Track the status of the matches with the primitives as they transition from frequent to infrequent status and vice versa.
3. Insert instances of frequent patterns into the FS-MAP to enable the discovery of larger frequent subgraphs.

4. When a frequent primitive turns infrequent, update the FS-MAP to reflect the transition of all its supergraphs into infrequent status.

The first two steps are about guiding the discovery of patterns by finding small “interesting” patterns while the third stage focuses on efficient aggregation of smaller patterns into larger ones. The final step is focused on incremental maintenance of the FS-MAP data structure as patterns transition from frequent to non-frequent status. Algorithm 10 provides a high-level description of the incremental processing algorithm.
Algorithm 10 INCR-MINER($e, G_d, G_{pd}, I_{join}, I_{sub})$

1: if FILTER-CANDIDATE($G_d, e$) then

2: \[ g_s = \text{MAP-TO-PATTERN}(e) \]

3: \[ \text{UPDATE-FS-MAP}(g_s, G_{pd}, I_{join}, I_{sub}) \]

6.3.1 Primitive pattern generation

Given a new edge in the graph, we check to see if it can be used to seed a new primitive pattern or it matches any of the discovered primitive patterns. In addition to the frequent subgraph support parameter, we introduce a degree threshold as a controlling parameter in the pattern generation process.

EXAMPLE: Suppose we get a new edge in the graph: \[ \text{src:}`\text{ip:123.345.678.901}`, \text{dst:}`\text{ip:facebook.com}`, \text{edge-type:}http\], i.e., the source node has type “ip” and label “123.345.678.901”. If the edge type ”http” is a frequent edge in the graph, we can extract the frequent pattern \{src:ip, dst:ip, edge-type:http\} from such data, which describes two IP addresses communicating over HTTP protocol. However, such patterns can lead us to find extremely generic, non-interesting patterns that reflect the schema of the data and referred to as “infrastructural patterns”. Assuming that we get many edges where the source node represents a random IP address and the destination node is always “ip:facebook.com”, a more specific pattern such as \{src:ip, dst:ip:facebook.com, edge-type:http\} provides a
much better representation of the context. This process of generating a vertex signature is described in Algorithm 11. Algorithm 12 uses Algorithm 11, and generates a string signature for a subgraph of the data graph by iterating over its edges.

**Algorithm 11** GET-SIGNATURE($G_d, v$)

1: if DEGREE($G_d, v$) > DEGREE-THRESHOLD then
2: RETURN “TYPE($G_d, v$) : LABEL($G_d, v$)”
3: else
4: RETURN TYPE($G_d, v$)

**Algorithm 12** MAP-PATTERN($G_d, g_s$)

1: str = “”
2: for all $e \in E(g_s)$ do
3: APPEND(str, GET-SIGNATURE(source(e)))
4: APPEND(str, GET-EDGE-TYPE(e))
5: APPEND(str, GET-SIGNATURE(dest(e)))
6: RETURN str

### 6.3.2 Update and Maintenance of FS-MAP

Assume that we found a subgraph $g$ which is an instance of a frequent primitive $g_p$. Given $g$, we look up the join index $I_{join}$ to find instances of all other frequent patterns that
are neighboring to $g$.

**EXAMPLE:** Assume that we are mining a dynamic graph that is always updated with new articles, its authors and topics. We discover that “author, topic:databases” is a frequent pattern, where nodes of type “author” and nodes of type “topic” and label “databases” are connected with an edge. Then at some point we start tracking a similar pattern as “author, topic:machine-learning”, although it is not frequent yet. Once the second pattern turns frequent, whenever we discover neighboring instances of these two patterns, we join them to produce a new, larger candidate pattern “author, databases, machine-learning”.

However, the set of possible join candidates can be large. Given $g$ and another join candidate $g'$, the number of candidates will vary depending on the number of common nodes and edges. We introduce a set of rules to minimize the number of redundant candidates.

**RULE 1** Given two subgraphs $g_1$ and $g_2$ that are instances of two frequent patterns $g_1^p$ and $g_2^p$, we will perform a join only if both $g_1$ and $g_2$ has some unique nodes. This rule ensures that we are not joining two graphs where one is the subgraph of another.

**RULE 2** We want to minimize the number of all $k$-subgraphs such that there is minimal overlap between them. If a candidate pattern $g_i$ is joined with $g_j$ to produce a bigger frequent pattern $g_k$, then do not join with any pattern $g_k$ that is a subgraph of $g_j$. In other words, we are joining only with maximal patterns.

Algorithm 13 describes this entire process step by step. Given any subgraph $g_s$ that
we wish to update the FS-MAP, we first compute its canonical signature (line 1), and update its support in the data graph. We use a hash-table to serve as a counter for the support information. The string representation of the canonical signature is inserted as key into the hash table. When the pattern turns frequent, we assign it a unique ID (line 5). Next, we use this id to update $I_{\text{join}}$ and $I_{\text{sub}}$. $I_{\text{sub}}$ is implemented as a hash table. $I_{\text{join}}$ is a multi-map where node ids from $G_d$ are keys, and the list of subgraph ids are maintained as values for each key. We also query $I_{\text{join}}$ to discover candidate subgraphs to join (line 9). We find all the unique subgraphs that $g_s$ can be joined with and perform a join obeying the rules mentioned above. If the join is successful we update the dependency graph with the data about the input and output to the join process, and finally repeat the process recursively to find larger subgraph patterns.

6.4 Experimental Results

We present experimental analysis on two real-world datasets: 1) New York Times \(^1\) and 2) CAIDA Internet Backbone Traffic data \(^1\). The experiments were performed on a Mac OS X system with 2.4 GHz Intel Core i7 processors with 16GB memory. The code was

\(^1\)http://data.nytimes.com

\(^1\)http://www.caida.org
**Algorithm 13** UPDATE-FS-MAP($g_s, G_{pd}, I_{join}, I_{sub})$

1. $s = \text{MAP-PATTERN}(G_d, g_s)$

2. \{ support, transition \} = \text{UPDATE-SUPPORT}(s)$

3. if support < FREQ-SUBGRAPH-SUPPORT then

4. if transition = true then

5. \text{PRUNE-FS-MAP}(G_{pd}, I_{join}, I_{sub})

6. RETURN

7. id = \text{NEXT-SUBGRAPH-ID}()

8. $I_{sub}[id] = g_s$

9. candidate-list = $\emptyset$

10. for all $v \in V(g_s)$ do

11. $G_{list} = \text{FIND}(I_{join}, v)$

12. APPEND(candidate-list, $G_{list}$)

13. \text{UPDATE-MULTIMAP}(I_{join}, v, id)

14. $L_g = \text{UNIQUES}$candidate-list)

15. for all $id \in L_g$ do

16. $g_c = I_{sub}[id]$

17. $g_{out} = \text{JOIN}(g_s, g_c)$

18. if $g_{out} \neq NULL$ then

19. \text{UPDATE-DEPENDENCY}(G_{pd}, g_s, g_c, g_{out})

20. \text{UPDATE-FS-MAP}((g_{out}, G_{pd}, I_{join}, I_{sub})
6.4.1 New York Times

We begin with presenting experimental results from the New York Times dataset described earlier (section 3.3). Figure 6.5 shows one of the most frequent patterns discovered from this news data. Observe that the pattern is more specific than just a structural pattern. Some nodes in the pattern graph have labels (e.g., “person::Obama, Barack”), which suggests that we found a dense, frequent subgraph around the nodes with specified labels. Figure 6.6 shows the variation in runtime as a function of the frequent subgraph support threshold. The Y-axis shows the time to process the entire dataset in logarithmic scale. Observe that the run time decreases exponentially as the support is increased, which is in line with expectations.

Figure 6.5: A sample pattern discovered from New York Times data.
**Figure 6.6:** Total runtime as a function of varying frequent subgraph support (NYT).

**Figure 6.7:** A sample pattern discovered from internet backbone traffic data where an IP address (one on right) is flooded with requests from another IP address using different source ports.
6.4.2 Cyber-Security

Each record in a network traffic dataset corresponds to a communication between two IP addresses. Each record also contains attributes such as source and destination ports, the protocol used, number of packets in the flow, and the length of the communication. Each IP address is treated as a node in the graph and each communication between two IP addresses is represented as an edge. Given the nature of the data, all nodes (representing IP addresses) are homogeneous. Every edge is typed by the communication protocol it represents (e.g. TCP, UDP, ICMP etc.). We used a sample of the internet backbone traffic between Chicago and Seattle (available from www.caida.org) for testing. Figure 6.7 shows one of the most frequent pattern in this dataset. Similar to the pattern reported from the news data, this top frequent pattern also finds a dense subgraph with many edges between two IP addresses.

Figure 6.8: Total runtime as a function of varying frequent subgraph support (network traffic).
Specifically, it reveals an attack pattern where one host is flooding another by sending requests from multiple source ports. Figure 6.8 shows the time to process 1M edges in the network traffic data as a function of varying support.

Both Figure 6.6 and 6.8 show the variation in processing time as a function of subgraph support. However, it is easy to note the wide difference in the characteristics and the support levels chosen for experiments. Our objective here is to point the reader to the high-level trend. The datasets vary widely in nature. Online news is a much more slowly evolving data stream when compared to the internet backbone traffic. When a major event happens in the real world, it often dominates the news for the next few days. Thus, we have large dense subgraphs forming over a long period of time in the data. On the contrary, network traffic data is extremely transient, and many significant patterns are very localized in the graph. Automatic selection of optimal values for frequent subgraph support and degree threshold remains a major topic for future exploration.

Figure 6.9 shows the growth in the number of subgraph patterns tracked as the graph grows over time. Similarly, Figure 6.10 shows the number of frequent subgraphs, or instances of patterns tracked with the graph’s evolution. Together, the number of tracked patterns and their instances provide a good approximation of the amount of memory consumed by the incremental algorithm. Figure 6.9 and 6.10 show how an intelligent selection of degree threshold (as specified in Algorithm 13) can lead to significant speedup.
Figure 6.9: Total number of frequent and infrequent subgraph patterns tracked during the course of the dynamic graph evolution.

6.5 Related Work

Previous research efforts have mostly been concentrated on developing frequent subgraph mining algorithms for static graphs. Frequent subgraph discovery algorithms can be categorized into either complete or heuristic discovery algorithms. Complete algorithms like SIGRAM [31] find all the frequent subgraphs that appear no less than a user specified threshold value. Heuristic algorithms like SUBDUE [84] discover only a subset of all frequent subgraphs by finding maximally compressing subgraphs. SIGRAM developed by Kuramochi et al. is a complete discovery algorithm which finds frequent subgraphs from a large sparse graph. The first frequent substructure based algorithm was designed by Inokuchi et al. [88] and was inspired by the Apriori algorithm for frequent itemset mining.
Figure 6.10: Total number of frequent pattern instances tracked during the course of the dynamic graph evolution.

[18]. Algorithms like gSpan [19], MOFA [89], FFSM [90], SPIN [91] and GASTON [92] were developed to avoid the overheads of the candidate generation approach. They use a pattern growth technique which attempts to grow the pattern from a single pattern directly.

Subgraph mining for dynamic graphs has received attention in the past few years only. Bifet et al. [93] compared several sliding window approaches to mining streaming graph transactions for closed frequent subgraphs using a coreset of closed frequent subgraphs discovered in the past. Aggarwal et al. [86] proposes two algorithms for finding dense subgraphs in large graphs with streaming updates. However they assume that the updates to the graph come in the form of edge disjoint subgraphs. Wackersreuther et al. [94] proposed a method for finding frequent subgraphs in dynamic networks, where a dynamic network is basically the union of time based snapshots of a dynamic graph. Our work is
distinguished from all these works as we attempt to find subgraphs in a single large graph which receives continuous updates.

6.6 Summary

The primary contribution of this chapter is the development of an algorithm for incremental discovery of frequent patterns in a dynamic graph. We presented a new indexing framework, FS-MAP, for efficient management of frequent patterns and their instances in a dynamic graph. The novelty lies in a bottom-up approach in which we discover larger patterns from single-edge updates to a dynamic graph. An artifact of the pattern mining algorithm is the dependency graph in FS-MAP, which captures the statistics in which smaller patterns join to form larger patterns. Thus, the dependency graph stands to provide critical insights on the evolution of the graph, which can be exploited for several applications such as optimal pattern search techniques in a dynamic graph and proximity pattern mining, where understanding the co-occurrences of various subgraph patterns is important.

Another contribution of our work is the introduction of degree based filtering for discovering frequent patterns. Data streams such as social media, online news and cyber-traffic are naturally modeled as $k$-partite heterogeneous graphs, and frequent patterns typically manifest in them as dense subgraphs for such applications. Introduction of degree-based constraints provides a very simple, but powerful tool in the pattern discovery process. Fi-
nally, our approach unveils opportunities for adaptive processing where the threshold may be obtained from sampling the graph stream.
CHAPTER 7. APPLICATIONS STUDIES

“It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.” Sherlock Holmes, A Scandal in Bohemia (1891)

Applications have played a critical role in shaping this thesis. Detecting emerging subgraph patterns in cyber network traffic has been the primary driver for our research [60, 95]. We have also explored online news and social networks to test and verify our hypotheses. Going beyond dynamic graphs, we also started investigating areas such as multi-cloud graph databases to apply the query optimizations techniques developed as part of this thesis. The goal of this chapter is to discuss a number of important problems (or “use cases”) from each of these areas and demonstrate how the research on dynamic graph search can make important contribution to each of them.

7.1 Cyber-Security

The number and sophistication of cyberattacks on industries and governments have grown dramatically in recent years. To counter this movement, new advanced tools and techniques are needed to detect cyberattacks in their early stages such that defensive ac-
tions may be taken to avert or mitigate potential damage. From a cybersecurity analysis perspective, detecting cyberattacks may be cast as a problem of identifying patterns in computer network traffic. Logically and intuitively, these patterns take on the form of a directed graph that conveys how an attack or intrusion propagates through the computers of a network. In the cybersecurity context, some limited research has been conducted on using directed graphs to model cyberattack patterns [96, 97]. Distributed event monitoring and minimizing the amount of false positives are the major challenges for these systems. As an example, a DDoS attack is often hard to separate from a flash crowd event. Ganguly et al. [98] present a streaming algorithm to monitor the distinct source frequencies to distinguish between benign and malicious activities. Venkatraman et al. [99] present an algorithm to detect sources that connect to a large number of distinct connections in a streaming setting with specified accuracy and memory requirements.

Identifying cyberattack graph patterns from within a larger graph of a computer network is a classic subgraph isomorphism problem. Another complexity is the requirement to conduct partial matching of the cyberattack graph pattern, such that one can detect the pattern before it is fully instantiated. In addition, the larger computer network graph would be dynamic or ever-changing with message patterns and host machines statuses constantly transitioning over time.
7.1.1 Example Queries

To enable subgraph pattern matching, various types of cyberattacks may be depicted as temporal, directed multi-graphs. In most cases, the graphical patterns of cyberattacks have repeating internal structures. To be a useful query graph, the patterns need to be simple enough to easily comprehend while capturing enough of the repeating structure to discriminate the cyberattack from normal or usual computer network traffic. We present a few illustrative cyberattack graph queries in Figure 7.1 that are further described below.

**Figure 7.1:** Cyberattack graph queries for a) Witty worm, b) Smurf DDoS, c) Fraggle DDoS, and d) DNS Amplifications DDoS cyberattack.

**Witty Worm** The Witty worm is an Internet worm that targets a buffer overflow vulnerability in Internet Security Systems products. It is known to attack port 4000 of Win-
machines with packets of sizes between 796 and 1,307 bytes. As shown in Fig. 1a, the associated query graph looks to detect infected machines that are sending out packets with Witty worm characteristics to at least five other machines and a path of at least three machines that have been infected in chronological order. In the diagram, the chronological order of the messages is indicated by edge color transitioning from light to dark blue.

**SMURF DISTRIBUTED DENIAL-OF-SERVICE (DDoS)** DDoS attacks typically involve a hacker sending messages to intermediate host machines with the spoofed source address of the victim machine. In the case of the Smurf DDoS attack of Fig. 1b, the hacker sends an “ICMP Echo Request” message to a broadcast IP address that appears to come from the victim. A router will pick up the message and broadcast it to intermediate host machines. In response, the intermediate host machines then floods the victim machine with “ICMP Echo Reply” messages. We consider flooding from at least three intermediate host machines as sufficient evidence of the Smurf DDoS attack occurring.

**FRAGGLE DDoS** As shown in Figure 7.1c, a Fraggle DDoS attack is the UDP version of a Smurf DDoS attack and has a similar graphical structure. In the Fraggle attack, a “UDP Echo Request” message is broadcast to port 19 of intermediate host machines, which in turn, sends the “UDP Echo Response” message to port 7 of the victim machine. The UDP version may be devastating because it may initiate a repetitive echo request-response loop between the intermediate host machines and the victim.

**DOMAIN NAME SYSTEM (DNS) AMPLIFICATION** DDoS In a DNS amplification
DDoS attack, zombies or agents generate a large number of DNS queries with a spoofed source address and send these queries to various DNS servers. As shown in Figure 7.1d, the DNS requests appear to come from the victim machine. The DNS servers respond with three different possible types of messages back to the victim machine, which are the “DNS Standard Query Response,” “ICMP Destination Unreachable,” and “Fragmented IP Address” messages. Such attacks are particularly effective because DNS response packets may be significantly larger in size in comparison to the initiating DNS request packets.

We are optimistic about real-time pattern matching in cyber data being a killer application for dynamic graph search. However, a number of domain specific issues need to be addressed in addition to the general challenges such as scaling and adaptive processing. Network traffic datasets are a classic example of multi-graphs where each vertex pair can have multiple edges between them. But introduction of multi-edges can affect query performance drastically, especially when the search is performed around high degree vertices. Usually, edge level aggregation is used to mitigate this problem, but too much aggregation can often lead a coarsened graph where the events are not visible. In short, finding the right combination of graph aggregation and search strategies will be critical to establish dynamic graph search as an effective tool in cyber defense.
7.2 Social

Today social networks have become a fixture of our daily lives. The number of social networks targeting different application areas have risen and so have the level of user adoption. Collectively, they result in information overload where we are drowned with too much information from our social stream. We are not particularly interested in every update we receive, and important updates are often lost in the noise. Thus, personalized stream queries, or registering patterns that search for events matched to an individual’s interest appears as a natural progression. As an example, Figure 7.2 shows a fictitious social query that one may use to search for when multiple friends are meeting in a nearby location.

Earlier, we had extensively used the LSBench benchmark for testing multiple query decomposition strategies. LSBench [100] is a RDF benchmark for logical reasoning on streaming data [101]. Figure 7.3 shows the logical schema used by the data generator. Figure 7.4 and 7.5 are two complex queries that are part of the benchmark that are particularly relevant to our work. The role of graph homomorphism as a core operation in graph-based reasoning is well established [102]. Given that both graph homomorphism and subgraph isomorphism share similar computational characteristics, we foresee strong application of our work to provide efficient implementation of SPARQL variants such as

\[\text{http://code.google.com/p/lsbench}\]
C-SPARQL [103].

![Figure 7.2: Example of a streaming query for a social stream with geo-location data.](image)

### 7.3 Cloud Databases

Oftentimes, answering the most interesting questions about data at scale require fusing data present in multiple datasets. Querying, analyzing, and fusing data at scale generally requires the use of large data clouds residing in data centers. However, when the datasets of interest exist in multiple different clouds, potentially under the control of different organizations, this affords a number of additional challenges. Ideally, all the datasets would be moved to a single cloud, but this may not be technically feasible. Data centers may be physically distant or rely on different cloud technologies. Likewise, administrative costs may be prohibitive for supporting such complex architectures. Finally, there may be policy issues acting as barriers to data collocation and integration, such as privacy or data sharing concerns.
Figure 7.3: Logical schema of the LSBench stream data generator.

Figure 7.4: Graph representation of SPARQL query 6 from LSBench benchmark.
Federated databases and distributed queries have been a topic of research for many years. Traditional query optimization techniques, such as the Magic Sets algorithm and others ([104–107]) have been developed for distributed query optimization and distributed query execution. Query algorithms across distributed graph stores generally have been focused on a single cloud, and assume that we can arrange the graph data freely across the data store to enable fast, efficient querying. In contrast, multi-cloud applications demand support for a single query targeting multiple graphs residing in multiple clouds. As an example, consider the query shown in Figure 7.6 that targets two graph databases stored in two different clouds.

For such applications, an obvious approach is to partition the query graph into edge disjoint subgraphs where each disjoint query subgraph corresponds to an unique cloud.

Figure 7.5: Graph representation of SPARQL query 7 from LSBench benchmark.
Figure 7.6: Example of a graph query targeting multiple clouds. The colors on the edges indicate the cloud database where the relation is stored.

database. In that case, a coordinator process can delegate the individual query execution to respective clouds and merge the results returned by them. However, that may not be efficient if the selective subqueries span across multiple databases. One may foresee producing fine-grained decompositions such as ones based on single relations, but then determining a good join order becomes critical. Each cloud database may be driven by different database technologies and, therefore, the coordinator, or the query planning process will need to rely upon basic graph statistics. We believe our query decomposition approach that relies on the distribution of primitives will be relevant for such applications. For example, if individual clouds could report statistics related to partial results, e.g., number of results for each subquery component or distribution of node ids over query variables, they can be used by the coordinator to determine the join order of partial results using the same SJ-Tree
construction algorithm. As an example, Figure 7.7 and 7.8 shows two potential SJ-Tree decompositions for the query in Figure 7.6.

![Figure 7.7: Edge based decomposition for query in Figure 7.6](image)

7.4 Summary

We covered three different application areas in this chapter, namely real time querying of cyber-security, social media data and multi-cloud database querying. We presented a number of complex queries for the first two application areas. The cyber-security queries motivate the need for supporting temporal dependencies between edges in a query graph, such as specifying that a set of edges in the query graph need to occur before the rest. They also underscore the need for creative user interfaces to describe the query graphs, as it will
be difficult for an end user to draw a large query graph with hundreds of edges. Auto-
recommendation of query constraints or structures to help the user compose a selective
(to aid with performance) and sufficiently descriptive query will be a critical area as well.
While the social queries are simpler than the cyber-queries, the challenge will be to support
concurrent registration and execution of hundreds of thousands or millions of such queries
in a social network. Creative user interfaces are more important for social networks; we
can not expect social network users to write graph queries using a language. Minimizing,
or zeroing the technical background requirement will be critical to bringing the dynamic
graph search to the masses. Finally, the work on cloud databases point to a different set of
challenges. As increasing number of databases migrate to the cloud, optimizing the query
execution will be critical for minimizing data movement and reducing energy footprint.
CHAPTER 8. CONCLUSIONS AND FUTURE WORK

“What you do in this world is a matter of no consequence. The question is, what can you make people believe that you have done?” Sherlock Holmes, A Study in Scarlet (1887)

This thesis presents an investigation of Dynamic Graph Search. Dynamic graph search is a nascent research area, even though the broader problems of subgraph isomorphism or subgraph matching has been studied extensively. Our goal has been to understand unique drivers that distinguish the dynamic graph search problem from its traditional counterpart of searching a static graph (or ad-hoc queries), and identify the theoretical areas that need to be developed to satisfy the drivers. The perfect example of a dynamic graph query is “tell me when X happens”. X can be a pattern describing a cyber-attack, or a pattern of an event unfolding in a social network. Contrast this with the traditional ad-hoc queries that say “find me all instances of X”.

Dynamic graph search challenges a number of fundamental assumptions that are critical to design of a graph database. First, in a streaming setting, we can not afford to run sophisticated indexing algorithms that enable fast search for ad-hoc queries. Second, constant evolution of the graph presents unique challenges for scalability. Graph partitioning is a hard problem, and it impacts the search performance in a distributed system. Understanding the evolution in terms of spatial and temporal features and adapting with the change is
a major challenge. Third, the stream processing constraints dictate that we can hold only limited information in memory. However, we can “learn” from the stream and apply that knowledge for efficient query answering. Identification of the stream features that are relevant to the search performance stays a widely open area to explore. This thesis attempts to address each of these questions as stated in our contributions below. We hope that they provide an initial foundation to build upon, and to iterate and perfect. We draw inspiration from how the field of continuous queries has evolved to address the limitations of traditional relational databases for streaming applications such as high frequency trading and sensor networks, and envision the same happening in the realm of graph search to enable novel applications such as cyber security and social media.

The following are the major contributions of this thesis.

1. We developed a new subgraph isomorphism algorithm for dynamic graph search. Our experiments on representative data sources (New York Times, DBLP) demonstrates multiple orders of magnitude improvement over the state of the art [1].

2. We developed a new data structure named Subgraph Join Tree (SJ-Tree) that represents the execution strategy for a query. We also developed a set of algorithms for efficient collection of graph stream statistics, and using the statistics for automatic generation of the SJ-Tree for any given query graph.

3. We developed a multi-level data structure for efficient storage and querying of a
distributed streaming graph, accompanied by a variant of the dynamic graph search algorithm for distributed system implementation.

4. We developed a new graph mining algorithm that discovers emerging patterns in a stream by leveraging the principles of efficient incremental search and subgraph join operations.

8.1 Future Work

The previous chapter on applications studies points out a number of domain specific challenges that are worth addressing. Following is a list of common, domain-agnostic problems that need to be addressed to advance further research in this area.

1. Adaptive Query Processing A long standing database query needs to be robust against shift in the data characteristics. While we propose a fast algorithm for periodic recompilation of the primitive distribution, we do not address the issues of migrating existing partial matches from one SJ-Tree to another. This is a first order problem that needs to be addressed [108].

2. Partial Match Pruning The number of partial matches tracked by the SJ-Tree is a critical factor for performance. Partial matches need to be pruned as they become older or as the total memory usage approach a system-specific limit. Developing
intelligent strategies for periodic pruning of partial matches is an important topic to address.

3. **APPROXIMATE SEARCH** Develop approximate algorithms for retrieving top-K matches with a pattern instead of finding all matches.

4. **BENCHMARKING** Benchmark the performance of the SGEM-based dynamic graph implementation with other stream processing systems such as Storm [109] and Real-Time Giraph [110].

5. **BENCHMARKING** Benchmarking the dynamic graph search implementation on a subset of the C-SPARQL (SPARQL for Continuous Queries) benchmark.

6. **MULTIPLE QUERY PROCESSING** A database system is expected to run multiple queries at the same time. Given that bulk of the query processing time is spent on subgraph isomorphism, developing search strategies that exploits common substructures across multiple queries appears to be a natural progression.

7. **QUERY OPTIMIZATION** Incorporate statistical knowledge of graph evolution to prune unpromising partial matches.

8. **QUERY OPTIMIZATION** Dense subgraphs, and high-degree vertices are major hindrances to performance. The Lazy Search algorithm enables different types of searches
on every node in the data graph. Optimizing this search by exploiting local neighborhood statistics can yield significant performance benefits.

9. **QUERY RECOMMENDATION/REWRITING** There is a tradeoff between the generality of the queries and the efficiency of these queries. By taking advantage of relationships between query selectivity and dynamic matching efficiency, the best possible set of queries, trading off recognition accuracy and speed, can be determined.

10. **SCALABLE IMPLEMENTATION** *Streaming Graph Partitioning* is a hard problem. Developing workload-aware graph partitioning strategies that specially target performance bottlenecks in dynamic graph search will be important.
BIBLIOGRAPHY


[50] P. Zhao, X. Li, D. Xin, and J. Han, “Graph cube: on warehousing and olap multidimensional networks,” in SIGMOD ’11.


198


[97] A. Godiyal and J. C. Hart, “Enhancing network trac visualization by graph pattern analysis.”


