Dynamic Large-Scale Graph Processing over Data Streams with Community Detection as a Case Study

by

Tariq Abughofa

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
August 2018

Abstract

Processing large graphs provides invaluable insights for the industry and research alike. The applications range from e-commerce, web, and social networking to analyzing gene expressions and cellular signaling. While numerous graph processing solutions have been developed with the capability to process graphs at the scale of a trillion edges, the ability to maintain and process a real-time graph still far from being handled. Processing data streams in real-time requires the graph to change over time which introduces several new challenges. First, the graph needs to be updated from the data stream efficiently. At the same time, applying these changes should not add an unacceptable overhead to graph queries. In addition, these changes need to be reflected in the new analytical insights, otherwise the value of the insights degrades with time.

In this work, we investigated the problem of dynamic graph processing over data streams. We started by studying the feasibility of maintaining a dynamic graph on top of Apache Spark, a data processing engine. The chosen solutions included RDDs, IndexedRDDs, and Redis. Results from our experiments indicated that Redis performed the best, and thus we concluded that storing the graph in an external big data store besides Spark is the best approach in terms of performance and practicality. After that, we designed and developed Sprouter, a streaming data analytics framework which utilizes Spark for dynamic graph processing. The framework enables storing enormous graph data, allows real-time graph updates, and supports efficient complex analytics and OLTP queries. Experiments showed that the system is able to support the above mentioned properties and update graphs with up to 100 million edges in under 50 seconds in a moderate underlying cluster. Finally, we selected community detection as a case study of incremental graph analytics with Sprouter. We proposed the Incremental Distributed Weighted Community Clustering (IDWCC), a novel algorithm that optimizes the Weighted Community Clustering metric to detect communities in unweighted undirected node-grained dynamic graphs. We validated the algorithm against the static centralized and distributed versions of WCC optimization. The experiments showed that the performance of IDWCC surpasses the static distributed version by up to three times while maintaining the same accuracy or better.

Publications

Chapter 3 is based on: Tariq Abughofa, Farhana Zulkernine. Towards online graph processing with spark streaming. In: *IEEE International Conference on Big Data.* pages 2787-2794. IEEE, 2017.

Chapter 4 is based on: Tariq Abughofa, Farhana Zulkernine. Sprouter: Dynamic Graph Processing over Data Streams at Scale. In: *29th International Conference on Database and Expert Systems Applications.* DEXA, 2018.

Chapter 5 is based on: Tariq Abughofa, Farhana Zulkernine. Progressive Graph Partitioning over Streaming Data. *In preparation.*

Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, and Farhana Zulkernine. A Survey of Distributed Data Stream Processing Frameworks. *Submitted.*

Ftoon Kedwan, Tariq Abughofa, Haruna Isah, Farhana Zulkernine, Patrick Martin. Connecting Big Data: A Survey of Large-Scale Graph Processing and Management. *In preparation.*

Dedication

To my dad, Adnan Abughofa. You were my rich dad, poor dad. You sacrificed everything you could so I could be the man I am now. "Rabbit" has grown up a bit.

To my mom, Lina Haddad, who spent countless nights nurturing and tutoring me as a kid. I hope I achieved something that you would be proud of.

To the memory of grandma. I still remember you sweet lullabies.

To Canada, my new home. Thanks for opening your doors to me.

Acknowledgments

No words can describe my sincere gratitude to my supervisor, Dr. Farhana Zulkernine, for her valuable advice, patience, and support for the past two years. She was with no doubt the most understanding and supportive supervisor you can ask for.

I don't know how I would have finished this work without the support, encouragement, and friendship of the people I've met here at Queen's and Kingston. Tarek Mamdouh for his advice and mentor-ship since the day I joined the lab. John and Krista Casnig and the rest of the Sunday Dinner Crew for being like a family to me here in Kingston. World University Service of Canada members at Queen's for making me feel at home from the moment I arrived. The members of the database laboratory for being great co-workers.

I am grateful to the School of Computing and Queen's University for the financial and academic support. I also like to thank the donors for funding my research. They all gave me the opportunity to complete my Master's degree.

Most of all, my gratitude is to my mom and dad for always being there for me. You are both incredible and special people and I cannot express how lucky I am to have you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graph processing is one of the most important topics in big data processing [7, 22, 6, 24]. The graph structure has proved to be a good representation for a wide range of problems including social networks [35], targeted advertising [5], knowledge bases [4], mobile communications [39], and biological cell signaling [21]. The structure is also a natural fit for distributed processing as its algorithms work in an iterative manner allowing parallelism. This importance makes many advancements in machine learning and data mining techniques bounded by the ability to process this data structure efficiently and reliably [7, 22]. However, graph processing encounters many challenges especially as graphs grow to have billions of edges.

In this chapter, we present the motivation and the research challenges we try to solve in the area of graph processing.

1.1 Motivation

Due to the importance of graph data and the need to address big data challenges, numerous systems were developed for distributed graph processing. Some of these systems are Pregel [22], Giraph [7], and Spark's GraphX [40]. These systems are capable of processing large-scale *static graphs*, some to the magnitude of a trillion edges [7]. However, in the real world where the data is flowing and changing all the time, graphs are not static. The inability to utilize incoming data in real-time means that the analytical results are not up-to-date and hence inaccurate in frequently changing environments. Still none of these systems support the ability to update and process graphs in real-time. We refer to this problem as *dynamic graph processing* or *incremental graph processing* [8, 43] interchangeably in this research. We demonstrate the significance of this problem using two use case scenarios that are described below.

Recommendation Systems: Nowadays, there is nearly no e-commerce website that does not give recommendations. Recommendation systems are based on the fact that most people follow "social navigation" [11] which means that people with common interests have high influence on each others' activities. In this digital era, the most powerful commercial companies such as Amazon and Netflix have high-level industry-strength recommendation systems. Besides commerce, recommendation systems are used in search engines to enhance search results, social networks for matchmaking, and software to show help tips that are relevant to user behaviour.

The number of web users has been growing rapidly over the past few years. This growth is generating a massive amount of behavioural data. This requires recommendation systems to have the ability to process all this data in real time to provide high-quality recommendation service to the users. Since the graph structure is used widely in recommendation systems [5, 12], the need to append graphs and process them in real-time is crucial to avoid problems like the cold-start problem or giving out-of-date recommendations.

Internet of Things (IoT): We are rapidly approaching an era where we will be interacting with and using hundreds and thousands of interconnected digital devices. One time-sensitive example is patients in intensive care units which are often monitored by a variety of digital health devices and sensors to collect real-time health monitoring data. The collected data needs to be linked with patients' historical data and doctors' reports to provide real-time analysis and notifications about patients' health status. Short delays in such analytics and minor inaccuracies can have catastrophic consequences. Graph theory and online graph processing have proven to be an effective approach to analyze and query IoT data [42].

1.2 Research Challenges

Most graph processing systems are designed to process static graphs efficiently. Therefore, some of their design decisions do not suit evolving the graph with time. For example, Giraph is designed to load the graph from disk each time the processing is done [22]. This is not a approach solution when the graph is updated and processed over data streams. Another example is the use of immutable data structure to hold the graph as done in Spark [45]. This requires copying the graph with the added data into a new structure on each update which is not efficient.

Another problem is reflecting the updates in the results of future OLTP queries on the graph. At the same time, responses to these queries should be delivered without long delays. Designing storage abstractions and/or systems that *update the graph in real-time while delivering these updates to subsequent queries efficiently* is a challenging problem.

Producing analytical insights from a dynamic graph is another challenging problem. OLAP queries are generally time and resource intensive, and with data streams the value of previous insights degrades rapidly. *Innovating efficient analytical algorithms that behave in an incremental fashion* instead of processing the full data every time[1], is crucial for delivering valuable high-quality timely insights.

1.3 Hypothesis

Dynamic graph processing can be achieved with: a) a stream processing engine that can ingest and process data in real time, b) graph storage abstraction(s) that support updates and OLTP/OLAP queries efficiently, and c) the necessary dynamic graph processing algorithms.

1.4 Research Objectives and Scope

To validate the hypothesis, in this research we will:

1. Explore:
   - (a) Cutting-edge stream data processing systems, big data storage systems, and in-memory graph structures to select the right tools for building the framework.
   - (b) Data sets that can be used to demonstrate our research prototypes.
   - (c) Interesting graph analytic problems to demonstrate real-time graph analytics.

[1]If it is possible for the problem in hand. Some problems cannot be programmed dynamically by nature. An example of that is an the longest path in a graph.

2. Design and implement:

    (a) A streaming data processing framework with a big data persistent storage system and an in-memory graph storage. The persistent storage will serve OLTP and pinpoint queries with the help of indexes, while the graph structure will serve OLAP queries and allow efficient graph analytics.

    (b) An incremental graph processing algorithm for the community detection problem to demonstrate the effectiveness of our framework in finding communities with continuously changing data.

1.5 Contributions

The research has contributed to two conference papers (one published, one accepted) and three co-authored journal papers (one submitted, two in-preparation) as listed in the beginning of this dissertation. The technical contributions of our research can be summarized as follows.

The *first contribution* is conducting a comprehensive study of incremental graph processing over high volume and velocity streaming data. Our study identifies the feasible approaches, including a variety of proprietary and open source data processing engines and big data storage systems.

The *second contribution* is designing and implementing Sprouter, an end-to-end framework for dynamic graph processing. The framework combines different open-source tools including Spark, Spark Streaming, Cassandra, and NiFi, and can be extended to suit other streaming data analytics applications. It supports OLTP queries as well as complex analytics in a timely manner on a dynamically updated graph.

The *third contribution* is a novel incremental graph algorithm we call the Incremental Distributed Weighted Community Clustering (IDWCC), which we developed, validated, and incorporated in Sprouter to demonstrate a real-life application scenario. The use case scenario is delivering real-time recommendations through community detection in a large dynamic graph. The IDWCC algorithm works for unweighted undirected node-grained dynamic graphs. The experiments show that the performance of our new algorithm surpasses the existing distributed algorithm by up to three orders of magnitude while maintaining an accuracy that is slightly less than that of the static centralized algorithm (less than %5), and same or better than the static distributed algorithm.

1.6 Organization of Thesis

We proceed by introducing background knowledge and related work in the next chapter. We present and evaluate approaches to implement incremental processing of graphs with Apache Spark in Chapter 3. Chapter 4 describes the design and implementation of Sprouter, our proposed framework with experiments to validate its effectiveness in processing dynamic graphs. Our novel incremental community detection algorithm, as well as its implementation in Sprouter to demonstrate its capabilities, are presented in Chapter 5. Chapter 6 concludes the thesis and outlines the future work.

Chapter 2

Background

This chapter presents the background concepts relevant to stream data processing, graph databases, incremental graph processing, incremental graph algorithms, and the community detection problem in graphs.

2.1 Stream Processing Engines

Batch processing systems can serve the purpose when dealing with bounded data [10]. However, when the data is unbounded i.e. it comes as an infinite stream, those systems start to fail to follow up with the data flow [36]. At that point, it becomes necessary to have a system capable of analyzing data in real-time.

To be able to process unbounded data, there are several requirements the processing engine must satisfy. These requirements were discussed by Stonebraker at el. [36] and they are summarized below.

1. **Low Latency:** Latency is the time the system takes to perform message processing. Lower latency can be better achieved by avoiding storing data into the disk and using an active processing model.

2. **High-level Query Capabilities:** The engine should have APIs to support high-level queries on structured data whether it is script based, SQL based, or with graph traversal languages.

3. **Data Correctness and Consistency:** The engine should provide resiliency by handling stream defects such as missing, out-of-order, or delayed tuples.

4. **Integrating Stored Data with Stream Data:** The engine should provide data integration tools with scalable distributed storage systems such as HDFS and NoSQL databases.

5. **Fault Tolerance and High Availability:** The engine should support good fault tolerance techniques against crashes with minimum overhead. Reprocessing the data from the start of the working workflow after failures is not practical in real-time applications so the system should be able to recover the data produced by the latest step possible before the fault.

   Also when the system becomes incapable of handling the volume of the incoming data, it should be able to continue without failure. Some techniques can be used when this problem occurs:

   - The Fail-fast mechanism: Dropping the data that causes the overflow. However, correctness cannot be guaranteed in this approach [37].
   - The Back-pressure mechanism: Keeping the overflowing data stored to process it when the system overcomes the problem [19].

6. **High Scalability and Data Partitioning:** Scalability is becoming essential for achieving high performance. That can be done by supporting multi-threaded

operations to take advantage of multi-processor architectures, distributing the processing across multiple machines instead of centralized processing, and having the ability to add and remove machines on the fly. Distributed processing requires implementing techniques to distribute the data among the cluster i.e. "data partitioning".

Many big data processing systems are designed to satisfy the mentioned requirements such as Apache Storm [37], Twitter Heron [19], IBM Streams, and Apache Samza [26]. However, Apache Spark [45] is one of the most used systems for stream processing in recent years. Some of the reasons are its big development and support community, an in-memory processing paradigm, having simple APIs that support development in different programming languages, and the ability to support a wide range of processing workloads including batch, streaming, and iterative processing.

Apache Spark was first introduced as an in-memory processing engine to support iterative processing with good performance. Spark uses Resilient Distributed Datasets (RDDs) [45] as a distributed memory abstraction for large-scale computations. The programming interface provided by Spark allows the programmer to describe the data processing steps with known functional programming operations e.g. "map", "join", "filter", "reduceByKey". To make fault-tolerance possible without introducing the overhead faced with Hadoop, Spark logs the transformations only instead of the whole data. Recovery is less expensive as well because when an RDD partition is lost, Spark is able to recover by calculating that exact partition alone instead of the whole RDD as it has enough information to redo only its corresponding transformations. The recalculation is also deterministic due to two important properties of RDDs: immutability and reproducibility.

Later, Spark Streaming [46] was developed as a library on top of Apache Spark that allows passing micro batches of data to the processing engine to effectively simulate the behaviour of real-time processing engines.

2.2 Graph Frameworks

2.2.1 Graph Storage Systems

There are many graph-specific storage systems such as Neo4j[1], Sparksee[2], and TigerGraph[3], but many of them have questionable scalability. Neo4j, for example, still does not support sharding[4]. On the other hand, there are many solutions where NoSQL systems are used to store graph structure. One of these solutions is *JanusGraph*[5] which is a scalable graph database that enables storing and querying large data sets composed of billions of vertices and edges in a structure that is distributed across multiple machines.

JanusGraph stores the data in an external storage system. It has support for many storage backends such as Cassandra, Berkeley DB, and HBase. Internally, it stores the graph as a collection of vertices with their adjacency list. An adjacency list of a vertex in graph theory is a list of all the incident edges of that vertex. This storage format, which is demonstrated also in Fig. 2.1, ensures that each vertex and its adjacent edges are stored compactly in the storage backend to speed up traversals. The disadvantage is that each edge is stored twice: once for each end vertex. In addition to that, JanusGraph keeps the vertex list ordered by a sort key and each

[1]`https://neo4j.com`
[2]`http://www.sparsity-technologies.com`
[3]`https://www.tigergraph.com/`
[4]distributing data across multiple machines
[5]`http://janusgraph.org/`

	Column Family: default					
RowKey	$< edge1key >$	$< edge2key >$	...	$< property1name >$	$< property2name >$	...
$< vertexId >$	$< edge1properties >$	$< edge2properties >$	...	$< property1value >$	$< property2value >$	...

Figure 2.1: Table Scheme in Adjacency List Graph

	Column Family: default				
RowKey	fromVertex	toVertex	$< property1name >$	$< property2name >$	...
$< vertexId >$	$< vertexId >$	$< vertexId >$	$< property1value >$	$< property2value >$	...

(a) Edge Table

	Column Family: default		
RowKey	$< property1name >$	$< property2name >$	...
$< vertexId >$	$< property1value >$	$< property2value >$	...

(b) Vertex Table

Figure 2.2: Table Scheme in HGraphDB

adjacency list ordered by edge labels. This order allows faster retrieval of a subset of an adjacency list.

Another example is *HGraphDB*[6] which is a client layer built directly on top of HBase to use the latter as a graph data store. Unlike JanusGraph, HGraphDB represents the graph as two separate tables: one for the vertices and one for the edges. The argument for this choice is that it scales better for huge graphs. However, additional indexes must be used to provide efficient access to the incident edges of a vertex. This structure is explained in Fig. 2.2.

2.2.2 Static Graph Processing Systems

The increasing importance of processing graph data encouraged the development of many specialized *distributed graph processing systems.* These systems expose graph abstractions and naturally support executing iterative graph algorithms and many

[6]`https://github.com/rayokota/hgraphdb`

graph-specific optimizations to provide the required efficiency. As a result, these systems are undoubtedly superior for graph processing compared to the general purpose data processing systems in terms of performance.

A vertex-centric approach is used in implementing *Pregel* [22], a distributed graph processing library written in C++ by Google. This library is designed to be efficient, scalable, and fault tolerant. The processing is done in a series of "super-steps". In each step of the processing S, every vertex v that receives a message or more from the previous super-step $S-1$: (a) reads the received messages, (b) executes computations that typically make changes to v and its outgoing edges, and (c) sends messages to other vertices that will be received in super-step $S+1$. The execution is terminated at a super-step S^* in which all vertices are idle (no vertex receives messages).

This design is heavily influenced by the Bulk Synchronous Parallel model [38], where the parallelization happens during each step S where parallel computations can occur at all the vertices which receive messages from the previous step $S-1$. The synchronization is assured at the beginning of each step S due to the fact that the processing at that step is blocked until all messages from the previous step $S-1$ are received.

The graph in Pregel is divided into partitions that are distributed on the underlying servers. The data, which consists of the vertices and edges, is typically never moved between machines, and only the messages are transferred. This pure message passing model allows good performance and offers exactly-one delivery but does not guarantee the order which is not a real concern.

Pregel remained as a closed source library used by Google, but its design description in Malewicz et al. [22] inspired Facebook to build *Giraph* [7] which was donated later to the Apache Foundation. Apache Giraph is not just an open source version of Pregel, but has many extensions to the original model to elevate its usability, and implements performance and scalability improvements to accommodate a social graph of Facebook scale.

Giraph is implemented to work within the Hadoop ecosystem as it reads data from HDFS and Hive tables and depends on YARN for cluster management, but it does not depend solely on MapReduce task level parallelism. In order to increase the performance, it creates multiple workers per machine and supports multi-threading to exploit multiple CPU cores.

Twitter had a different philosophy regarding graph processing. Sharma et al. [35] assumed that the entire graph can be held in the memory of a single server. They estimated that their graphs cannot exceed 80 GB in size and hence they depended on expanding the resources vertically rather than horizontally. This assumption simplifies the system as it drops the need for graph partitioning and data movement. They developed many *centralized graph processing systems* starting with WTF (Who To Follow) [17] and Cassovary[7] until their graph systems evolved into real-time processing with *GraphJet* [35]. GraphJet is a graph-based system for generating content recommendations. It is an in-memory graph processing engine that maintains real-time user-tweet relations as a graph and works on a single machine.

Although the above-mentioned systems enable high-performance graph processing, they have some disadvantages. Since analytic processes generally require other data structures such as collections with graph processing, using multiple systems adds complexity and data movement overhead. Another problem is that these graph systems have fairly more complicated APIs compared to the basic data processing

[7]`https://blog.twitter.com/2012/cassovary-a-big-graph-processing-library`

operators such as map, join, and group by. Also, they generally lack the advanced fault tolerance techniques like in GraphJet or rely on snapshot recovery approach as used in Pregel. These problems led to the trend of bringing back *general-purpose data processing systems with graph abstractions* on top of them.

Apache Spark was the leader in this direction by introducing an embedded graph framework called *GraphX* [14]. Since Spark is an iterative processing system by nature, GraphX was developed as a graph abstraction library that sufficiently expresses graph APIs using basic data flow operators such as reduce, join, and map. The graph in GraphX is represented as distributed partitions and is highly fault tolerant like RDDs. This representation is based on the fact that graphs can be logically represented as a pair of vertex and edge collections where each vertex has to be identified with a unique key. The vertex collection contains a list of vertex properties associated with the mentioned key, and the edge collection contains pairs of source and destination vertices keys.

During the past few years, Spark started shifting focus to structured data by adding higher-level APIs such as Spark SQL. Spark SQL introduced DataFrames which are collections of structured records that can be manipulated using Spark's original procedural API, or using new relational APIs. As a part of this shift in Spark, *GraphFrames* [9] was introduced by moving to use DataFrames instead of RDDs for graph processing. This change allowed to generalize the idea behind GraphX from just executing iterative algorithms to supporting pattern matching and relational queries.

2.2.3 Dynamic Graph Processing

Most real-world graphs are dynamic as they change over time with new data producing new vertices and edges that need to be merged into the graphs. These frequent updates usually come as streams of data requiring dynamic graph processing. Dynamic graph processing typically requires merging the new data to the graph efficiently, serving online user queries, and supporting real-time analytics.

Twitter tried to solve the problem of incremental graph processing over data streams with the assumption that the entire graph can be held in the memory of a single server. This assumption simplified the system as it dropped the need for graph partitioning and data movement. Based on that, they developed their centralized real-time dynamic graph processing system GraphJet which we explained earlier.

Iyer et al. [18] presented GraphTau, a graph processing framework built on top of GraphX to process dynamic graphs. However, the framework is not open-sourced and is built on the RADIX trees similar to IndexedRDDs, which do not serve pinpoint lookups on graph streams as we show in Chapter 3.

Choudhury et al. [8] designed a framework with Spark to process dynamic knowledge graphs. The authors implemented many applications. However, none of the implemented applications require evolving the graph in real-time. Pattern discovery is done over data streams but it uses stream of edges passed to Spark to find patterns and thus incrementing the graph in real-time is not needed. Another application, question answering, is done with graph search but over a static graph which is loaded from HDFS and no streaming is done. Our focus is to find a solution that enables real-time incremental graph processing.

Spark has a library called Spark Streaming [46] that allows passing micro batches of data to the processing engine instead of big batches and thus serves as a near real-time processing engine. Although this allows Spark to support stream processing, evolving the graph over data streams is not presented as a feature in GraphX or GraphFrames.

2.3 Dynamic Graph algorithms

Dynamic graph scenarios require real-time solutions since applying the traditional static analytical algorithms to the whole graph each time the graph is updated is computationally expensive and impractical. Instead, dynamic graph algorithms typically process only the newly added data and propagate changes to the parts of the graph that are connected to the newly added data if needed. This optimization reduces the processing time significantly.

We can consider two different scenarios for time-evolving graphs when developing dynamic graph algorithms [43]. In the first scenario, the new nodes are added to the graph with all their incident edges simultaneously. These graphs are referred to in the literature as *node-grained dynamic graphs*. An example of such graphs is a graph of scientific papers and their references. Once a paper is published all the papers that it references are known as well and no new references (connections) are added later. In the other scenario, new edges can be added or removed at a later point of time for already existing vertices. Such graphs are known as *edge-grained dynamic graphs*. Social networks are a good example of these graphs as people add new friends and “un-friend” old ones all the time. Thus, the assumption of knowing all the connections of a person once we add them to the graph is not viable. In these networks, the sequence of adding new edges should be taken into consideration.

Dynamic graph algorithms can support one of these scenarios or both.

In the next section, we will introduce the graph community detection problem and discuss some of the existing solutions for static and dynamic graphs.

2.3.1 Community Detection

Community detection in graphs is the process of identifying groups of nodes that are highly connected among themselves and sparsely connected to the rest of the graph. Such groups are referred to in the literature as "communities" and occur in various types of graphs.

Detecting communities within large-scale graphs has become a very important topic [30, 43, 29]. First, it helps to discover new structural properties about the graph that cannot be found otherwise [3] such as the high-influence nodes also known as "community centroids". Second, the generated communities indicates special relationships between the nodes that can be hidden among the complex structure of the original graph. That unveils, for example, the tightly connected entities in business [39]. Third, these communities reveal the least connected parts of the graph. Thus, they can be utilized to implement optimized graph partitioning strategies to distribute the graph storage on multiple machines. This helps to increase the performance of distributed graph algorithms.

Community detection is an expensive problem since most existing solutions are based on expensive computations that are hard to scale. Therefore, the scalability and efficiency are critical aspects of any solution. In dynamic graphs, the problem becomes more complex as exploring the communities for the whole graph again on each new micro-batch of streaming data becomes unfeasible.

One of the most popular community detection methods is Label Propagation [48, 31], which is already implemented as a part of the GraphX library. This algorithm chooses the community of the current node using the labels of neighbor nodes. However, it only helps in finding overlapping communities.

A variety of community detection algorithms were developed in the past few years to detect disjoint communities. Modularity is considered the most prominent quality measure for community detection [2]. It prioritizes communities based on their internal edge density. One of the most popular algorithms based on modularity optimization is the Louvain Algorithm which is presented in detail by Blondel et al. [2]. This algorithm is a greedy optimization that can be used for weighted graphs. The algorithm starts with each vertex as its own community. Then, it progresses in an iterative manner where each iteration consists of two phases. The first phase calculates the gain in modularity from adding each vertex to a neighboring community and adds the vertex to a community which produces the highest gain. This gain in modularity ΔQ when a node i is moved into a community C is calculated using equation 2.1. $\sum_{in}$ is the sum of the weights of the links inside C, $\sum_{tot}$ is the sum of the weights of the links incident to nodes in C, k_i is the sum of the weights of the links incident to node i, $k_{i,in}$ is the sum of the weights of the links from i to nodes in C and m is the sum of the weights of all the links in the network.

$$\Delta Q = \left[\frac{\Sigma_{in}+k_{i,in}}{2m} - \left(\frac{\Sigma_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \tag{2.1}$$

In the second phase, a new graph is built consisting of the generated communities as its nodes. Then, the first phase is applied again to the generated graph until further changes fail to improve the modularity anymore.

The Louavain algorithm proved to have a good scalability as it successfully processes graphs up to the size of 118M Nodes/1B edges [2]. However, the modularity value can become a bottleneck when the implementation is in a distributed computing environment since the value has to be calculated each time a vertex changes its community. In addition, many studies reported that the algorithm maintains good performance as the graph grows in size, but it degrades in quality [13, 1].

There are several other community detection algorithms that are based on random walks. When a random walk is performed in the graph, there is a high probability that close-knit communities are explored first, since the density of internal connections is high. One of the most famous methods based on random walks is Infomap [32].

More recently another measure was introduced called the Weighted Community Clustering (WCC) metric [28]. This measure defines the quality of communities based on how dense they are in terms of triangles. Later, it was used to propose the Scalable Community Detection (SCD) algorithm [30] which can be used to detect communities in undirected unweighted graphs. Experiments showed that the algorithm has better result quality than the Louvain algorithm while keeping a good performance. A distributed version of the algorithm based on the vertex-centric paradigm produced by the Pregel platform [23] was developed and introduced later by Saltz et al. [33].

In the area of incremental community detection, many modularity-based solutions exist. Shang et al. [34] introduced an algorithm that depends on the Louvain algorithm to find an initial community structure. After that, it starts detecting communities for new vertices. Thus, its results depend tightly on the Louvain algorithm. Pan et al. [27] designed a method for edge-grained graphs. The problem with this method is it assumes that the edges are added in a certain order. As a result, they cannot handle node-grained graphs properly as the edges are added simultaneously resulting in poor performance [43].

Very few solutions exist for incremental community detection on node-grained graphs. A recent one called the Node-Grained Incremental community detection (NGI) is proposed by Yin et al. [43]. The algorithm is developed for node-grained dynamic graphs based on modularity optimization. However, it was implemented for centralized processing only.

2.4 Summary

In this chapter, we discussed stream data processing and Apache Spark. Then we talked about graph processing including graph storage systems, processing engines, and incremental graph algorithms. The next chapter presents our work on studying incremental graph processing over streaming data.

Chapter 3

Incremental Graph Processing with Spark Streaming

The first challenge in implementing dynamic graph processing is being able to maintain a dynamic graph while delivering the updates in any subsequent queries in a timely manner. In this chapter, we present *our study towards implementing an online graph structure that can be updated from a data stream while, at the same time, respond to OLTP queries.* An acceptable solution should provide both abilities efficiently.

As a first step, we decided on using Apache Spark, a popular in-memory processing engine as it supports stream and graph processing. After that, we studied the possible solutions to implement incremental graph with OLTP queries abilities with Spark. We chose two data stores that comply with the basic requirements namely, good fault tolerance, fast writes, and fast reads. These data stores are: IndexedRDDs, a new library for Spark based on Spark's original RDD structures to enable fine-grained updates as a key-value store, and Redis, a popular distributed in-memory key-value store.

This chapter discusses incremental processing in Spark and explains IndexedRDD and Redis in the background section. Then, the detailed research and experiments conducted to select the suitable in-memory graph structure are presented in the implementation and validation sections respectively.

3.1 Background

3.1.1 Incremental processing in Spark

One of the key points in the design of RDDs is *immutability*, which allows task replayability to ensure two features: fault tolerance by replaying failed tasks and straggler mitigation by replicating slow tasks. The lack of data mutability does not form a problem for Spark as it is mainly a computational engine, not a data store.

However, most real-life applications do not only require performing computations on data streams. Instead, the steam processing comes as a step in a larger application. In Zaharia et al. [44] such applications were called ”continuous applications”. The article mentioned many examples of these applications, but we are interested in a specific subset in which the data changes with time, and thus the immutability can be a challenge. We will call this specific set of applications ”incremental continuous applications”. Here are some of these scenarios:

1. **Modifying data that will be served in real-time**. These cases require aggregations over streaming data, such as the number of followers a user has or the number of likes on a certain post.

2. **Evolving data that will be served in real-time**. The data in these systems evolve with time. An example would be online graphs. The data store should

handle both updates from the streaming engine and the queries coming from the front-end system.

3. **Online machine learning**. These continuous applications often combine large static datasets with real-time data for online predictive analytics.

There are three feasible solutions for such applications with Spark as the stream processing engine:

1. **A separate data store.** Saving batches of the data in a transactional data store like Cassandra or Redis. This can add efficiency problems by adding the serializing/deserializing cost and complicate parallel recovery.

2. **The RDD abstraction.** Making new RDDs for each update. However, RDDs were designed for coarse-grained transformations, and can be inefficient for fine-grained updates over streaming data.

3. **A different dataset abstraction in Spark.** Designing a new dataset abstraction other than RDDs that can perform efficient fine-grained updates and still guarantee good fault tolerance.

To study incremental graph processing with Spark, we chose three data stores that represents the mentioned solutions and capable of holding a graph structure. Next, we will discuss and explain our choices.

3.1.2 Redis

Key-value stores are a category of NoSQL databases that follows a simple data model based on associated map data structure. Each single data object is a pair of a key representing a unique identifier and a value that can be a string, an integer, an array, a set, or other basic data structures, making this type of databases schema-less [16]. Redis[1] falls into this category of NoSQL databases that applies an in-memory storage model making it perfect for use cases such as web sessions, shopping carts, and data caching. This data store provides a fast querying interface and supports data structures such as strings, hashes, lists, sets, and sorted sets.

To provide a more robust storage, Redis implements a master-slave replication model with asynchronous replica reflection [16]. In master-slave replication, a single node is assigned as the master and the rest as replicas. Writes are only allowed to the master node and then they are propagated to the replicas, while reads are allowed to any node. This way, replication increases the availability of reads without adding extra latency on the writes since a write operation succeeds before the change is propagated to the replica nodes.

Redis has also a cluster implementation that allowed it to act as a distributed database. This implementation adds the ability to split data among multiple servers which is known as sharding, and to tolerate network partitioning i.e. continuing to operate in the case of server failures.

The reason behind choosing Redis over many other NoSQL options is that it is a high-performance data store with simple structured data types which allows it to work as a good solution for big data applications. For example, it was suggested as a solution to store social graphs for real-time processing in Sharma et al. [35]. In distributed environments with increasing availability and reliability requirements, it becomes more fit for such a purpose. It is also the most popular in-memory key-value store.

[1]`https://redis.io/`

3.1.3 IndexedRDDs

To be able to update RDDs in real-time, IndexedRDDs were introduced as an efficient fine-grain update key-value stores for Spark[2]. An indexedRDD is an RDD of key-value tuples that ensures key uniqueness and still distributes storage and supports all the operations of a normal RDD. The index is implemented using a data structure called Persistent Adaptive Radix Tree (PART)[3] [20]. The PART structure allows efficient updates without changing the original data using a versioning mechanism where new RDDs share most of the structure and the data with older RDDs, while only differing on the parts that contain the new and updated keys. The distribution is done by hash-partitioning on the keys and maintaining a radix tree for each partition.

Another problem that affects the performance of frequent updates to RDDs is the check-pointing strategy. Spark performs check-pointing regularly for long RDD chains by writing the current RDD data to disk to avoid very long fault tolerance operations. In the case of faults, instead of depending solely on the lineages, Spark can recover from the checkpoint and redo only the lineages from that point forward. However, that can be expensive in the case of IndexedRDDs since they change very frequently. To make checkpointing faster, IndexedRDDs implement a tree partitioning strategy on the Radix tree to segregate the frequently-changed parts of the tree from the less frequently-changed ones. This idea is based on the observation that the top nodes of the tree have higher changing frequency than the lower ones. For example, the top most partition of the tree changes on each update.

We chose IndexedRDD since it is a native library to Spark and because it was built, as previously explained, to support fine-grain updates which matches our purpose.

[2]`https://github.com/amplab/spark-indexedrdd`
[3]`https://github.com/ankurdave/part`

3.2 Experiments

To choose a suitable data store for incremental stream processing in spark, we compare the performances of RDDs, IndexedRDDs, and Redis. We need a data store that ensures the following properties:

1. Allows for very fast fine-grain updates on the data.
2. Supports fast data read operation.
3. Has good fault tolerance to overcome network partitions and crashes.

We used the Twitter public API to consume data representing popular tweets and their corresponding authors. We chose to save the data in a graph structure represented by two lists of vertices (users and tweets) and edges (a pair of ids of the user and their tweet). In the experiments we performed the following operations for each type of data store:

1. Inserting a micro-batch of vertices.
2. Inserting a micro-batch of edges.
3. Looking up a vertex by id.
4. Looking up an edge by id.
5. Multiple vertex lookups by ids.
6. Multiple edge lookups by ids.

3.2.1 The Environment

To test our system, we used 5 identical machines each with 8 cores 2.10 GHz Intel Xeon CPU, 64-bit architecture, 16 GB of RAM, and 300 GB of disk space. We installed the Hortonworks Data Platform (HDP-2.6.1.0) with Apache Spark 2.1.1 and the Spark Streaming library. We also installed Redis on one of the machines with two other replicas.

3.2.2 Data Source

For the purpose of conducting our experiments, we needed to build a graph structure that is big enough for the results to reflect the difference in performance among the chosen data stores. We chose the Twitter Streaming API[4] as a source to build the graph first and then to do the insert in the testing phase. We chose to use the “public streams” endpoint which gives the public data that flows through twitter for all users. This was perfect for our purpose since we wanted to build a graph which has users and tweets as its nodes and ”tweeted by” relations as its edges. The API returns an array of *Status* objects which represent tweets and each one of these has as one of its properties the *User* object as shown in Fig. 3.1. Both *User* and *Tweet* objects have an id property which will serve as the index key for the nodes, while the edge key will be a combination of the ids of its user id and tweet id. We should mention that the figure includes only one *Status* object and that many of the irrelevant attributes were truncated for a simpler view.

[4]`https://dev.twitter.com/streaming/overview`

```
[{
  "id": 3252356,
  "text": "this is a sample tweet",
  "user": {
      "id": 78230235,
      "name": "John Doe",
      "screenName": "John Doe",
      "followersCount": 22232,
      "friendsCount": 1908
  },
  "isTruncated": false,
  "isRetweeted": true,
  "place": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "country": "US",
    "postalCode": "10021-3100"
  }
}]
```

Figure 3.1: An example of Twitter Streaming API response

3.2.3 Experimental Details

The experiment to benchmark the performance of our data stores included the following steps:

1. Loading the data from HDFS as a data stream of RDDs.

2. Constructing a graph from the data and saving the graph in an RDD, an IndexedRDD, and a Redis hash map.

3. Starting a streaming context from the Twitter public API and pass the data to Spark Streaming.

4. For each RDD read from the stream, executing: (a) a *bulk insert* operation (which include inserting a user node, a tweet node, and a ”tweeted by” relation for each item in the RDD), (b) a *lookup* for a random key chosen from the previously inserted nodes and edges, and (c) a *multi-lookup* for 50 random keys

chosen from the previously inserted nodes and edges. These three operations were done for each data store (RDD, IndexedRDD, Redis).

Graph Operations with RDDs

The graph structure in RDDs consists of two RDDs. First, the vertex RDD which consists of tuples of IDs and values. Each tuple represents a user or a tweet. Distinguishing between the two is irrelevant for the purpose of this experiment. Second, the edge RDD, which is also a list of tuples of IDs and values of a "tweetedBy" relation between a user and a tweet. The ID is generated by concatenating the IDs of the user and the tweet that represent the edge. We should mention that this structure is very similar to the one in GraphX [14].

```scala
// 1. insert
// We use "reduceByKey" to remove duplicates
fullVertexRDD.union(vertexRDD).reduceByKey {
   (value1, value2) => value2
}
fullEdgeRDD.union(edgeRDD).reduceByKey {
   (value1, value2) => value2
}

// 2. lookup
fullVertexRDD.lookup(vertexKey)
fullEdgeRDD.lookup(edgeKey)

// 3. multi lookup
vertexKeyList.map(fullVertexRDD.lookup(_))
edgeKeyList.map(fullVertexRDD.lookup(_))
```

Listing 3.1: Operations with RDD (in Scala)

The **bulk insert** operation is done with the `union` transformation available through the RDD API which returns a union of two RDDs. After that a reduceByKey transformation is needed to remove duplicates since RDDs does not ensure any key uniqueness. The **lookup** operation can be done straightforward with the `lookup` transformation. However, the RDD API does not have a native support for **multi-lookups**, so we do it by iterating on the keys while doing a single lookup in each iteration. In Listing 3.1 we present the code for the above operations.

```scala
// 1. insert
indexedVertices.multiputRDD(vertexRDD)
indexedEdges.multiputRDD(edgeRDD)

// refresh the RDD to reflect the insertion
indexedVertices.cache().foreachPartition {_ =>}
indexedEdges.cache().foreachPartition {_ =>}

// 2. lookup
indexedVertices.get(vertexKey)
indexedEdges.get(edgeKey)

// 3. multi lookup
indexedVertices.multiget(vertexKeyList)
indexedEdges.multiget(edgeKeyList)
```

Listing 3.2: Operations with IndexedRDD (in Scala)

Graph Operations with IndexedRDDs

The graph structure here is identical to the one in RDDs. However, the structure is indexed on the first element of the tuples (the ID) which should support faster lookups.

We show the operations in Listing 3.2. The code shows that the **bulk insert** operation is done with the `multiputRDD` transformation available through the IndexedRDD API which returns a union of an IndexedRDD with an RDD (from the stream). The **lookup** operation is available with the `get` operation, while the **multi-lookup** operation is available with the `multiget` operation.

Graph Operations with Redis

We use two hash maps in Redis to create the graph data structure. The first one is for the vertices, which contains the tweet/user id as a key and the related information as the value. The second hash is for the edges and has the concatenated ID of the corresponding user and tweet (as in the previous solutions) as the key, and the edge information as the value.

```
// 1. insert
client.hmset("vertices", vertexRDD.map(mapper))
client.hmset("edges", edgeRDD.map(mapper))

// 2. lookup
client.hget("vertices", vertexKey)
client.hget("edges", edgeKey)

// 3. multi lookup
client.hmget("vertices", vertexKeyList)
client.hmget("edges", edgeKeyList)
```

Listing 3.3: Operations with Redis (in Scala)

As demonstrated in Listing 3.3, the **bulk insert** operation is done with the `hmset` command available in Redis. The **lookup** and the **multi-lookup** operations are done with the `hget` and `hmget` commands respectively.

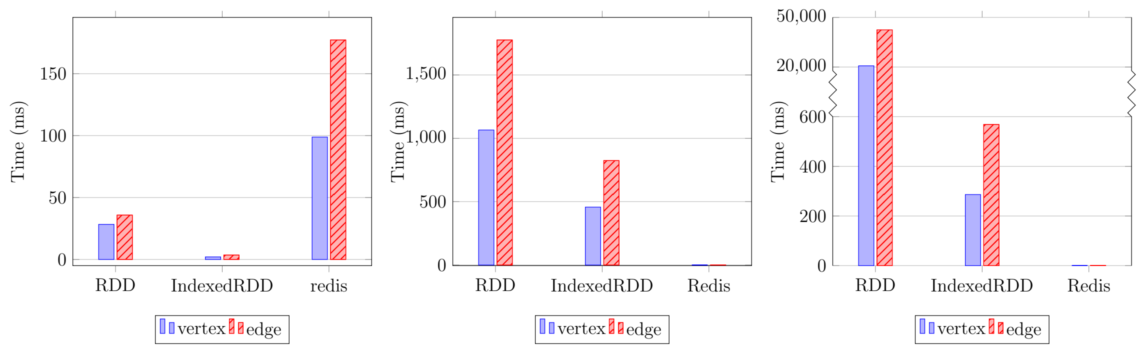
The graph was constructed using Spark Streaming by reading the data from the twitter public streams and saving the RDDs into HDFS as object files. The constructed graph had 898,632 vertices and 449,316 edges.

We repeated the experiment for 80 consecutive windows from the stream where each window was 10 seconds long. That means that the operations were executed 80 times for each data store as the graph grew with each window. For each operation, we measured the time it took to be executed and we took the average. The results are shown in Fig. 3.2.

3.2.4 Results

The performance of RDDs in our experiments was poor, which was expected since they are not built for frequent changes and do not use indexing for lookups. The bulk insertion took 28.33 ms for vertices and 35.83 ms for edges which is long for our requirements. The lookups were far worse than other data stores: edge and vertex lookups took 1,065.4 ms and 1,776.2 ms and multi-lookups took 21,375.8 ms and 42,700.2 ms respectively. This is due to the fact that RDD data is not indexed and that multi-lookup is basically the same operation as doing the lookups sequentially. The time for lookup increases with time as the data grows bigger since the operation does a sequential scan of the RDD.

Next, we experimented with IndexedRDDs. 2.05 ms was required to perform the bulk vertex insertion and 3.55 ms for the bulk edge insertion. Lookups were good as well: 456.88 ms for vertex lookups, 824.18 ms for edge lookups, 286.35 ms for vertex multi lookups, and 568.56 ms for edge multi lookups. The results were impressive, but the store requires doing a refresh for the index after each insert. This refresh operation was slow and increased the lookup cost (in case it was done on each



(a) Insertion (b) Lookup (c) Multi-lookup

Figure 3.2: Cost of Operations

lookup) to 1,136.59 ms.

Finally, we tried Redis. This key-value data store proved to be a very fast store for lookups and insert operations. However, we wanted to prove that it performs as fast with our large dataset. The results matched our expectations: bulk vertex insert took 98.84 ms, bulk edge insert took 177.28 ms, vertex lookup took 0.32 ms, edge lookup took 0.66 ms. Multi lookups are built into Redis, and they were close in execution time to normal lookups as expected; Vertex multi-lookups took: 0.46 ms and edge multi-lookups took: 0.82 ms. Therefore, Redis can serve as a very good store on top of Spark but the problem is that it does not support saving property graphs[5]. Also using a hash structure to save graphs neglects the partitioning capabilities of Redis which makes the scalability of this approach questionable.

We summarize our conclusions from experimenting with different solutions for incremental graph processing with Spark in Table 3.1.

[5]A property graph is a graph with vertices and edges that can be associated with properties

Table 3.1: Incremental Graph Processing Approaches with Apache Spark

Approach	**Advantages**	**Disadvantages**	**Summary**	**Examples**
RDD Abstraction	– Good fault tolerance. – Native embedded store in Spark.	– No support for multi-lookup. – Very expensive lookups.	Does not work as a fine-grain-updates store.	–
Different Spark Abstraction	– Good fault tolerance. – Embedded in Spark. – Built for fine-grained updates.	– The existing solution forces rebuilding the index on each insert. – Developing a solution is very expensive as it requires drastic changes to Spark's APIs and libraries.	The only existing solution has fast inserts but the lookups are slow after updates, and thus not suitable for streaming data where lookups are expected all the time.	IndexedRDDs
External Big Data store	– Replication and good fault tolerance. – Fast reads/writes. – A plethora of options to choose from.	– Adds a serialization/deserialization cost. – Might need to implement graph abstraction. – Might not have support for property graphs. – Might have limited support for complex graph queries. – Higher complexity with more systems to manage.	Should investigate the solution scalability, performance, and suitability for storing complex graphs.	Redis, HBase, Neo4J, Cassandra... etc.

3.3 Summary

In this chapter, we conducted a study on incremental processing in Spark. We chose graphs as the struture to increment and three data stores including RDDs, IndexedRDDs, and Redis as possible candidates to store a graph data structure on top of Spark Streaming. After that, we conducted a set of experiments to compare the performances in terms of the ability to represent and manipulate graph data, and support fast fine-grain updates and acceptable read latency. The results of our experiments showed good insertion times for IndexedRDDs but it had large lookup times as the index has to be rebuilt on each insert to be able to look up for recent data, which adds an extra time to each lookup that was preceded by an insert. Redis showed good results for both inserts and lookups, but it lacks support for property graphs. We concluded that the most practical and efficient solution is to have an external storage next to Spark to support OLTP queries on a dynamic graph.

Chapter 4

Sprouter: Dynamic Graph Processing over Data Streams at Scale

We propose Sprouter[1], a framework that supports dynamic graph processing. In order to do so, the framework processes real-time data streams using Apache Spark, and thereby, updates an associated large-scale dynamic in-memory graph. At the same time, it maintains the graph in a big data store. The framework efficiently answers OLTP queries using the persistent storage, and OLAP queries, which requires complex analytics, using the in-memory graph.

In this chapter, we build on our conclusions from the previous chapter by using an external storage next to Spark to hold the graph for fast writes and reads. We explain the design of the framework and demonstrate its abilities by ingesting streams of social data. Our experiments show that a graph with up to 100 million edges can be updated in under 50 seconds in a moderate underlying cluster. The implementation builds up for implementing community detection as an incremental graph processing use case in the next chapter.

[1] `https://github.com/TariqAbughofa/sprouter`

4.1 The Sprouter Framework

Based on our research objectives, the proposed system should support the following:

1. In contrast to most graph processing systems which are built to process static graphs, the graph construction is viewed as an incremental process as the graph evolves with time.

2. The system supports executing simple queries as well as complex algorithms on the dynamically updated graph.

3. All components should scale to accommodate the huge growth of data.

To accomplish that, the framework requires the following components:

- Data Ingestion: The data is ingested from the source. The data ingestion can be for historical data to be processed in bulk or for new data as a stream.

- Graph Storage: Data is stored as a graph structure in both a persistent storage and an in-memory storage.

- Bulk Processing: Historical data is read from the data lake and processed. A graph is constructed from this data.

- Stream Processing: Newest data is consumed as a stream. The graph is updated with the new nodes and relations and then any required graph processing is executed.

- Graph Search: The system supports entity and relationship queries (OLTP), and analytical graph algorithms (OLAP).

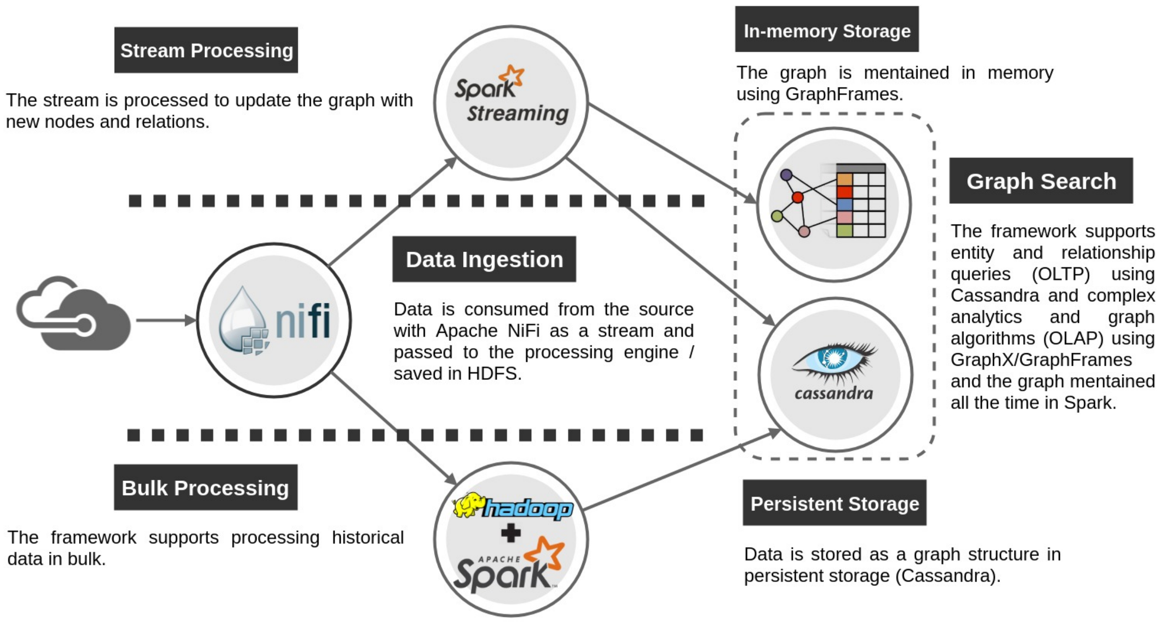

Stream Processing
The stream is processed to update the graph with new nodes and relations.
Spark Streaming
In-memory Storage
The graph is mentained in memory using GraphFrames.
Graph Search
The framework supports entity and relationship queries (OLTP) using Cassandra and complex analytics and graph algorithms (OLAP) using GraphX/GraphFrames and the graph mentained all the time in Spark.
nifi
Data Ingestion
Data is consumed from the source with Apache NiFi as a stream and passed to the processing engine / saved in HDFS.
cassandra
Bulk Processing
The framework supports processing historical data in bulk.
hadoop
Apache Spark
Persistent Storage
Data is stored as a graph structure in persistent storage (Cassandra).

Figure 4.1: The Sprouter Framework Design

The design is shown in Fig. 4.1. The design decisions and the setup are explained next.

4.1.1 Data Ingestion

We chose to use Apache NiFi[2] as it supports building data flows through a web-based drag-and-drop graphical interface without the need to write any code.

This component performs two tasks: First, it ingests the historical data and stores it in HDFS for bulk processing. Second, it ingests the most recent data as a stream for real-time processing.

[2]`https://nifi.apache.org/`

4.1.2 Graph Bulk Processing and Stream Processing

We need to support bulk processing to be able to process historical data as well as data left in the queue in the case of downtime. We chose Apache Spark as our processing engine for many reasons. First, it is a unified engine that supports bulk and stream processing which simplifies the framework design. Second, it has two graph abstractions that allow efficient graph processing namely GraphX and GraphFrames. Third, it is a reliable mature engine that has proven efficiency and scalability [47].

4.1.3 Persistent and In-memory Storage

We keep the graph stored in-memory for efficient graph analytics with Spark. Evolving the graph with GraphFrames, however, is not a straightforward operation. Listing 4.1 shows how to increment the graph using GraphFrames in Scala.

Listing 4.1: Graph Appending in GraphFrames

```scala
val  fullVertices  = oldGraph.vertices
  .join(newVertices, Seq("name"), "outer")
  .select("name").withColumn("id", monotonically_increasing_id)
val fullEdges = graph.edges.union(newEdges)
val fullGraph = GraphFrame(fullVertices, fullEdges).cache()
```

RDDs are not efficient for pinpoint lookups especially with fine-grained updates over streaming data as we found in Chapter 3. We also stated that no out of the box solution exists today to address this problem. To support simple graph queries such as vertex lookup or getting edges connected to a certain vertex, we need to use an external graph-based storage system.

Redis proved to be a good solution as we found in the previous chapter. However, Redis does not support storing property graphs and has limited graph query capabilities. Hence, we needed to choose a different storage layer. The choice was based on the following criteria to guarantee that the solution works with streaming data:

1. The storage system has to be a scalable distributed NoSQL data store that is write-efficient and performs well for Big Data.

2. It should have the ability to save/load Spark Streaming micro-batches into the datastore efficiently in bulks.

3. Availability with eventual consistency is definitely preferred over strict consistency.

4. It should support OLTP queries such as finding the neighbors of a vertex.

Our initial candidates were HBase and Cassandra. Both databases are scalable NoSQL systems that proved to have good performance for Big Data applications. However, HBase will not have bulk loading functionality until its third release[3]. In addition to that, it does not offer full availability in order to ensure full consistency. Cassandra, on the other hand, has bulk loading/saving and highly available.

We chose to adapt the HGraphDB[4] storage model on top of Cassandra instead of other models such as JanusGraph to achieve greater scalability and efficiency for a broader scope of queries. HGraphDB uses separate tables for the edges and the vertices as explained in Chapter 2.

[3]`https://issues.apache.org/jira/browse/HBASE-14150`
[4]`https://github.com/rayokota/hgraphdb`

4.1.4 Graph Search

We are concerned with two types of graph search: simple OLTP queries and OLAP or analytical queries such as PageRank and Community Detection. GraphX and GraphFrames are not efficient for OLTP queries as we found out in Chapter 3, but Cassandra provides an efficient solution for that. However, GraphFrames provides highly flexible APIs and many pre-implemented graph algorithms with the ability to implement new ones. Based on that, we decided that any OLAP or analytical queries should be executed on the in-memory graph maintained at all times using GraphFrames.

4.2 Experiments

4.2.1 The Environment

We used 6 identical machines each with 8 cores 2.10 GHz Intel Xeon CPU, 64-bit architecture, 24 GB of RAM, and 300 GB of disk space. We installed the Hortonworks Data Platform (HDP-2.6.3.0) on our machines with Spark v 2.2.0. HDFS is deployed on all the nodes and has 870 GB in total as reserved space. In addition to that, Cassandra (v 3.11.1) is installed on all 6 nodes to form a cluster. Two of the six Cassandra servers are used as seed nodes for the cluster, which serve as contact points for the new nodes that attempt to join the cluster.

For all the experiments, we used 12 executors with 3 cores each and 5 GB of RAM. This configuration makes sure that there are enough cores to be used by all the executors and keeps some to be used by other services (including Cassandra). We validated our framework by demonstrating dynamic and distributed graph updates from streaming data as new data is appended to an existing graph. We also executed pin-point queries on the graphs in both Cassandra and GraphFrames.

4.2.2 Data Source

We used the Global Data on Events, Location, and Tone (GDELT)[5] as a data source to feed the framework. GDELT contains data from news media from all over the world in print, broadcast, and web formats. In this study, we used the Global Knowledge Graph (GKG) v2.1 which contains the full news graph connecting persons, organizations, locations, emotions, events, and news sources. This dataset provides a good source for bulk data with more than 250 million historical events. In addition, it can be used as a data stream since updates are published every 15 minutes. Based on these properties we find that this dataset is suitable to demonstrate the capabilities of our framework. A record of the dataset is demonstrated in Listing 4.2.

Listing 4.2: A record from the test data in JSON format excluding irrelevant fields

```
{
  "GKGRECORDID": "20180128150000-0",
  "V2.1DATE": "20180128150000",
  "V2SOURCECOMMONNAME": "msn.com",
  "V2DOCUMENTIDENTIFIER": "https://www.msn.com/en-ca/news/canada/no-one-denied-flight-
       because-of-no-fly-list-mistakes-government-memo-says/ar-BBIlaAr", ...
  "V2ENHANCEDPERSONS": [
    { "name": "Scott Bardsley", "offset": "1082" },
    { "name": "Justin Trudeau", "offset": "1690" },
    { "name": "Catharine Tunney", "offset": "199" },
    { "name": "Khadija Cajee", "offset": "2045" } ...
  ], ...
}
```

[5] https://www.gdeltproject.org/

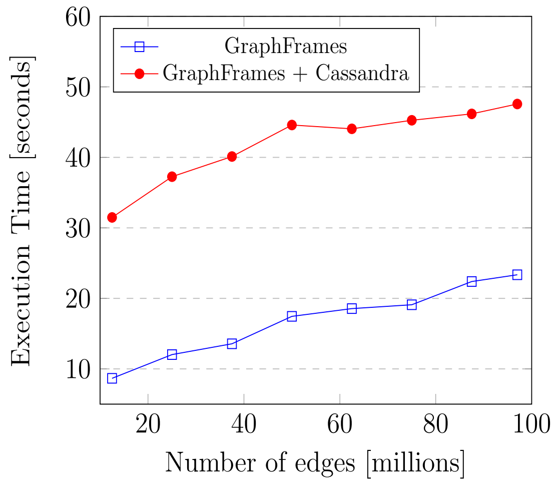
The file is parsed and the person list, which is under the attribute `V2ENHANCEDPERSONS`, is extracted and then sorted by the “offset” attribute. This attribute defines how far (in number of characters) from the beginning of the article a person was mentioned. We define that two people have an *undirected connection* if they are mentioned in the same paragraph. For simplicity, we assumed that it means that they are less than 1000 character apart (the average length of a paragraph) based on their offsets. We then drop self-relations and duplicates.

We explain the implementation details of the data ingestion component for the GDELT data in Appendix A.

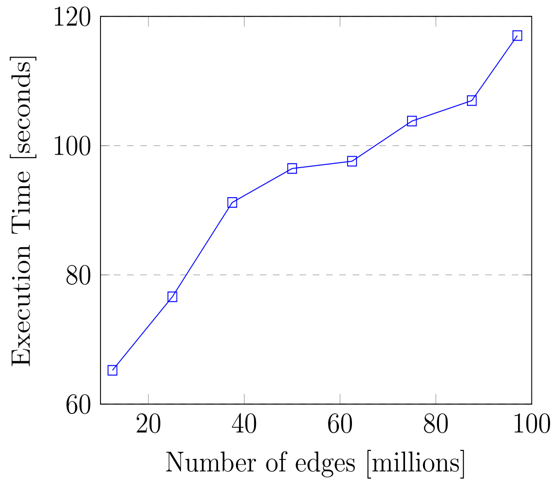
4.2.3 Results

To conduct the experiments, we consumed historical data from the GDELT repository for the years of 2017 and 2018 and we stored this data in our cluster in HDFS format. This data allowed us to construct a graph consisting of 13,290,365 vertices and 97,134,816 edges using Spark, which we stored in Cassandra in the HGraphDB format as described in Chapter 3. Then we ran the streaming job which loads the graph into the in-memory graph structure using Spark Streaming and then reads the GDELT “last” file. We made sure that the stream consumed always the same “last” file for all the experiments to ensure repeating the exact experiment and thus the consistency of our results. The file we chose contained information about 12,738 articles which generated 51,060 vertices and 87,099 edges that needed to be appended to the full graph.

We implemented two approaches for the append operation. **The first approach** merges the new data is merged with the GraphFrames graph directly as described



(a)



(b)

Figure 4.2: Appending data: (a) directly to GraphFrames (b) into Cassandra first then loading it to GraphFrames

earlier in Listing 4.1. In Fig. 4.2a, we show the time it took to update different sizes of graphs in GraphFrames. The plot in blue shows the time spent to append the graph in GraphFrames without persisting in Cassandra. The red plot shows the total time to append the in-memory graph and then bulk-saving the updates in Cassandra. In practice, we will need to persist the new vertices and edges in Cassandra, and therefore, the red plot represents the actual time to append a graph in the framework. Fig. 4.2a clearly shows that the time increases linearly with the graph size from 8.66 seconds for a graph with 125m edges to 23.35 seconds for a graph with 97m edges.

Figure 4.2b illustrates the performance of **the second approach**. In this solution, we first update the tables in Cassandra with new edges and vertices. Then we bulk-load the complete graph from Cassandra to GraphFrames. This approach showed poor execution time that increases with graph size which is expected since it reads the full data from disk to memory. The slope for this approach is bigger and the execution time starts at 65.23 seconds for a graph of size 12.5m edges and increases to 117 seconds for a graph having 97m edges. This also shows that appending in GraphFrames directly remains a better solution as the graph grows.

Table 4.1: Summary of the experiment numerical results

Graph	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
Links	12.5m	25m	37.5m	50m	62.5m	75m	87.5m	97m
Nodes	4.7m	7m	8.4m	9.5m	10m	11m	11.9m	13m
GF	8.66s	12.03s	13.56s	17.45s	18.55s	19.1s	22.39s	23.35s
$GF \rightarrow C$	31.48s	37.26s	40.12s	44.59s	44.05s	45.26s	46.17s	47.57s
$C \rightarrow GF$	65.23s	76.6s	91.2s	96.46s	97.59s	103.8s	106.97s	117.03s

Table 4.1 gives a summary of the performances of appending graphs of various sizes by updating GraphFrames only (GF), both GraphFrames and Cassandra ($GF \rightarrow C$), and updating Cassandra first then loading to GraphFrames ($C \rightarrow GF$). The table displays computation times in seconds for each approach.

In addition to having a good performance, the first approach can be adapted to apply periodic updates to Cassandra on each GraphFrame update since GraphFrames are inherently fault-tolerant. This can greatly decrease the operational cost and processing time which we intend to explore further as a future work.

To prove that Cassandra is needed for OLTP graph queries, we chose two simple queries: looking-up a vertex by Id from the vertex table/RDD. and looking-up the neighbors of a vertex from the edge table/RDD. We used *Graph H* from Table 4.1. Results show in Fig. 4.3 demonstrate a big advantage of using Cassandra for such queries. We also executed PageRank on GraphFrames as an example of OLAP queries. Such algorithms cannot be executed on Cassandra.

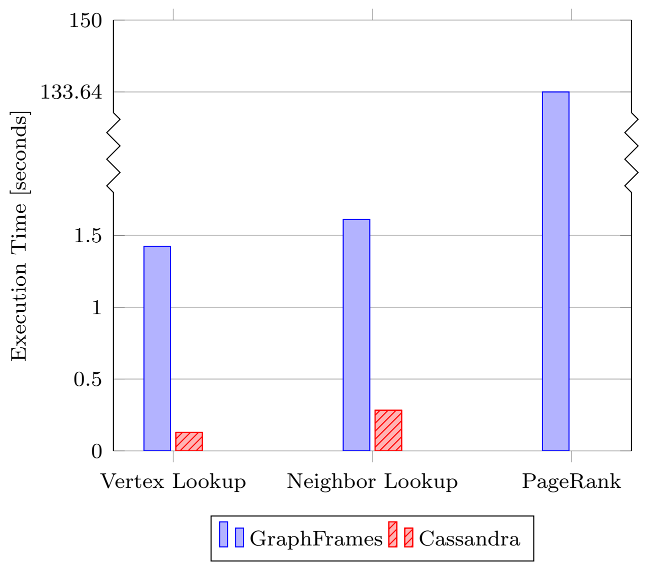


Figure 4.3: Graph queries: GraphFrames vs Cassandra

4.3 Summary

Graph analytics applications where data is streamed continuously require dynamic graph processing in real-time. In this chapter, we proposed Sprouter, a framework to process and store dynamic graphs over data streams using Apache Spark. The framework design has many contributions. In contrast to most graph processing systems which are built to process static graphs, the graph construction is viewed as an incremental process as the graph evolves with time. The system supports executing simple queries as well as complex algorithms on the dynamically updated graph. Also, all components scale to accommodate huge data growth.

The experiments showed that the framework was capable of appending graphs up to 100 million edges in under 50 seconds and allowed efficient pin-point queries. Our framework can be applied to complex streaming data analytics applications that require incremental graphs like recommendation and shortest routes using data from multiple domains such as logs and IoT. The in-memory graph enables more complex multi-level data analytics and knowledge linking for decision support.

Chapter 5

Incremental Community Detection with Sprouter

One of the applications of dynamic graph processing is *community detection* which we present in this chapter as a case study of incremental graph algorithms. Community detection helps to identify groups of nodes that are highly connected among themselves and sparsely connected to the rest of the graph. This problem gets more important for large graphs due to its broad range of applications.

Social networks are everywhere around us from networks of friends to networks of business personnel. A use case scenario of Sprouter is to ingest streams of social data and keep a social graph up-to-date with the stream. The communities in this graph will be found in real-time to allow giving real-time recommendations and to help avoiding the cold-start problem in recommendation engines. In this scenario, detecting the communities is our incremental analytical application, while providing the recommendations in real-time is a graph OLTP query.

In this chapter, we present a distributed implementation of a community detection algorithm based on Weighted Community Clustering (WCC) optimization for static graphs. This implementation is used to detect communities in historical data in Sprouter. We also propose a new algorithm, the Incremental Distributed WCC

(IDWCC). This new algorithm is used to assign newly added vertices to the most suitable communities in a dynamic graph. The algorithm assumes that the graph is undirected and unweighted, the incremental process is done in a node-grained manner, and the generated communities are disjoint. Both algorithms are implemented in Scala using GraphX [41, 15] to work with Spark as the streaming engine [47]. To the best of our knowledge, this is the first distributed node-grained incremental community detection algorithm.

5.1 Background

We chose the WCC metric [28] and its optimization method [30] to be the foundation of our community detection research. Given a graph $G(V, E)$ composed of a set of vertices V and a set of edges E, $t(x, V)$ denotes the number of links between the neighbors of the vertex x or the triangle count, and $vt(x, V)$ denotes the number of neighbor vertices that close at least one triangle with x for each vertex of the graph. Given a community C in G, $t(x, C)$ and $vt(x, V)$ are the same as the previous measurements taking into account the vertices inside C only. Based on these four measurements, the WCC value for a vertex x can be calculated using Equation 5.1

$$WCC(x, C) = \begin{cases} \frac{t(x,C)}{t(x,V)} \cdot \frac{vt(x,V)}{|C\setminus\{x\}|+vt(x,V\setminus C)} & \text{if } t(x, V) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Then, the WCC value for the whole graph is the weighted average of the WCC of all the vertices in the graph as described in Equation 5.2.

$$WCC(G) = \frac{1}{|V|} \sum_{i=1}^{n} WCC(x_i, C) \tag{5.2}$$

We chose the WCC metric since the existing state-of-the-art algorithms to optimize this metric proved to provide a good trade-off between performance and quality [33, 30, 29]. In addition, the optimization of WCC can be distributed easily; the calculations of the best movement and the WCC value for each vertex can be done locally, and thus the computations can be done simultaneously. It is the best solution to our knowledge for community detection in large-scale graphs. Unlike Louvain [2], which we explained in Chapter 3, WCC optimization does not take into account edge weights in the computations. However, the weights do not contribute to communities in many graphs such as social networks. For other use cases such as computer networks, Louvain works better.

Although the Node-Grained Incremental community detection algorithm (NGI) [43] is an incremental solution, it has not been implemented for distributed processing. We preferred WCC optimization since its distributed version has already proven to have good results [33].

The WCC optimization algorithm is explained in details in Prat-Pérez et al. [30, 33]. However, in this section, we will summarize the basic steps. WCC optimization works in three main phases: preprocessing, initial partitioning, and partition refinement.

5.1.1 Preprocessing

This phase aims at calculating $t(x, V)$ and $vt(x, V)$ values for each vertex of the graph. After these measurements are calculated, a graph optimization is performed to reduce the graph size by removing edges that do not close any triangles.

5.1.2 Initial Partitioning

The next step is to compute an initial partition of the graph. First, the vertices are sorted by their clustering coefficients in a descending order. Then, the vertices are iterated on and for each vertex x not previously visited, we create a new community C that contains x and all its neighbors that were not visited before.

The algorithm requires the following conditions to be met in an initial partition:

1. Every community should contain a single center vertex and a set of border vertices connected to the center vertex.

2. The center vertex should be the vertex with the highest clustering coefficient in the community.

3. Given a center vertex x and a border vertex y in a community, the clustering coefficient of x must be higher than the clustering coefficient of any neighbor z of y that is the center of its own community.

5.1.3 Partition Refinement

In this step, the initial partition is improved iteratively using a hill climbing method. The execution stops when no further improvement to the global WCC can be achieved, or when the number of iterations made without a significant improvement exceeds a threshold.

Next, we will discuss our distributed implementation of WCC optimization for GraphX. The proposed IDWCC algorithm is explained after that.

5.2 Distributed WCC for GraphX

To be able to find communities for historical data with Sprouter, and to validate our proposed algorithm, we need an implementation of WCC optimization. There is an available implementation of the centralized WCC optimization version, which is called the Scalable Community Detection (SDC), by the original authors[1]. However, it is written in C++ for symmetric multiprocessing (SMP) systems. A distributed version is also available in Java for the Giraph processing engine[2]. We are looking for an implementation that is distributed and works with Spark, GraphX and/or GraphFrames.

Since no solution was available, we implemented our own version of WCC optimization for GraphX and GraphFrames in Scala for distributed processing on Spark. We refer to this implementation as Distributed WCC (DWCCC). Next, we explain our implementation details for each step of the algorithm.

5.2.1 Preprocessing

The Triangle Count algorithm in GraphX[3] requires the graph to be *canonical* which means that the graph should ensure the following:

- Free from self-edges (edges with the same vertex as a source and a destination).
- All its edges are oriented (the source vertex is greater than the destination vertex using a pre-defined comparison method).
- Has no duplicate edges.

[1]`https://github.com/DAMA-UPC/SCD`

[2]`https://github.com/saltzm/distributed_wcc`

[3]`https://spark.apache.org/docs/latest/graphx-programming-guide.html#triangle-counting`

The optimization is done using the *subgraph* API provided by GraphX. We keep the calculated statistics, namely the triangle count and the vertex degree, as they are used later. We take advantage of the fact that GraphX supports property graphs and hence we can save these statistics as properties of the graph vertices.

5.2.2 Initial Partitioning

The *Pregel* API in GraphX helps in executing the partitioning while respecting all the initial partitioning conditions we mentioned earlier. In the first superstep, each vertex sends its clustering coefficient (cc) to all its neighbors. In the second superstep, each vertex chooses the vertex with the highest cc from the messages it received (including its own cc) as its community center. Then the vertices send their choices to their neighbors to ensure that a vertex does not choose a vertex as its community center while that vertex is a member of another community. In that case, the vertex has to change its community. The changes are propagated until no further adjustments are needed.

5.2.3 Partition Refinement

Computing the improvement of the global WCC using Equation 5.2 requires the computation of the internal triangles of each community of the graph, which makes it really inefficient to compute for all possible movements of each vertex. Prat-Pérez et al. [30] present a heuristic for calculating WCC improvement caused by moving a single vertex to a new community using statistics about the vertex and its neighboring communities. The heuristic, which can be seen in Equation 5.3, gives an approximated value and does not require the computation of the internal triangles of each community. Instead, it depends on calculating the following statistics: d_{in} the number of edges that connect the vertex v to vertices inside the community C where it is moving, d_{out} the number of edges that connect v to vertices outside C, b the number of edges that are in the boundary of C, δ the edge density of C, r the number of vertices in C, and ω the clustering coefficient of the graph.

$$WCC_I^{'}(x, C) = \frac{1}{|V|}.\,(d_{in}.\Theta_1 + (r - d_{in}).\Theta_2 + \Theta_3) \tag{5.3}$$

where :

$$\begin{aligned}
\Theta_1 &= \frac{(r-1)\delta + 1 + q}{(r+q)((r-1)(r-2)\delta^3 + (d_{in}-1)\delta + q(q-1)\delta\omega + q(q-1)\omega + d_{out}\omega)}. \\
&\quad (d_{in}-1)\delta; \\
\Theta_2 &= -\frac{(r-1)(r-2)\delta_3}{(r-1)(r-2)\delta^3 + q(q-1)\omega + q(r-1)\delta\omega}.\frac{(r-1)\delta + q}{(r+q)(r-1+q)}; \\
\Theta_3 &= \frac{+d_{out}(d_{out}-1}{d_{in}(d_{in}-1) + d_{out}(d_{out}-1)\omega + d_{out}d_{in}\omega}.\frac{d_{in}+d_{out}}{r+d_{out}}; \\
q &= (b - d_{in})/r;
\end{aligned}$$

We use the same heuristic due to its efficiency. Since this computation occurs independently within each vertex, all vertices may perform their movements simultaneously, meaning that this part of the algorithm can be distributed effectively.

5.3 Incremental Distributed WCC for GraphX

In this work, We limit our scope of interest to dynamic graphs that satisfy two properties: First, the graph progresses over a window of time in which a small batch of vertices and their edges are added. These edges connect the new vertices to each other and to the full graph generated from the last micro-batch. Second, the edges are equal in value i.e. the edges are not weighted or directed.

We denote the graph from the previous iteration as $G_t = \{V_t, E_t\}$ where V_t and E_t are its sets of vertices and edges at time t respectively. Let's refer to the vertices in the newly arriving batch as V^* and the edges as E^*. We define a micro-batch from the stream of a node-grained dynamic graph δ^* as follows:

$$\delta^* = V^* \cup E^* \text{ where } \forall e_{j,k} \in E^* : v_j \in V^* \setminus V_t \vee v_k \in V^* \setminus V_t \tag{5.4}$$

In this section, we propose IDWCC, an algorithm for node-grained dynamic graphs. The algorithm has many similar steps as DWCC. However, it has optimizations that avoid repeated calculations while processing previous micro-batches and consequently reduce the memory, data movement, and computational costs without sacrificing the quality of the result as we prove in our evaluation. We calculated the execution time for each small step of DWCC as shown in Fig. 5.2. Based on these calculations, we developed an algorithm that works in three phases. First, it merges the batch with the maintained evolving graph, updates the vertex statistics, and optimizes the graph. Second, it assigns the new vertices to initial communities. Finally, it optimizes the WCC metric to generate better communities.

5.3.1 The Merging Phase

As a first step, a new graph $G^* = \{V^*, E^*\}$ is generated from the newly arrived batch δ^*. The produced graph is then merged with the full graph to produce $G_{t+1} = \{V_t \cup V^*, E_t \cup E^*\}$ as demonstrated in Fig. 5.1.

For optimization purposes, we identify a set of vertices we call the *border vertices.* These vertices exist in both G_t and G^* and are a part of the edges that connect the

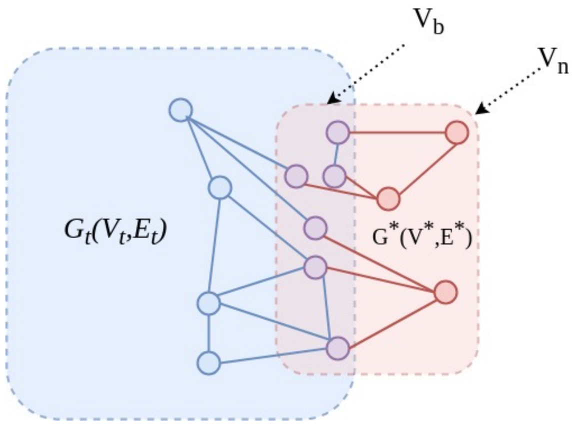


Figure 5.1: Merging G^* with G_t

newly arriving batch with the old graph. Let's denote this set as $V_b = V_t \cap V^*$. We refer to the rest of the vertices in the new graph which are not part of the border vertices as the *inner vertices* $V_n = V^* \setminus V_b$.

The problem with the border vertices is that they have already been assigned to communities in G_t. But since they have new connections, now they are likely to belong to different communities. We isolate each of these vertices in its own community in the full graph G_{t+1}.

The merge phase also calculates $t(x, V_{t+1})$ and $vt(x, V_{t+1})$ for each vertex x in G_{t+1}. To perform the calculations efficiently, we recognize three situations: (a) The old vertices statistics stay the same as they were in the previous micro-batch t. (b) The inner vertices need to calculate the statistics. (c) The border vertices have new connections and thus might belong to new triangles and need to update their statistics.

The definition of the stream batches, which is presented in 5.4, is important for updating the statistics of the border vertices as it assures that the graph holds the following. Let's denote the set of triangles that pass through a vertex x in graph G

as $T_{x,G}$ and the set of vertices that form at least one triangle with x as $VT_{x,G}$. Then the following holds true:

$$T_{x,G_t} \cap T_{x,G^*} = \emptyset \text{ and } VT_{x,G_t} \cap VT_{x,G^*} = A$$

Where A is the set of vertices that are neighbors of x and form triangles with it in both G_t and G^*. Based on these statements, the statistics for the border vertices are calculated as follows:

$$t(x, G_{t+1}) = t(x, G_t) + t(x, G^*) \tag{5.5}$$

$$vt(x, G_{t+1}) = vt(x, G_t) + vt(x, G^*) - |A| \tag{5.6}$$

Using these two measurements we can compute the local clustering coefficient for each vertex and the global clustering coefficient ω which is needed to calculate WCC'_I.

At the end of this phase, we optimize the graph in the same way as it is done in DWCC to reduce the memory consumption and the processing required in the succeeding phases. Furthermore, it is a relatively cheap operation as can be seen in Fig. 5.2.

5.3.2 Initial Partitioning

In this phase, we choose communities for the vertices that appear in the new batch. These vertices include the inner vertices V_n which have no communities assigned to them yet, and the border vertices V_b which were removed from their communities during the previous phase. We use the same algorithm from DWCC but we limit it to the above mentioned sets of vertices only. Hence, every vertex in the new batch chooses the vertex with the highest clustering coefficient that does not belong to a community of another vertex as its community center.

5.3.3 Partition Refinement

This phase follows the same steps as its counterpart in the DWCC algorithm. However, it includes two optimizations since it is the most expensive phase in terms of computations:

1. Calculations of the community movements are still done on all the vertices, but we drop calculating the value of WCC on each iteration.

2. Extra iterations are not performed when good WCC improvements appear. This might result in missing community movements that can have a good impact on WCC. However, as we process subsequent micro-batches, all the vertices start changing their communities again and any previous changes that were missed should be recovered. This way, the degradation of WCC overtime is avoided.

5.4 Complexity Analysis

In this section, We aim to compare the complexity of a sequential implementation of our incremental algorithm to its static counterpart when both are applied to detect communities in a dynamic graph.

Let n be the number of vertices and m the number of edges in the graph. We assume that the average degree of the graph d at any point of time t is $d = m/n$ and that real graphs have a quasi-linear relation between vertices and edges $O(m) = O(n \cdot \log n)$. Under these assumptions, The complexity of the static WCC optimization algorithm, as calculated in Prat-Perez et al. [30], is $O(m \cdot \log n)$.

Let's now calculate the complexity of a centralized version of the sequential incremental WCC optimization algorithm. For the first phase, we do not consider the complexity of merging the graphs since the operation is necessary for any dynamic graph. That leaves the cost of computing the triangles and degrees for the vertices in the new batch $O(m^* \cdot d + m^*) = O(m^* \cdot \log n_{t+1})$.

The second phase requires sorting the vertices based on the local clustering coefficient. However, the vertices are already sorted from processing a previous micro-batch. Hence, the cost is only for organizing the new vertices in the right order which requires sorting the new vertices and then executing a full scan of the vertices in the worst case $O(n^* + n_t) = O(n_t)$

For the third phase, let α be the number of iterations required to find the best possible communities, which is a constant. In each iteration, we compute in the worst case $d + 1$ movements for each vertex of type WCC'_I which has a cost $O(1)$. That makes the total cost $O(n \cdot (d+1)) = O(m)$. Then we apply all the movements which are equal to the number of vertices so it costs $O(n_{t+1})$. We also need to update, for each iteration in the second phase, the statistics ω, *cout*, *din*, and *dout* for each vertex and community, which has a cost of $O(m_{t+1})$. We sum all the costs to get the full cost of this phase $O(\alpha \cdot (m_{t+1} + n_{t+1} + m_{t+1})) = O(m_{t+1})$.

The full cost of the algorithm is the sum of the cost of the three phases: $O(m^* \cdot \log n_{t+1} + n_t + m_{t+1})$. Since $m^* \ll n_{t+1}$, then $m^* \cdot \log n_{t+1} \ll n_{t+1} \cdot \log n_{t+1} < m_{t+1}$. The cost can be simplified to become $O(m_{t+1})$. This cost is much smaller than $O(m_{t+1} \cdot \log n_{t+1})$ the cost of applying the static algorithm on the whole graph on iteration t.

5.5 Experiments

In this section we present our experimental study which included three sets of experiments: (a) Computing the efficiency of the optimizations used in IDWCC by comparing the execution time of each step with its counterpart in DWCC. (b) Comparing the quality of the results of DWCC, IDWCC, and SCD. (c) Comparing the quality of the results and the execution time for the DWCC and IDWCC algorithms over time when applied to a data stream.

5.5.1 The Environment

To test our system, we used 8 identical machines each with 8 cores 2.10 GHz Intel Xeon 64-bit CPU, 30 GB of RAM, and 300 GB of disk space. We installed Apache Spark v 2.2.0 on all the machines. Both algorithms were implemented in Scala 2.11 and the code is publicly available on Github[4].

We used 5 as the number of iterations that both DWCC and IDWCC execute. This number was recommended by the original authors of the original WCC optimization algorithm.

5.5.2 Data Source

For experimentations, we chose to use a set of different real-life undirected graphs that have ground-truth communities. We took these graphs from the SNAP graph repository[5]. The chosen graphs and some statistics about them can be found in Table 5.1.

[4]`https://github.com/TariqAbughofa/incremental_distributed_wcc`

[5]`http://snap.stanford.edu/`

Table 5.1: Properties of the test graphs

	Vertices	Edges	Communities
Amazon	334,863	925,872	75,149
DBLP	317,080	1,049,866	13,477
YouTube	1,134,890	2,987,624	8,385
LiveJournal	3,997,962	34,681,189	287,512

Table 5.2: Streaming information about the test graphs

	Bulk Vertices	Bulk Edges	Stream Edges	# of Micro-batches
Amazon	258,464	576,718	349,154	10
DBLP	253,119	852,754	197,112	10
YouTube	903,959	2666,836	320,788	10
LiveJournal	768,792	13,997,342	20,683,847	30

After loading a graph from the experimental sets, we make sure to clean it up of duplicate edges and self-edges. We also noticed that the edges are sorted by the source vertices which makes the graph canonical from the start and ready for processing.

5.5.3 Experimental Details and Results

Using multiple graphs for the experiments allows us to compare the results for different graph sizes. In addition, it gives us the ability to experiment with different sizes of micro-batches easily. Table 5.2 shows the choices we made for each graph regarding the size of the bulk graph (the initial graph size before consuming from the data stream), the stream size, the micro-batch number and size. It can be noticed that we chose to have smaller bulk graph and micro-batches, and greater numbers of micro-batches for large graphs to limit the use of the resources on each iteration.

The first experiment shows the benefits of the optimization techniques we applied in our IDWCC algorithm, which resulted in shorter execution times compared to the DWCC algorithm. We calculated the time that each step in both DWCC and IDWCC took to be executed on the full Amazon graph and we summed up the results

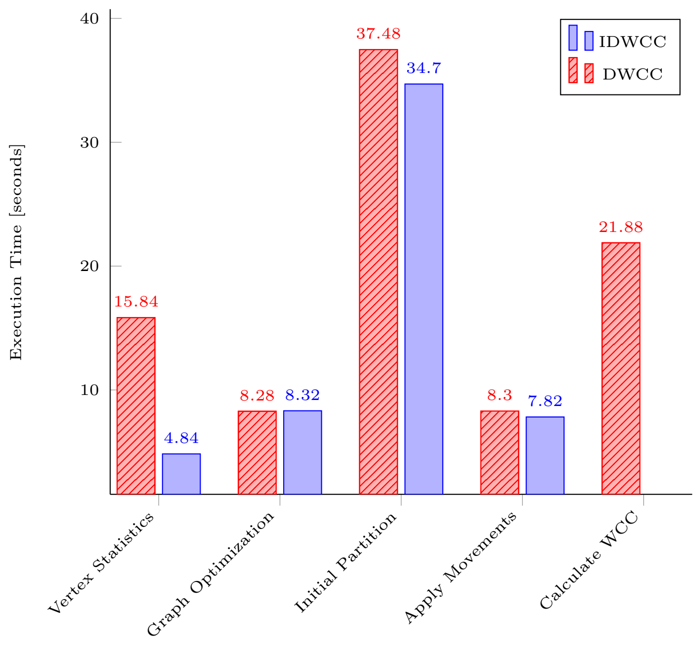


Figure 5.2: Execution Time for Each Step in DWCC vs IDWCC.

in Fig. 5.2. The *vertex statistics* step is the one responsible for calculating each vertex triangle count and degree. We noticed that the execution time was reduced in IDWCC almost three times as we applied it only on the new vertices in addition to updating the border vertices. The *graph optimization* and *apply movements* steps had no change in execution time as there were no adjustments were done to them. The *initial partition* step had a small decrease in execution time caused by the way we altered this stage. However, the biggest gain was expected in memory consumption as proven later. Finally, the *calculate WCC* step, which is highly expensive, was eliminated completely from the IDWCC algorithm and thus its cost was saved.

The second set of experiments aimed at proving the result quality of the DWCC and IDWCC algorithms. SDC already proved that it has better quality than many other centric community detection algorithms while having faster execution [30]. The

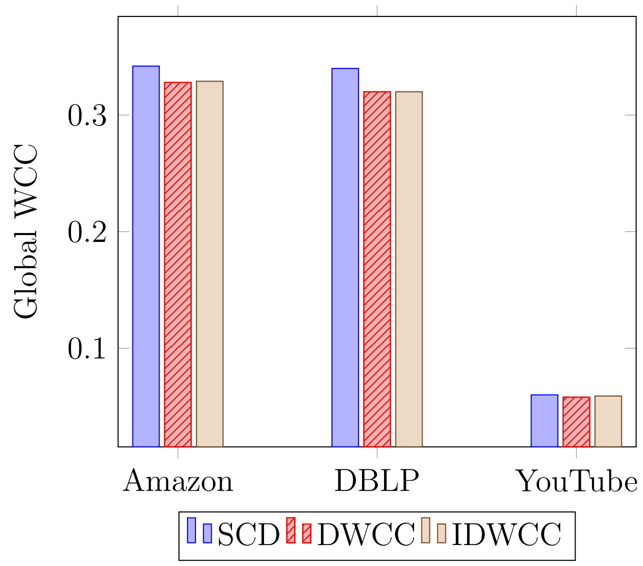


Figure 5.3: Global WCC: SCD vs DWCC vs IDWCC

experiments compared SDC to algorithms like Infomap [32] and Louvain [2]. Therefore, we compared DWCC and IDWCC to SCD. The comparison was done by calculating the global WCC on the test graphs described earlier when the last micro-batch was added to the graph. In other words, the algorithms were applied to the full graph. The results, which are displayed in Fig. 5.3, show that both DWCC and IDWCC produce good WCC values. These WCC values show only up to 5% decrease from their SCD counterparts. On top of that, IDWCC gives slightly better results than DWCC. It can be observed that we do not show the results for the LiveJournal graph as both DWCC and IDWCC failed to process the whole stream due to memory consumption that exceeded the available resources.

We also conducted another set of experiments with the goal of proving the quality of the results and the efficiency of the IDWCC algorithm. For each graph and each micro-batch in the stream as described in Table 5.2, we merged the micro-batch with the full graph. Then we applied both DWCC and IDWCC on the generated graph.

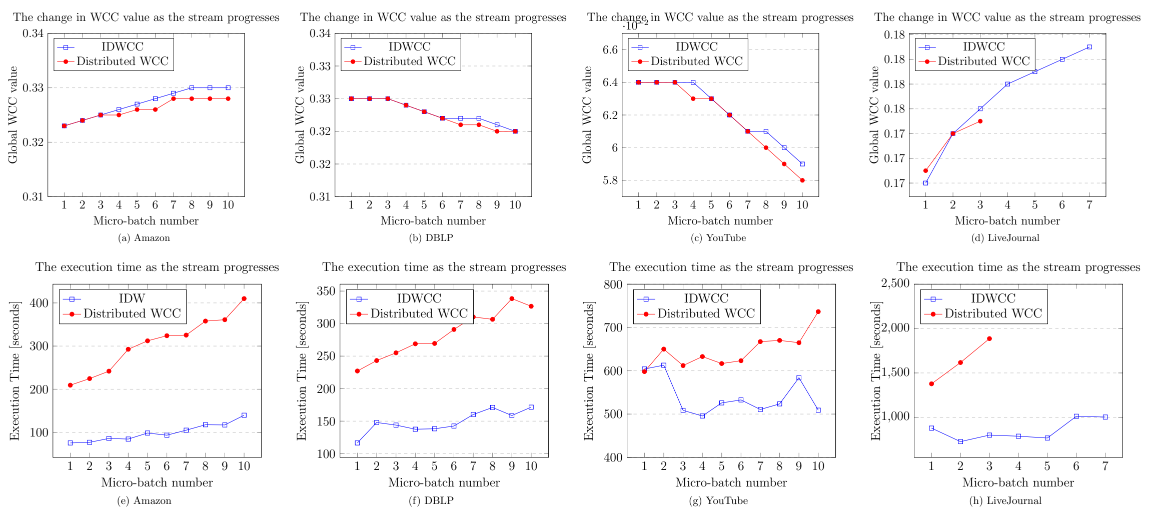


Figure 5.4: Comparing WCC values and execution time IDWCC vs DWCC

In this streaming context, DWCC finds the communities for the whole graph again, whereas IDWCC finds communities for the new vertices and reflects these changes on the old vertices. For each micro-batch, we compared the global WCC values of the final results generated by IDWCC and DWCC in addition to the execution times.

Fig. 5.4 shows the results of the data streaming experiments. Missing data points indicate that the algorithm was not able to continue processing the graph due to memory consumption that exceeded the available resources. We can easily notice that IDWCC produces communities with global WCC values that are very close to the ones produced by DWCC. We can even see that the results start to be better than DWCC in later iterations. Regarding the execution time, IDWCC performed two to three times better than DWCC. For the LiveJournal graph, we can see that both algorithms failed to continue with the available resources. However, IDWCC continued for 7 micro-batches before it crashed. While DWCC could only work for 3 micro-batches. This shows that IDWCC has significantly less memory consumption than DWCC.

5.6 Case Study

We further examine the communities formed by IDWCC in a real-world application that can be applied to our previously proposed framework, Sprouter. The application we chose is *product purchase recommendation* and for that, we used the Amazon dataset one of the experimental datasets. The metadata for this graph is available on the SNAP website and contains the titles of the products.

The product recommendation problem aims to find products that are usually bought together to suggest them to the users. This graph represents a network of products, where each vertex is a product and an edge connects two products if they have been co-purchased frequently. Therefore, the communities in the Amazon graph should contain similar products that can be recommended together.

We ran the incremental algorithm on the graph and we chose three communities which are reported in Table 5.3. The table has 10 randomly selected vertices from each community. In the case of the first community, we see that it is formed of classic novels. The second community consists of Shakespearean literature. Finally, the third one is mostly political and allegorical novels. We observe that the algorithm is able to perform a good selection of the relations in the graphs in order to give meaningful communities.

5.7 Summary

In this chapter, we studied community detection as an example of incremental graph algorithms and a case study of Sprouter. We presented IDWCC, a distributed algorithm for community detection over node-grained dynamic graphs based on the WCC metric. The algorithm and its static counterpart (DWCC) were implemented using

Table 5.3: Examples of communities of Amazon products produced by IDWCC.

community #1	Gulliver's Travels	Robinson Crusoe: His Life and Strange Surprising Adventures
	Science Fiction Classics of H.G. Wells	Treasure Island
	Swiss Family Robinson	Gulliver's Travels
	The War of the Worlds	The Swiss Family Robinson
	Anne of Avonlea	Robinson Crusoe: Life and Strange Surprizing Adventures
community #2	Merchant of Venice	Hamlet: The New Variorum Edition
	The Merchant of Venice	Hamlet
	Macbeth	The Merchant of Venice
	Othello : The Applause Shakespeare Library	A Midsummer Night's Dream
	Much Ado About Nothing	Othello
community #3	1984	To Kill a Mockingbird
	A Separate Peace	John Knowles's a Separate Peace
	Lord of the Flies	Joseph Heller's Catch-22
	Romeo and Juliet	The Grapes of Wrath
	1984	1984

GraphX and tested with Spark Streaming. The experiments showed that IDWCC outperforms DWCC for large-scale dynamic graphs. IDWCC produced the same or better WCC values compared to DWCC. It was also two to three times faster than DWCC. The memory consumption was more optimized in IDWCC as well. We conclude that, to our knowledge, IDWCC is the best performing incremental community detection algorithm for node-grained dynamic graphs. We also demonstrated how the detected communities can be used within Sprouter to give recommendation in an e-commerce setting.

Chapter 6

Conclusions and Future Work

In this work, we explored cutting-edge big data processing technologies to build a streaming data processing framework that facilitate real-time data extraction and linking in a graph structure and serves OLTP and OLAP queries in a timely manner. We demonstrated the framework functionalities and applications using a case study of social data analytics.

We started by conducting a thorough literature study on cutting-edge stream processing engines, graph processing, and graph storage systems in Chapter 2. We also discussed the problem of incremental graph processing and community detection as an example of graph analytics.

We selected Apache Spark to build a solution for real time data processing since it is one of the most advanced streaming engines, an open-source solution, offers both bulk and streaming APIs, and supports large-scale graph processing. Based on this choice, we studied maintaining a dynamic graph on top of Apache Spark. We chose IndexedRDDs and Redis as possible solutions in chapter 3. We compared the two to RDDs which is the basic storage abstraction in GraphX. The experiments showed that IndexedRDDs, although provide fast fine-grained updates, retrieving the inserted values proved to be slow. Redis was more efficient but still has no support for storing property graphs.

In Chapter 4, we built on the results we got in Chapter 3 to design Sprouter, a framework for dynamic graph processing that supports OLTP/OLAP queries. We chose Cassandra as a graph store with Spark as it is a NoSQL solution with fast read/write performance and property graphs storage support. We presented three approaches to creating, storing and querying information from the graph storage: a) dynamically updating and querying the in-memory graph in GraphX, b) updating the in-memory graph first and then persisting the changes in Cassandra, or c) persisting the data in Cassandra then reloading the full graph to the in-memory graph structure. The experimental results showed that the second approach is the most efficient while depending on Cassandra for pinpoint or OLTP queries and GraphX for analytical queries. After implementing a full framework that is capable of ingesting data streams and processing them in real-time, we conducted many experiments to measure the efficiency of the solution. The experiments showed that the framework is capable of maintaining graphs up to 100 million edges while updating them with stream data in under 50 seconds. It also allowed efficient pinpoint queries and fast analytics.

Finally, we showcased the framework applications with a case study on incremental graph algorithms by exploring the analytical problem of community detection in large distributed dynamic graphs. We chose WCC optimization as a static graph algorithm to extend into an incremental community detection method. As described in chapter 5, we implemented WCC optimization for Spark's GraphX in addition to a new incremental WCC optimization which we called Incremental Distributed WCC (IDWCC). We compared the two as the original WCC optimization was already tested against state-of-the-art community detection algorithms and showed superiority in terms of efficiency and performance. The comparison results showed that IDWCC was more efficient in terms of computational cost and memory consumption and it was two to three times faster than DWCC in all our experiments. In terms of the quality of results, IDWCC performed the same or better than its static counterpart, DWCC, in all experiments.

6.1 Future Work

Fast processing and analytics of streaming data is increasingly becoming an important problem as we need recent data to be reflected in the insights instantly. We describe our future work directions based on the limitations that we observed in current approaches, possible enhancements, and potential new problem areas that we need to address.

6.1.1 Exploring Other Graph Storage Systems

We plan to investigate more data storage systems. One group we are interested in exploring is in-memory SQL data stores such as MemSQL[1], a distributed, highly-scalable SQL database that supports both OLTP and OLAP queries, and SnappyData [25] which embeds an in-memory transactional database into Spark to avoid serialization/deserialization time. We also intend to explore graph data stores such as OrientDB, Neo4J, and Titan.

The results of this research will allow us to determine the most robust scalable storage for graphs to use within Sprouter. We aim to make it perform faster updates

[1]`https://www.memsql.com/`

and support a wider range of OLTP queries efficiently.

6.1.2 Validating IDWCC for Very Long Runs

We want to study the stability of the results of IDWCC over long periods of time. In the case of result degradation, one possible approach to overcome it might be to apply the static algorithm (DWCC) periodically on the full graph to maintain high accuracy of the results.

6.1.3 Alleviate the Memory Consumption of IDWCC

We also aim to address the memory consumption of the IDWCC algorithm which causes a bottleneck for the computation of new communities for each vertex. We plan to further optimize the iteration phase by limiting the number of vertices for which we change communities on each iteration by using statistics calculated from the previous iteration.

6.1.4 Expand the Scope of IDWCC

We only addressed undirected unweighted node-grained dynamic graphs in this research. In the future, we like to extend our framework to work with edge-grained dynamic graphs. Another area to explore is adding the edge weights to the community detection process.

Bibliography

[1] James P Bagrow. Communities and bottlenecks: Trees and treelike networks have high modularity. *Physical Review E*, 85(6):066118, 2012.

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[3] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006.

[4] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. AcM, 2008.

[5] Hsinchun Chen, Xin Li, and Zan Huang. Link prediction approach to collaborative filtering. In *Digital Libraries, 2005. JCDL'05. Proceedings of the 5th ACM/IEEE-CS Joint Conference on*, pages 141–142. IEEE, 2005.

[6] Xi Chen, Huajun Chen, Ningyu Zhang, Jue Huang, and Wen Zhang. Large-scale real-time semantic processing framework for internet of things. *International Journal of Distributed Sensor Networks*, 11(10):365372, 2015.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[8] Sutanay Choudhury, Khushbu Agarwal, Sumit Purohit, Baichuan Zhang, Meg Pirrung, Will Smith, and Mathew Thomas. Nous: Construction and querying of dynamic knowledge graphs. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1563–1565. IEEE, 2017.

[9] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2016.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Paul Dourish and Matthew Chalmers. Running out of space: Models of information navigation. In *Short paper presented at HCI*, volume 94, pages 23–26, 1994.

[12] Guillaume Durand, Nabil Belacel, and François LaPlante. Graph theory based model for learning path recommendation. *Information Sciences*, 251:10–21, 2013.

[13] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. 14:599–613, 2014.

[15] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[16] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: advances, systems and applications*, 2(1):22, 2013.

[17] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. pages 505–514, 2013.

[18] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 5. ACM, 2016.

[19] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[20] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. pages 38–49, 2013.

[21] Azi Lipshtat, Susana R Neves, and Ravi Iyengar. Specification of spatial relationships in directed graphs of cell signaling networks. *Annals of the New York Academy of Sciences*, 1158(1):44–56, 2009.

[22] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. pages 135–146, 2010.

[23] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[24] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[25] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. Snappydata: A unified cluster for streaming, transactions and interactice analytics. In *CIDR*, 2017.

[26] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[27] Gang Pan, Wangsheng Zhang, Zhaohui Wu, and Shijian Li. Online community detection for large complex networks. *PloS one*, 9(7):e102799, 2014.

[28] Arnau Prat-Pérez, David Dominguez-Sal, Josep M Brunat, and Josep-Lluis Larriba-Pey. Shaping communities out of triangles. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1677–1681. ACM, 2012.

[29] Arnau Prat-Pérez, David Dominguez-Sal, Josep-M Brunat, and Josep-Lluis Larriba-Pey. Put three and three together: Triangle-driven community detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 10(3):22, 2016.

[30] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd international conference on World wide web*, pages 225–236. ACM, 2014.

[31] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[32] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[33] Matthew Saltz, Arnau Prat-Pérez, and David Dominguez-Sal. Distributed community detection with the wcc metric. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1095–1100. ACM, 2015.

[34] Jiaxing Shang, Lianchen Liu, Feng Xie, Zhen Chen, Jiajia Miao, Xuelin Fang, and Cheng Wu. A real-time detecting algorithm for tracking community structure of dynamic networks. *arXiv preprint arXiv:1407.2683*, 2014.

[35] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. Graphjet: real-time content recommendations at twitter. *Proceedings of the VLDB Endowment*, 9(13):1281–1292, 2016.

[36] Michael Stonebraker, Uur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[37] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[38] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[39] Tengjiao Wang, Bishan Yang, Jun Gao, Dongqing Yang, Shiwei Tang, Haoyu Wu, Kedong Liu, and Jian Pei. Mobileminer: a real world case study of data mining in mobile communication. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1083–1086. ACM, 2009.

[40] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[41] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[42] Bing Yao, Xia Liu, Wan-jia Zhang, Xiang-en Chen, Xiao-min Zhang, Ming Yao, and Zheng-xue Zhao. Applying graph theory to the internet of things. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 2354–2361. IEEE, 2013.

[43] Siwen Yin, Shizhan Chen, Zhiyong Feng, Keman Huang, Dongxiao He, Peng Zhao, and Michael Ying Yang. Node-grained incremental community detection for streaming networks. In *IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 585–592. IEEE, 2016.

[44] Matei Zaharia. Continuous applications: Evolving streaming in apache spark 2.0. `https://databricks.com/blog/2016/07/28/continuous-applications-evolving-streaming-in-apache-spark-2-0.html`, 2016.

[45] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient

distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[46] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. pages 423–438, 2013.

[47] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[48] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. 2002.

Appendix A

The Ingestion Component Implementation for GDELT

A.1 Ingesting Historical Data

Fig. A.1 shows the historical data pipeline as it is implemented in NiFi. The pipeline consists of six processing steps.

1. Fetch History File: A "GetHTTP" processor that initiates an HTTP request to fetch the master file.

2. Split File To Lines: A "Split Text" processor that split the master file into lines.

3. Parse GKG File URL: Parses each line of the master file and extracts URLs of the GKG files only.

4. Fetch GKG File: Another "GetHTTP" processor that initiates an HTTP request for each GKG file URL.

5. Update Attribute: Add an attribute "filename" to each file which represents the name of the file.

6. Save To HDFS: Saves the file as is into HDFS using the name in the attribute.

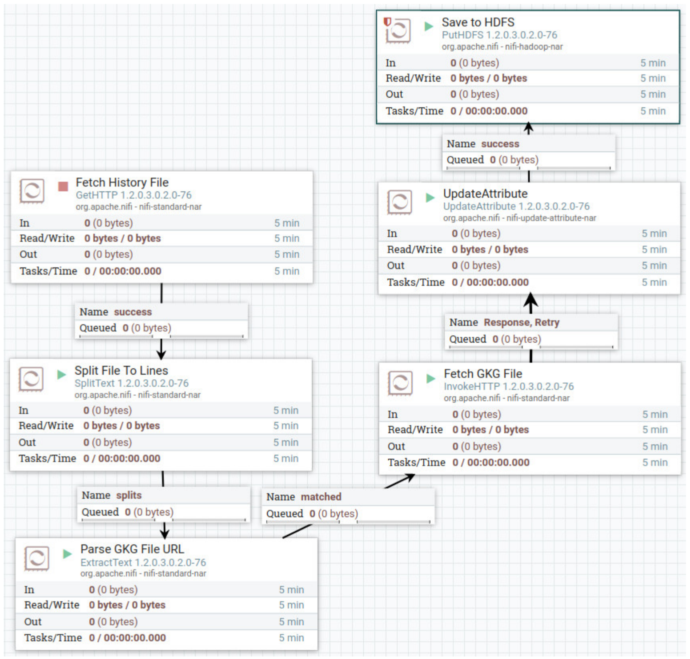

Save to HDFS
PutHDFS 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-hadoop-nar
Name success
Queued 0 (0 bytes)
UpdateAttribute
UpdateAttribute 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-update-attribute-nar
Name Response, Retry
Queued 0 (0 bytes)
Fetch GKG File
InvokeHTTP 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
Fetch History File
GetHTTP 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
Name success
Queued 0 (0 bytes)
Split File To Lines
SplitText 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
Name splits
Queued 0 (0 bytes)
Name matched
Queued 0 (0 bytes)
Parse GKG File URL
ExtractText 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
In 0 (0 bytes) 5 min
Read/Write 0 bytes / 0 bytes 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 / 00:00:00.000 5 min

Figure A.1: The NiFi Pipeline for Historical Data

A.2 Ingesting New Data

The real-time data ingestion from the latest file consists of five processors as listed below and is shown in Fig. A.2.

1. Fetch Latest File: A “GetHTTP” processor that initiates an HTTP request to fetch the latest file.

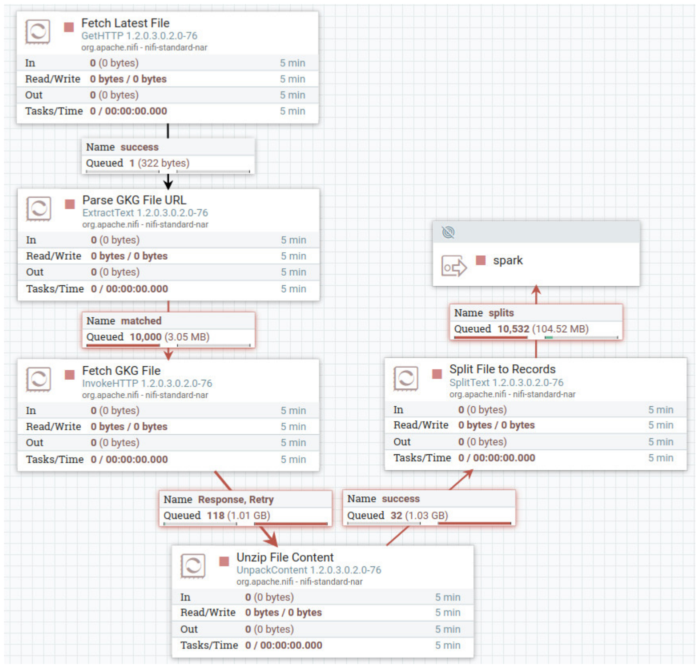
2. Parse GKG File URL: Parses the latest file and extracts URLs of the GKG files only.

3. Fetch GKG File: Another “GetHTTP” processor that initiates an HTTP request for each GKG file URL extracted from the latest file.

4. Unzip File Content: Unzips the GKG file content since it is in “zip” format.

5. Split Files to Records: Splits each GKG file into lines.

6. Spark: An “Output Port” to which NiFi directs the output of the last processor and from which Spark then reads the data.



```
Fetch Latest File
GetHTTP 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
In            0 (0 bytes)            5 min
Read/Write    0 bytes / 0 bytes      5 min
Out           0 (0 bytes)            5 min
Tasks/Time    0 / 00:00:00.000       5 min

Name    success
Queued  1 (322 bytes)

Parse GKG File URL
ExtractText 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
In            0 (0 bytes)            5 min
Read/Write    0 bytes / 0 bytes      5 min
Out           0 (0 bytes)            5 min
Tasks/Time    0 / 00:00:00.000       5 min

Name    matched
Queued  10,000 (3.05 MB)

Fetch GKG File
InvokeHTTP 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
In            0 (0 bytes)            5 min
Read/Write    0 bytes / 0 bytes      5 min
Out           0 (0 bytes)            5 min
Tasks/Time    0 / 00:00:00.000       5 min

Name    Response, Retry
Queued  118 (1.01 GB)

Unzip File Content
UnpackContent 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
In            0 (0 bytes)            5 min
Read/Write    0 bytes / 0 bytes      5 min
Out           0 (0 bytes)            5 min
Tasks/Time    0 / 00:00:00.000       5 min

Name    success
Queued  32 (1.03 GB)

Split File to Records
SplitText 1.2.0.3.0.2.0-76
org.apache.nifi - nifi-standard-nar
In            0 (0 bytes)            5 min
Read/Write    0 bytes / 0 bytes      5 min
Out           0 (0 bytes)            5 min
Tasks/Time    0 / 00:00:00.000       5 min

Name    splits
Queued  10,532 (104.52 MB)

spark
```

Figure A.2: The NiFi pipeline for Real-time Data

Appendix B

The IDWCC Algorithm

Algorithm 1 Phase 2, Initial Partition

```
1: Let $P$ be a set of communities generated at the last micro-batch;
2: $S \leftarrow sortByClusteringCoefficient(V_{t+1})$;
3: for all $v$ in $S$ do
4:     if $notVisited(v)$ then
5:         $markAsVisited(v)$;
6:         if $v \in V^*$ then
7:             $C \leftarrow \{v\}$;
8:         else
9:             $C \leftarrow P.getCommunity(v)$;
10:        for all $u$ in $neighbors(v)$ do
11:            if $notVisited(u)$ then
12:                $markAsVisited(u)$;
13:                if $u \in V^*$ then
14:                    $C.add(u)$;
15:        $P.add(C)$;
```

Algorithm 2 Phase 3, Refinement

```
1: Let $P$ be the initial partition;
2: $iteration \leftarrow 1$;
3: **repeat**
4:     $M \leftarrow \phi$
5:     **for all** $v$ in $V$ **do**
6:         $M.add(bestMovement(v, P))$;
7:     $P \leftarrow applyMovements(M, P)$;
8:     $iteration = iteration + 1$;
9: **until** $iteration > maxIterations$;
```