Co-funded by the
Erasmus+ Programme
of the European Union

Scalable Streaming Graph and Time Series Analysis Using Partitioning and Machine Learning

ZAINAB ABBAS

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
and
Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium, 2021

School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2021:55
ISBN: 978-91-7873-979-0

Akademisk avhandling som med tillstånd av Kungliga Tekniska Högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i informations- och kommunikationsteknik på måndagen den 4 oktober 2021 kl. 14:00 i Sal C, Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista.

Printed by Universitetsservice US-AB

To Abbas Ibn Ali

Abstract

Recent years have witnessed a massive increase in the amount of data generated by the Internet of Things (IoT) and social media. Processing huge amounts of this data poses non-trivial challenges in terms of the hardware and performance requirements of modern-day applications. The data we are dealing with today is of massive scale, high intensity and comes in various forms. MapReduce was a popular and clever choice of handling big data using a distributed programming model, which made the processing of huge volumes of data possible using clusters of commodity machines. However, MapReduce was not a good fit for performing complex tasks, such as graph processing, iterative programs and machine learning. Modern data processing frameworks, that are being popularly used to process complex data and perform complex analysis tasks, overcome the shortcomings of MapReduce. Some of these popular frameworks include Apache Spark for batch and stream processing, Apache Flink for stream processing and Tensor Flow for machine learning.

In this thesis, we deal with complex analytics on data modeled as time series, graphs and streams. Time series are commonly used to represent temporal data generated by IoT sensors. Analysing and forecasting time series, i.e. extracting useful characteristics and statistics of data and predicting data, is useful for many fields that include, neuro-physiology, economics, environmental studies, transportation, etc. Another useful data representation we work with, are graphs. Graphs are complex data structures used to represent relational data in the form of vertices and edges. Graphs are present in various application domains, such as recommendation systems, road traffic analytics, web analysis, social media analysis. Due to the increasing size of graph data, a single machine is often not sufficient to process the complete graph. Therefore, the computation, as well as the data, must be distributed. Graph partitioning, the process of dividing graphs into subgraphs, is an essential step in distributed graph processing of large scale graphs because it enables parallel and distributed processing.

The majority of data generated from IoT and social media originates as a continuous stream, such as series of events from a social media network, time series generated from sensors, financial transactions, etc. The stream processing paradigm refers to the processing of data streaming that is continuous and possibly unbounded. Combining both graphs and streams leads to an interesting and rather challenging domain of streaming graph analytics. Graph streams refer to data that is modelled as a stream of edges or vertices with adjacency lists representing relations between entities of continuously evolving data generated by a single or multiple data sources. Streaming graph analytics is an emerging research field with great potential due to its capabilities of processing large graph streams with limited amounts of memory and low latency.

In this dissertation, we present graph partitioning techniques for scalable streaming graph and time series analysis. First, we present and evaluate the use of data partitioning to enable data parallelism in order to address the challenge of scale in large spatial time series forecasting. We propose a graph partitioning technique for large scale spatial time series forecasting of road traffic as a use-case. Our experimental results on traffic density prediction for real-world sensor dataset using Long Short-Term Memory Neural Networks show that the partitioning-based models take $12\times$ lower training time when run in parallel compared to the unpartitioned model of the entire road infrastructure. Furthermore, the partitioning-based models have $2\times$ lower prediction error (RMSE) compared to the entire road model. Second, we showcase the practical usefulness of streaming graph analytics for large spatial time series analysis with the real-world task of traffic jam detection and reduction. We propose to apply streaming graph analytics by performing useful analytics on traffic data stream at scale with high throughput and low latency. Third, we study, evaluate, and compare the existing state-of-the-art streaming graph partitioning algorithms. We propose a uniform analysis framework built using Apache Flink to evaluate and compare partitioning features and characteristics of streaming graph partitioning methods. Finally, we present *GCNSplit*, a novel ML-driven streaming graph partitioning solution, that uses a small and constant in-memory state (bounded state) to partition (possibly unbounded) graph streams. Our results demonstrate that *GCNSplit* provides high-throughput partitioning and can leverage data parallelism to sustain input rates of 100K edges/s. *GCNSplit* exhibits a partitioning quality, in terms of graph cuts and load balance, that matches that of the state-of-the-art HDRF (High Degree Replicated First) algorithm while storing three orders of magnitude smaller partitioning state.

Sammanfattning

De senaste åren har bevittnat en massiv ökning av mängden data som genereras av Internet of Things (IoT) och sociala medier. Att bearbeta enorma mängder av denna data innebär icke-triviala utmaningar när det gäller hårdvaru- och prestandakrav för dagens tillämpningar. De uppgifter vi har att göra med idag är i stor skala, med hög intensitet och finns i olika former. MapReduce var ett populärt och smart val av hantering av big data med hjälp av en distribuerad programmeringsmodell, vilket gjorde det möjligt att bearbeta stora datamängder med kluster av standarddatorer. MapReduce passade emellertid inte bra för att utföra komplexa uppgifter, såsom grafbehandling, iterativa program och maskininlärning. Moderna ramverk för databehandling, som ofta används för att bearbeta komplexa data och utföra komplexa analysuppgifter, övervinner bristerna i MapReduce. Några av dessa populära ramverk inkluderar Apache Spark för batch- och streambearbetning, Apache Flink för streambearbetning och Tensor Flow för maskininlärning.

I denna avhandling behandlar vi komplexa analyser av data modellerade som tidsserier, grafer och strömmar. Tidsserier används ofta för att representera tidsdata som genereras av IoT-sensorer. Att analysera och prognostisera tidsserier, dvs extrahera användbara egenskaper och statistik och förutsäga data, är användbart för många områden till exempel neurofysiologi, ekonomi, miljöstudier, transport etc. En annan användbar datarepresentation vi arbetar med är grafer. Grafer är komplexa datastrukturer som används för att representera relationsdata i form av hörn och kanter. Grafer finns i olika tillämpningsdomäner, såsom rekommendationssystem, vägtrafikanalys, webbanalys, och sociala medianalyser. På grund av den ökande storleken på grafdata är en enskild dator ofta inte tillräcklig för att bearbeta hela grafen. Därför måste beräkningen såväl som data distribueras. Grafpartitionering, processen att dela in grafer i subgrafer, är ett viktigt steg i distribuerad grafbehandling av storskaliga grafer eftersom det möjliggör parallell och distribuerad bearbetning.

Majoriteten av data har sitt ursprung i en kontinuerlig ström, såsom händelseserier från ett socialt medianätverk, tidsserier genererade från sensorer, finansiella transaktioner etc. Strömbehandlingsparadigmet hänvisar till bearbetning av dataströmmar vilka är kontinuerliga och möjligen obegränsade. Att kombinera både grafer och strömmar leder till en intressant och ganska utmanande domän för analys av strömmande grafer. Grafströmmar hänvisar till data som är modellerad som en ström av kanter eller hörn med närliggande-listor som representerar relationer mellan enheter för kontinuerligt utvecklande data som genereras av en eller flera datakällor. Analys för strömmande grafer är ett framväxande forskningsfält med stor potential på grund av dess möjligheter att bearbeta stora grafströmmar med begränsade mängder minne och låg latens.

I denna avhandling presenterar vi grafdelningstekniker för skalbar analys av strömmande grafer och tidsserier. Först presenterar och utvärderar vi användningen av datapartitionering för att möjliggöra dataparallellism för att hantera skalans utmaning i stora rumsliga tidsserieprognoser. Vi föreslår en grafuppdelningsteknik för storskalig rumslig tidsserieprognostisering av vägtrafik som användningsfall. Våra experimentella resultat om trafikdensitetsförutsägelse för verkliga sensordatamängder med hjälp av Long Short-Term Memory Neural Networks visar att de partitionsbaserade modellerna tar $12\times$, om de körs parallellt, mindre träningstid jämfört med den opartitionerade modellen av hela väginfrastrukturen. Dessutom har de partitioneringsbaserade modellerna $2\times$ färre förutsägelsesfel (RMSE) jämfört med hela vägmodellen. För det andra visar vi den praktiska användbarheten av strömmande grafanalys för storskalig rumslig tidsserieanalys med den verkliga uppgiften att upptäcka och minska trafikstockningar. Vi föreslår att man använder strömmande grafanalys genom att utföra användbar analys på trafikdataströmmar i stor skala med hög genomströmning och låg latens. För det tredje studerar, utvärderar och jämför vi befintliga toppmoderna partitioneringsalgoritmer för strömmande grafer. Vi föreslår ett enhetligt analysramverk byggt med Apache Flink för att utvärdera och jämföra partitioneringsfunktioner och egenskaper hos partitioneringsmetoder för strömmande grafer. Slutligen presenterar vi *GCNSplit*, en ny ML-driven partitioneringslösning för strömningsdiagram, som använder ett litet och konstant tillståndsminne i minnet (finit tillstånd) för att partitionera (eventuellt obegränsade) grafströmmar. Våra resultat visar att *GCNSplit* tillhandahåller partitionering med hög kapacitet och kan utnyttja dataparallellism för att upprätthålla ingångshastigheter på 100K $\mathtt{kanter/s}$.

GCNSplit uppvisar en partitioneringskvalitet, i termer av grafskärningar och belastningsbalans, som matchar den toppmoderna HDRF-algoritmen (High Degree Replicated First) medan den lagrar tre storleksordningar mindre partitioneringstillstånd.

.

Résumé

Les dernières années ont été témoin d'une augmentation massive de la quantité de données générées par l'Internet des objets (IoT) et les médias sociaux. Le traitement d'énormes quantités de ces données pose des défis non triviaux en termes d'exigences matérielles et des performances d'applications modernes. Les données que nous traitons aujourd'hui sont d'échelle massive, de haute intensité et se présentent sous diverses formes. MapReduce était un choix populaire et astucieux pour gérer les données volumineuses à l'aide d'un modèle de programmation répartie, ce qui rendait possible le traitement d'énormes quantités de données à l'aide de clusters de machines de base. Cependant, MapReduce n'était pas adapté à l'exécution de tâches complexes, telles que le traitement de graphes, les programmes itératifs et l'apprentissage automatique. Les plates-formes modernes de traitement de données, utilisées communément pour traiter des données complexes et effectuer des tâches d'analyse complexes, surmontent les lacunes de MapReduce. Certaines de ces plates-formes populaires incluent Apache Spark pour le traitement par lots et par flux, Apache Flink pour le traitement par flux et Tensor Flow pour l'apprentissage automatique.

Dans cette thèse, nous traitons des techniques d'analyse complexes sur des données modélisées sous forme de séries temporelles, de graphes et de flux. Les séries temporelles sont couramment utilisées pour représenter les données temporelles générées par capteurs IoT. L'analyse et la prévision des séries temporelles, c'est-à-dire l'extraction des caractéristiques et des statistiques utiles des données et la prédiction des données, sont utiles dans de nombreux domaines, notamment la neurophysiologie, l'économie, les études environnementales, les transports, etc. Une autre représentation de données utile avec laquelle nous travaillons est le graphe. Les graphes sont des structures complexes utilisées pour représenter des données relationnelles sous forme de noeuds et d'arêtes. Les graphes sont présents dans divers domaines d'application, tels que les systèmes de recommandation, l'analyse du trafic routier, l'analyse Web, l'analyse des médias sociaux. En raison de la taille croissante des données en forme de graphes, souvent un seul ordinateur n'est pas suffisant pour traiter tout le graphe. Par conséquent, le calcul, ainsi que les données, doivent être répartis. Le partitionnement de graphes, c'est-à-dire la division de graphes en sous-graphes, est une étape essentielle pour le traitement de graphes à grande échelle car il permet un traitement parallèle et réparti.

La majorité des données générées à partir de l'IoT et des médias sociaux existent comme un flux continu, telles que les séries d'événements d'un réseau de médias sociaux, les séries temporelles générées à partir de capteurs, les transactions financières, etc. Le paradigme de traitement de flux fait référence aux flux de données continus et potentiellement non bornés. Combiner ensemble des graphes et des flux conduit à un domaine intéressant et plutôt difficile d'analyse. Les flux de graphes font référence à des données modélisées comme un flux d'arêtes ou de noeuds avec des listes d'adjacence représentant des relations entre des entités en constante évolution générées par une ou plusieurs sources de données. L'analyse des graphes en continu est un domaine de recherche émergent avec un grand potentiel en raison de ses capacités à traiter de grands flux de graphes avec une mémoire limitée et une faible latence.

Dans cette thèse, nous présentons des techniques de partitionnement de graphes pour l'analyse à grande échelle des graphes en continu et des séries temporelles. Tout d'abord, nous présentons et évaluons l'utilisation du partitionnement pour permettre le parallélisme des données afin de relever le défi de l'échelle dans la prévision de grandes séries temporelles spatiales. Nous proposons une technique de partitionnement de graphes pour la prévision de séries temporelles spatiales à grande échelle du trafic routier comme cas d'utilisation. Nos résultats expérimentaux sur la prédiction de la densité du trafic pour l'ensemble de données de capteurs du monde réel à l'aide de réseaux neuronaux récurrent à mémoire court et long terme (Long Short-Term Memory, LSTM) montrent que les modèles basés sur le partitionnement prennent 12 fois moins de temps d'apprentissage lorsqu'ils sont exécutés en parallèle par rapport au modèle non partitionné de l'ensemble de l'infrastructure routière. De plus, les modèles basés sur le partitionnement ont une erreur de prédiction 2 fois inférieur (erreur quadratique moyenne, RMSE) par rapport au modèle routier en entier. Deuxièmement, nous présentons l'utilité pratique de l'analyse des graphes en continu pour l'analyse de grandes séries temporelles spatiales avec la tâche réelle de détection et de réduction des embouteillages. Nous proposons d'appliquer l'analyse

des graphes en continu en effectuant des analyses utiles sur les flux de données de trafic à grande échelle avec un débit élevé et une faible latence. Troisièmement, nous étudions, évaluons et comparons l'état de l'art des algorithmes de partitionnement de graphes en continu. Nous proposons une plate-forme d'analyse uniforme construite à l'aide d'Apache Flink pour évaluer et comparer les fonctionnalités de partitionnement et les caractéristiques des méthodes de partitionnement de graphes en continu. Enfin, nous présentons *GCNSplit*, une nouvelle solution de partitionnement de graphes en continu pilotée par l'apprentissage automatique, qui utilise un état en mémoire petit et constant (un état borné) pour partitionner des flux de graphes (peut-être non bornés). Nos résultats démontrent que *GCNSplit* fournit un partitionnement à haut débit et peut tirer parti du parallélisme des données pour maintenir des taux soutenus de 100000 arêtes par seconde. *GCNSplit* présente une qualité de partitionnement, en termes de coupes de graphes et d'équilibrage de charge, qui correspond à celle de l'algorithme avancé HDRF (High Degree Replicated First) tout en stockant un état de partitionnement trois ordres de grandeur plus petit.

Acknowledgements

"If a person teaches me one single word, he has made me his servant for a lifetime."
— Ali Ibn Abi Talib (as)

My PhD journey is almost complete now and this is the section I have been waiting to type since the day I started my PhD. The reason for that is my mentors who supported me from day 1 of this journey, without their support I could not have done it. I have learned a lot and developed myself as a researcher and a human being all because of the support of my family, friends and teachers. Here comes the list of people who were part of this journey.

My primary supervisor Vladimir Vlassov a.k.a Vlad, thank you for your utmost support, guidance and care throughout my PhD. I am very grateful for your advice and help during my research work. You have given me knowledge and life lessons that will help me throughout my life. I will miss our conference travels and zoom meetings.

My supervisor Peter Van Roy, thank you for the valuable guidance, insights and support. You provided me with very helpful recommendations and advice in shaping my research direction and presenting my research work. Thank you for helping me with the french translation too.

My co-supervisor Vasiliki Kalavri a.k.a Vasia, you have been an inspiration for me. I am glad that I have worked under your supervision and gained immense knowledge. You have always encouraged me and motivated me to move forward and you have also stood by me during the highs and lows.

My co-supervisor Paris Carbone, you have provided me with invaluable expertise during my research. I am happy to have you as a mentor and greatly thankful for your support, encouragement and guidance during my PhD. Also, thank you for the great parenting tips you shared with me during my tough times.

My (ex) co-supervisor Sarunas Girdzijauskas, thank you for being part of my journey by providing useful guidance and help at the start of my research.

My internal thesis reviewer Henrik Boström, thank you for your timely, in-depth and detailed comments. Your review has helped in improving my thesis.

My collaborator Ahmad Al-Shishtawy, thank you for your support and guidance. It was a great collaboration working on real traffic datasets. I gained proficient knowledge working with you during my internship.

Our head of department Thomas Sjöland, thank you for always being supportive to me and special thanks for helping me with the Swedish abstract. I will miss our coffee break discussions. It was a delight to have an in-depth and engaging discussion with you.

My collaborators and co-authors Amir Hossein Payberah, Muhammad Arsalan, Paolo Sottovia, Mohamad Al Hajj Hassan, Daniele Foroni and Stefano Bortoli, it was a pleasant and great learning experience working with you all.

Christian Schulte and Seif Haridi for their valuable support and advice during my PhD.

Coolest Professor Johan Montelius, you have always been a breath of fresh air in the department. Thank you for visiting my office and lighting up my mood with your cheerful visits.

My lovely friend, colleague and office mate Sana Imtiaz, special gratitude and thanks to you for being there with me, listening to me and supporting me throughout my journey. Thank you for tolerating my music and singing in the office. I would also like to thank my dear friends Sahar Imtiaz, Amira Soliman, Shatha Jaradat and Leila Bahri for the wonderful time we had together.

My EMJD-DC friends and colleagues, especially Leila Sharifi and Khulan Batbayar for inspiring me and encouraging me during my work. My department colleagues Edward, David Gureya, Anis Nasir, Igor, Hooman and Kamal for their fruitful discussions we had during lunch and coffee times. My group members Sina Sheikholeslami and Tianze Wang for providing access to their machines during my work. My talented students Mikał Zwolak, Jon Reginbald Ivarsson, Sonia-Florina Horchidan and Thorsteinn Thorri Sigurdsson for their collaboration and hard work. Thank you for contributing to my work.

Susy Mathew, Madeleine Printzsköld, Vanessa Maons and Laurence Bertrand for handling all the administrative work throughout my PhD.

All anonymous reviewers, especially reviewer no.2, for accepting and rejecting my work by giving constructive and useful feedback.

My dear parents, thank you mama and papa for your constant love, prayers and support. I could not have gone this far without you. Thank you for calling me every day to check if I am doing well and sending me your love. My siblings, Onnera and Hassan, thank you for your constant love and the entertainment you both bring to my life. Your faith in me has given me immense strength.

My love Mudasser, you have lived this journey together with me. You have taken care of me throughout my PhD. I give all the credits of this work to you because it could not have been complete without your support and patience. Thank you for tolerating my late work hours during deadlines, my conference visits and research trips. Thank you for helping me raise our lovely baby Husayn. You are a great friend, partner, husband and dad.

This work was supported by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030 and FoFu.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER

1

Introduction

"You cannot run faster than you run"
— Vladimir Vlassov a.k.a Vlad when I started writing my thesis

1.1 Research Motivation and Context

Analytics on complex and multidimensional data representations, such as graph streams and spatial time series, is becoming increasingly important today because it allows extracting knowledge in various useful domains where the traditional flat file data, i.e., text data, fails to do so. Some of these domains include traffic analysis [1], social media analysis [2], web analysis [2], machine learning [3], recommendation systems [4], environmental studies [5] and economics [6]. However, it is difficult to make these analytics scalable on such complex data representations because we have to take into account the multiple data dimensions, the size of data, and the computation that itself is complicated. In this thesis, we are looking into two types of complex and multidimensional data representations, graph streams and spatial time series. Moreover, we apply partitioning and Machine Learning (ML) methods to improve the performance and scalability of analytics on these complex representations based big datasets. In the rest of the introduction, we introduce necessary definitions and background.

Big Data. Every day, a large amount of data is being produced through social media, the World Wide Web and Internet of Things (IoT) devices. The term *Big Data* is popularly used to refer to this ever-increasing data in terms of its volume, velocity and variety [7]. Even though current developments in the data processing domain have made the handling of Big Data convenient for both industry and academia, they are still not enough to handle massive data efficiently. In terms of volume, the available memory and disk space of a single machine fall short to fit the concerned

data. Considering the current speed (Velocity) at which different types (Variety) of data is being generated, in the form of raw data, semi-structured and unstructured data, new frameworks are required to not only combine the data coming from various sources in different forms but also to process the high-intensity data with low-latency.

Graph Streams. Modern day data processing systems are evolving to handle complex tasks such as stream processing, machine learning and graph analytics [8, 9, 10]. Graphs are complex data structures used to model and represent relational data present in various useful application domains such as web analysis, recommendation systems, social networks analysis, road infrastructure data processing and community detection. Since, the majority of data originates as a continuous stream, such as series of events from social media, time series generated from sensors, financial transactions, thus, it is reasonable to represent real-life continuous graphs as streams. Graph streams are (possibly unbounded) sequences of timestamped events that represent relationships between entities: user interactions in social networks, online financial transactions, driver and user locations in ride-sharing services. Graph streams are continuously ingested from external, often distributed, sources and are modeled either as streams of edges or as vertex streams with associated adjacency lists. Streaming graph analytics deals with the processing of data graphs in motion, i.e., streaming graphs in the form of a stream of vertexes or edges. Streaming graph processing combines both graph analysis and stream processing. It is an emerging application area that aims to extract knowledge from evolving networks modeled as graphs in a timely and efficient manner [11, 12].

Spatial Time Series. Another common type of streaming data representation is time series data generated largely by IoT sensors. Time series are used to represent temporal data. Multiple time series that correspond to different spatial locations are referred to as spatial time series [13]. Analysing and forecasting spatial time series, i.e. extracting useful characteristics and statistics of data and predicting data, is useful for many fields that include, neuro-physiology [14], economics [6], environmental studies [5], transportation [15] etc. Spatial time series often contain both spatial dependencies (inter-dependencies) and temporal dependencies (intra-dependencies), that need to be taken into account when analysing time series data [13, 16, 17]. These spatio-temporal dependencies are challenging to model due to their dynamic nature and varying ranges, where the range refers to the closeness in space and time.

Data Partitioning. Processing massive streaming graphs and time series data on a single machine is not efficient in terms of memory and compute requirements. Data partitioning is an efficient way of dividing data and computations across distributed compute nodes. Both graph streams and time series data require different partitioning strategies. In the context of graphs, graph partitioning is

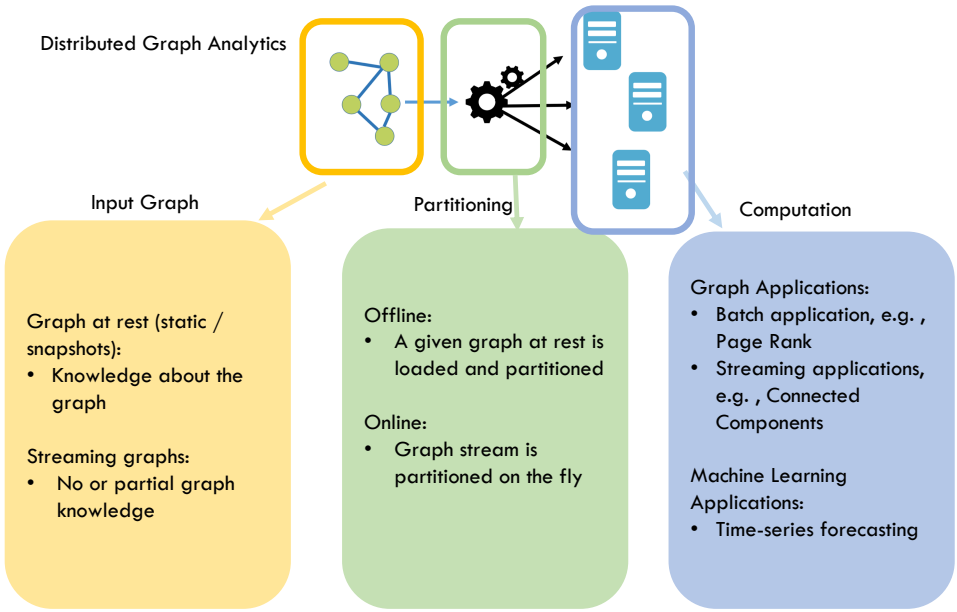


Figure 1.1 – Distributed graph analytics; using graph partitioning to do parallelization/distribution of work for scalabilty and performance improvement.

the process of dividing a graph into a predefined number of subgraphs. In such settings, each cluster node operates on one partition in parallel with other nodes and communicates with other nodes through message-passing [18, 19, 20]. Hence, partitioning directly affects the communication across nodes and computations of compute nodes thus making it crucial for graph application performance. Online graph partitioning methods process graph streams and assign edges or vertices to the partitions on-the-fly [21, 22, 23, 2, 24]. Whereas, for partitioning spatial time series, the spatio-temporal dependencies need to be taken into account to partition the data efficiently by placing correlated time series in the same partition. Naive partitioning affects the performance of time series analysis tasks by placing uncorrelated time series in the same partition causing low prediction accuracy and slow training of Deep Learning (DL) based forecasting methods trained on partitioned data. Therefore, we identify an emerging need to tackle complexity in data partitioning.

Partitioning Problem. Figure 1.1 gives a summary of our research topic in the domain of distributed graph analytics. We embark on the idea of using graph partitioning to do parallelization and distribution of work, with minimal dependencies for graph and machine learning applications. Graph applications consist of iterative batch applications, such as Page Rank and aggregation based streaming

applications. Machine learning applications that we work with consist of neural networks based forecasting models which are trained on the subset of data generated after partitioning. Both graph applications and machine learning applications are different, but they can benefit from good data partitioning strategies. In this thesis, we investigate finding efficient techniques for partitioning the data, which in our case is in time series or graph stream form, for improving the performance and scalability of machine learning and graph processing applications. We identify an emerging need to tackle the complexity of data parallelism for time series and graph data. Time series contain dependencies that need to be preserved during partitioning for performing efficient analytics. Similarly, for graphs, the data locality should be preserved during partitioning to reduce the communication cost during distributed processing.

1.2 Thesis Statement

Partitioning and machine learning methods can help to improve the scalability and analytics for both

- Streaming Graph Processing by reducing the I/O communication cost;
- Machine learning-based time series prediction by parallelising the training process and improving the accuracy

1.3 Dissertation Outline

The thesis is structured as follows. The remainder of Chapter 1 contains details on the research challenges addressed in this thesis, research contributions of this thesis, research methodology adapted during this work and the publications list. Later, Chapter 2 gives background on graph partitioning, streaming graph partitioning and graph embeddings. In Chapter 3 we propose a graph partitioning approach to partition spatial time series for scalable time series forecasting. Chapter 4 showcases the practical usefulness of streaming graph analytics for large spatial time series analysis with the real-world task of traffic jam detection and reduction. In Chapter 5, we explore the domain of streaming graph partitioning by studying, comparing, and evaluating various streaming graph partitioning algorithms in detail. In Chapter 6 we propose a novel streaming graph partitioning algorithm that uses elements of machine learning, in particular, graph convolutional networks, to learn the structure of graphs for making partitioning decisions. Finally, the conclusion and future work is presented in Chapter 7.

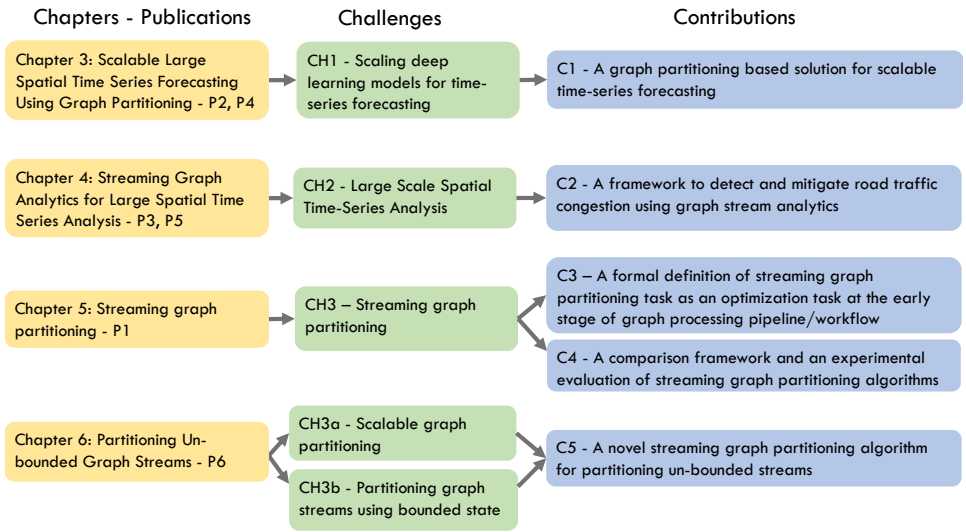


Figure 1.2 – Road Map highlighting chapters, publications, challenges and respective contributions

Figure 1.2 presents the road map for this dissertation mapping the chapters with their related publications, research challenges and contributions. The research challenges and contributions are mentioned in detail in Section 1.4 and 1.5. The list of publications is mentioned in Section 1.7.

1.4 Research Challenges

Ever-increasing quantities of data generated by internet users as well as by the IoT ecosystem makes processing computationally and memory intensive for data-intensive applications. IoT data is usually in the form of time series generated from various spatial locations, also it is of massive scale. In this thesis, we deal with processing large scale spatial time series data generated by huge complex systems. We use a graph to model these systems. We look into applying graph analytics and graph partitioning for efficient processing of large spatial time series data since graph partitioning is useful to parallelize and scale the computations over time series data. We also work with graphs arriving as streams, referred to as streaming graphs. Processing streaming graphs add up to the complexity of our research due to the inherent limitations of the streaming model such as lack of shared state, iterations and cross-task communication. We focus on developing efficient partitioning techniques for streaming graphs to deal with the streaming graph partitioning challenges mentioned later in this section.

In the course of our research with the aforementioned focus, we address the following research challenges and the research questions associated with these challenges. The first challenge is related to spatial time series forecasting using partitioning (CH1), the second challenge is related to spatial time series analysis using streaming graph processing algorithms (CH2), and the remaining challenges (CH3, CH3a and CH3b) are associated with streaming graph partitioning.

1.4.1 [CH1] Scaling Deep Learning Models for Time Series Forecasting

Performing a learning task on large scale spatial time series is a non-trivial task because it is computationally expensive to do on a single machine. Especially if the task involves deep learning, it becomes computationally and memory intensive to train the model. Data parallelism helps to make training and inference parallel, and can in turn improve training time. We investigate this matter further by studying the use of graph partitioning for improving the performance not only in terms of training time but also in terms of accuracy, complexity and scalability. We want to address the challenge of scale and improve the performance (including training time, accuracy, scalability and complexity) for DL models using graph partitioning. Partitioning spatial time series is challenging due to the fact that we need to model the spatial and temporal dependencies in the data [13, 16, 17] and perform careful partitioning to preserve these dependencies for accurate time series forecasting. Careless partitioning causes degradation in the model accuracy. For example, too small partitions might not provide enough information to train the model; Placing unrelated data in the same partition might negatively affect the accuracy or complexity of the model. The negative effect can happen because uncorrelated data will make it hard for the learning model to learn trends in the data and thus affect the overall accuracy. Also, adding uncorrelated data will increase the dataset size and in the case of large scale data, it will increase the parameters of the learning model, thus making it complex and hard to train. We want to design a partitioning algorithm that results in data that is correlated and which does not increase the training complexity of the learning model. We describe in more detail how we address this challenge in Chapter 3 and Papers P2 [25] and P4 [1].

Research Question. How does graph partitioning help to scale and improve the performance of deep learning models? P2 [25] and P4 [1].

1.4.2 [CH2] Large Scale Spatial Time Series Analysis

Large scale spatial time series analysis includes processing multiple correlated time series. These time series often contain spatial dependencies (inter-dependencies) and temporal dependencies (intra-dependencies). It is challenging to model time

series data for capturing both spatial and temporal dependencies at scale because these dependencies are dynamic and are both short and long-range [13, 16, 17]. The range here is the closeness in terms of time and space. For example, spatial dependency is not always related to the physical closeness of the data source generating the time series. Also, modern systems have real-time requirements that require low latency based solutions. In order to address the challenge of scale and performance, we propose to model spatial time series as graph streams for capturing time and space dependencies in large scale time series. We use streaming graph analytics to perform useful analysis of multiple time series at scale because stream processing models give low latency and high throughput, which is suitable for real-time analysis tasks. The other challenge is to find a suitable graph stream processing algorithm for the required application task. We consider a practical use-case of road traffic analysis. We model traffic data streams as graph streams and use the streaming connected components algorithm for detecting and reducing traffic jams in a real-life application of road traffic analytics. More details on how we address this challenge are explained Chapter 4 and Papers P3 [26] and P5 [27].

Research Question. How does streaming graph processing on time series give better performance compared to the state-of-the-art analytics methods?

1.4.3 [CH3] Streaming Graph Partitioning

Balanced graph partitioning is an NP-hard optimization problem with two main objectives: 1) balancing load across partitions and 2) reducing communication cost (fewer graph cuts) between the partitions. The problem becomes non-trivial when the graph arrives as a stream and partitioning has to be done on-the-fly without having prior knowledge of the graph. This online style of partitioning is termed "streaming graph partitioning". Little work has been done on surveying streaming graph partitioning algorithms. Also, to our knowledge, no work has been done considering unbounded streams and single-pass graph stream aggregations, an emerging application domain with increasing system support [28, 29, 30, 31, 10]. Additionally, few of the state-of-the-art algorithms are open source. We aim for providing a comparison framework for state-of-the-art streaming graph partitioning algorithms and survey their shortcomings.

Research Questions. In our survey, we seek to answer the following research questions. 1) What are the benefits, if any, of using more complex, data-centric partitioning methods compared to a generic hash-based strategy? 2) What is the partitioning overhead for an application using each partitioning algorithm? 3) How does the partitioning quality affect the application performance?

Based on the experimental results of our survey in Chapter 5 and Paper P1 [24], we conclude that existing state-of-the-art streaming graph partitioning algorithms, such as HDRF (High Degree Replicated First) [21], are stateful. They accumulate some in-memory state to make smart partitioning decisions based on the knowledge extracted so far from the incoming stream. The problem with an increasing state is that the state needs frequent updates and is growing with the input graph size, making them impractical to process high-intensity and unbounded streams. In our next work Chapter 6, we do research for finding an online graph partitioning method capable of handling unbounded streams by addressing the challenges CH3a and CH3b described below.

1.4.4 [CH3a] Scalable Graph Partitioning

Almost all state-of-the-art streaming graph partitioning methods base their partitioning decision using an in-memory state that contains information, such as current vertex assignment, partition capacities, or vertex degree distributions. This state is frequently updated with each arriving stream element, i.e., an edge or a vertex, and on each partitioning decision and it is also shared among parallel partitioning processes for a global view of state across parallel instances. This requirement of a global view increases the communication cost between parallel partitioning instances and it makes such stateful partitioning methods impractical to be used for a no-shared state architecture that is used by modern stream processing engines. We investigate developing a class of partitioning algorithms that do not require a global shared state, are easy to scale and give good partitioning quality.We give more details on how we address this challenge in Chapter 6 and Paper P6 [32].

Research Question. How to partition graph streams efficiently without a global shared state?

1.4.5 [CH3b] Partitioning Graph Streams Using Bounded State

State-of-the-art online graph partitioning methods that make high-quality (i.e. balanced and minimal graph cuts) partitioning, such as HDRF, keep accumulating in-memory growing state proportional to the number of vertices of the graphs $O(|V|)$. The state size keeps on increasing during the processing of unbounded streams. These growing state-based methods are impractical for applications that continuously process unbounded graph streams and for applications that process large graphs in rest. Also, the state cannot be re-used to partition unseen graphs. Every time a graph edge or vertex arrives, the state has to be updated. We intend to develop streaming graph partitioning methods that give bounded state guarantees with state size independent of the length of a streaming graph or the size of the graph in rest. We describe in more detail how we address this challenge in Chapter 6 and Papers P6 [32].

Research Question. How to partition unbounded graph streams with a bounded state efficiently?

1.5 Contributions

The main contributions of this dissertation addressing the aforementioned challenges CH1-CH3 are as follows.

C1 **A graph partitioning based solution for scalable time series forecasting.** We propose to represent a complex system generating large spatial time series data in the form of a directed weighted graph for capturing spatio-temporal dependencies among the time series data generated by the components of the system. We develop graph partitioning techniques to enable scalable and accurate time series forecasting of large scale spatial time series data addressing *CH1-Scaling Deep Learning Models for Time Series Forecasting* (Papers P2 [25] and P4 [1]). Our application domain is the analysis of road traffic data collected by road infrastructure sensors and modelling of traffic flow behaviour for the task of traffic prediction. Our experimental results on traffic density prediction show that the partitioning-based models take $12\times$, if run in parallel, lower training time, compared to the unpartitioned model of the entire road infrastructure. Furthermore, the partitioning-based models have $2\times$ lower prediction error (RMSE) compared to the entire road model. Other works done in this domain [33, 34, 35] on road traffic prediction do not fully address the issue of scale and model complexity at large scale that we tackle in our work using graph partitioning.

C2 **A framework to detect and mitigate road traffic congestion using streaming graph analytics.** We present a practical application of streaming graph analytics for modeling spatial and temporal dependencies to control road traffic jams at scale. In this contribution we tackle *CH2-Large Scale Spatial Time Series Analysis* (Papers P3 [26] and P5 [27]). We propose to offer an end-to-end traffic control framework based on Apache Flink [36], which is a modern distributed stream processing engine. Our system comprises of 1) an online traffic jam detection mechanism for detecting jams on streaming data collected from traffic sensors, and 2) a congestion reduction mechanism based on streaming graph analytics for reducing the effect of congestion in the congested area. Existing works on congestion detection [37, 38, 39] mostly use historic information and require pre-processing, thus making them unsuitable for real-time processing. In our proposed congestion reduction approach, we identify correlated traffic jams and essential parts in the road network on which new traffic light policies are deployed for congestion control. We develop dynamic traffic light policies based on our congestion reduction

mechanism that helps in mitigating the impact of congestion by reducing the travel time of cars during traffic jams. Our experimental results indicate our dynamic traffic light policies result in 27% less travel time at the best and 8% less travel time on average compared to the travel time with default traffic light policies. Our scalability results show that our system is able to handle high-intensity streaming data collected from 900 sensors every second with a throughput of 57K records/sec at best.

C3 **A formal definition of streaming graph partitioning task as an optimization task at the early stage of graph processing pipeline/workflow.** We define the domain of online graph partitioning and its role in graph processing workflows, decoupling the partitioning step from the application logic computation, whether staged or pipelined. We are the first ones to define online graph partitioning as an optimisation task that aims to minimize three objective functions: 1) minimize replication factor, 2) minimize load imbalance and 3) minimize the partitioning state dealing with *CH3-Streaming Graph Partitioning* (Paper P1 [24]).

C4 **A comparison framework and an experimental evaluation of streaming graph partitioning algorithms.** We propose a uniform analysis framework to evaluate and compare partitioning features and characteristics of streaming partitioning methods addressing *CH3-Streaming Graph Partitioning* (Paper P1 [24]). We classify algorithms with regards to their data model, strategy, constraints, complexity, state requirements, and objectives. To provide an unbiased performance comparison, we implement all studied methods on top of a common evaluation framework based on Apache Flink, a distributed stream processing engine. We use bulk synchronous and single-pass graph streaming algorithms to evaluate distributed graph application performance in terms of partitioning cost amortization. Compared to existing works on graph partitioning [40, 41, 42, 43, 44, 45], our work is, to our knowledge, the first dedicated study to online graph partitioning methods that includes stream-specific properties (e.g., ingestion order) as well as considering single-pass graph stream aggregations, an emerging application domain with increasing system support [28, 29, 30, 31, 10]. Our experimental results showcase that model-dependent online partitioning techniques such as low-cut algorithms offer better performance for communication-intensive applications such as bulk synchronous iterative algorithms, albeit with higher partitioning costs (cost here refers to latency). Otherwise, model-agnostic techniques trade-off data locality for lower partitioning costs and balanced workloads which is beneficial when executing data-parallel single-pass graph algorithms.

C5 **A novel streaming graph partitioning algorithm for partitioning unbounded streams.** The state-of-the-art streaming graph partitioning algorithms are unsuitable for unbounded streams because they keep a global state which becomes a bottleneck when distributing the partitioning logic across parallel instances. We introduce a novel approach to streaming graph partitioning that is practical for use with unbounded streams without sacrificing partitioning quality or load balance tackling *CH3a-Scalable Graph Partitioning* and *CH3b-Partitioning Graph Streams Using Bounded State* (Paper P6 [32]). We propose the novel application of inductive graph convolutional networks (GCNs) to the streaming graph partitioning problem for the first time. We provide a solution to online graph partitioning with a bound-size state that does not compromise partitioning quality and load balancing performance. We implement GCNSplit, an extensible framework that unifies the training and serving partitioning pipelines.Our approach to streaming graph partitioning generalizes to unseen graphs. GCNSplit 's models can be used to partition not just unseen edges of the input graph but also entirely unseen graphs with similar structure and feature set. Our results demonstrate that *GCNSplit* provides high-throughput partitioning and can leverage data parallelism to sustain input rates of 100K edges/s. At the same time, *GCNSplit* generates high-quality and well-balanced partitions that matches that of the state-of-the-art HRDF algorithm, while storing three orders of magnitude smaller partitioning state. Furthermore, *GCNSplit* scales linearly to the number of parallel processes outperforming HDRF.

1.6 Research Methodology

In this section, we give an overview of the methods used throughout our research work. We first start by presenting the general approach of our work followed by the implementation choices and experimental evaluation methods used by us. In the end, we mention certain challenges we faced during our research.

1.6.1 General Approach

We chose to work with processing complex data representations that include graphs and time-series data because they allow extracting knowledge that is not possible to do with simple text data. Our main goal is to provide scalable analytics for graphs and time series and we mostly consider data arriving as a stream because the majority of data originates as a continuous stream.

We use a quantitative and empirical approach throughout our research work. First, we identify the research problem by performing a detailed literature review of the existing state-of-the-art in our research domain, i.e., time series and streaming graph processing. We work with quantitative data throughout our work. All data

is publicly available except for the traffic data, which is owned by the company that collects the data. Next, we identify the problem after careful observation and review of the existing related work. To verify our problem empirically, we implement the existing solutions and perform experiments that help to formulate the research challenges and the research questions associated with the challenges (Section 1.4). For example, for time series prediction, we implement various prediction models using Tensor Flow and also for streaming graph partitioning we implement existing state-of-the-art partitioning methods on Apache Flink. Then, we performed experiments using the standard performance metrics that are domain-specific for time series prediction and analysis of traffic data. Similarly, for streaming graphs, we computed the quality of partitioning methods using the quality metrics mentioned in various graph partitioning surveys. After drafting the challenges and research questions based on our experimental results, we make careful design decisions to handle the challenges. For example, for time series prediction in Chapter 3 we observed that the current solutions take too long to train, thus we decided to reduce the complexity of the model and make it scalable by using graph partitioning. Similarly, for streaming graph partitioning, after performing an experimental survey in Chapter 5, we identified the bottlenecks during online partitioning of unbounded streams. We designed algorithms and techniques to solve the aforementioned challenges and experimentally compared our solutions with the existing work. Our solution for scaling road traffic prediction using graph partitioning in Chapter 3 show promising results in terms of scalability and performance. Similarly, for unbounded graph stream partitioning, in Chapter 6, our experimental results show that the algorithm is scalable and provides good partitioning quality overcoming the bottlenecks that were present in the state-of-the-art methods. Our results are highlighted as contributions (Section 1.5) addressing the research challenges and research questions (Section 1.4).

1.6.2 Implementations

We implemented our work using popular open-source stable and modern libraries and frameworks available. We used Apache Spark [9] to run, and TensorFlow to implement and train our DL based models in Chapter 3. We used Apache Flink[30] to implement streaming graph algorithms for data analytics and streaming graph partitioning algorithms in Chapter 4 and 5. Some of the algorithms of Chapter 4 are also implemented using Apache Spark. We used Python and Pytorch to implement our GCN-based streaming graph partitioning algorithms in Chapter 6. We provide open-source implementations of streaming graph partitioning algorithms that are part of Chapter 5and 6 and some streaming graph analytics algorithms that are part of Chapter 4. The provided software is mentioned under section 1.7.2 and is free for use. The remaining implementations were part of the industrial internship work and thus cannot be open-sourced.

1.6.3 Experimental Evaluation

We used the latest stable versions of the aforementioned open-source platforms. Most of our work with graph datasets in Chapter 5 and 6 is done using open-source datasets, the majority of which were taken from SNAP [46] data. Other sources of graph datasets are mentioned in the papers. Work in Chapter 3 and 4 were done using a traffic dataset taken from the Swedish Transport Administration and it is kept private. Part of the work in Chapter 4 was done during a research internship at Huawei and the dataset provided was anonymised for privacy preservation and kept for private use only by the organisation. The machines used in our work consist of both on-premises clusters and virtualised environments. In both cases, the setup was made making necessary software installations. The experiments were performed carefully avoiding interference from other parallel running programs. Some of the experiments in Chapter 4 are also done using simulation tools because the scenarios in the work with traffic light control systems were impractical to test for experiments on a large scale at the city level.

1.6.4 Challenges

Our first major challenge working with big graphs was the in-availability of on-premises machines that can process huge graph datasets. We had to buy expensive virtual machines for that purpose. Besides this, for our work in Chapter 5 and 6, some of the state-of-the-art streaming graph partitioning methods, i.e., Fennel, DBH, Grid (details in Chapter 5), were not open source. We had to implement them from the scratch. In some cases, we emailed the authors but did not get any response. We decided to carefully study the related work and implement the necessary ones to the best of our knowledge. After successful implementations, our experimental results matched the related work-based implemented algorithms results. We decided to open-source these algorithms for the researchers. During the experimental environment setup and implementation, we encountered several challenges, one of the reasons being version updates for a platform like Apache Flink and Tensorflow. However, the research community for these respective platforms is very active and they provided us good help to resolve the issues.

1.7 Publications and Software

1.7.1 Papers

The results presented in this thesis are published in journal papers, conference and workshop papers as the following.

P1 **Streaming Graph Partitioning: An Experimental Study**, Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 44th International Conference on Very Large Data Bases 2018 (VLDB 2018), Rio De Janeiro, Brazil, August 27-31, 2018. [24]

Contribution The author of this dissertation designed and developed the streaming graph partitioning algorithms, performed the experiments, and participated in writing the paper.

P2 **Short-Term Traffic Prediction Using Long Short-Term Memory Neural Networks**, Zainab Abbas, Ahmad Al-Shishtawy, Sarunas Girdzijauskas, and Vladimir Vlassov. IEEE International Congress on Big Data (BigData Congress), San Francisco, CA, USA, July 2-7, 2018. [25]

Contribution The author of this dissertation brainstormed, designed and developed traffic prediction algorithms, performed the experiments, and participated in writing the paper.

P3 **Evaluation of the Use of Streaming Graph Processing Algorithms for Road Congestion Detection**, Zainab Abbas, Thorsteinn Thorri Sigurdsson, Ahmad Al-Shishtawy, and Vladimir Vlassov. 16th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA 2018), Melbourne, Australia, December 11-13, 2018. [26]

Contribution The author of this dissertation brainstormed, helped in designing and developing traffic congestion detection algorithms and participated in writing the paper.

P4 **Scaling Deep Learning Models for Large Spatial Time Series Forecasting**, Zainab Abbas, Jon Reginbald Ivarsson, Ahmad Al-Shishtawy, Vladimir Vlassov. IEEE Big Data conference (Big Data), December 9-12, 2019, Los Angeles, CA, USA. [1]

Contribution The author of this thesis brainstormed, helped designing and developing the traffic prediction models and participated in writing the paper.

P5 **Real-time Traffic Jam Detection and Congestion Reduction Using Streaming Graph Analytics**, Zainab Abbas, Paolo Sottovia, Mohamad Al Hajj Hassan, Daniele Foroni, Stefano Bortoli. IEEE Big Data conference (Big Data), December 10-13, 2020, USA [27]

Contribution The author of this dissertation brainstormed, designed and developed traffic congestion detection algorithms and participated in writing the paper.

P6 **GCN-based Partitioning of Unbounded Graph Streams with Bounded State**, Michał Zwolak, Zainab Abbas, Sonia Horchidan, Paris Carbone, Vasiliki Kalavri, (under submission). [32]

Contribution The author of this dissertation brainstormed, helped in designing and developing the partitioning algorithms, performed the scalability experiments, and participated in writing the paper.

Figure 1.2 maps the publications, research challenges and contributions.

1.7.2 Software

We provide the following open-source software which was developed as a part of this research work.

- Road traffic congestion detection: A framework[1] to perform road traffic congestion detection on traffic data streams using streaming graph analytics. The software takes time-stamped sensor readings in the form of flow and speed values of vehicles as input along with the road network topology information. It feeds these time-stamped measures as a stream of traffic data to the congestion detection module, which uses the streaming graph analytics algorithms mentioned in P3 for congestion detection and generates an output of sensors that are part of the congested region. The algorithms are build using Apache Spark version 2.5. A sample of input data is also available for reproducibility purposes.

- Streaming graph partitioning: A framework[2] to evaluate and compare partitioning features and characteristics of streaming partitioning methods. The methods partitioning methods are implemented using Apache Flink version 1.2.0. The partitioning methods implemented consist of several vertex partitioning methods, i.e., Fennel, Linear Distribute Greedy (LDG) and Hash. Also edge partitioning methods, i.e., HDRF, DBH, Greedy, Grid etc. The details for these algorithms are explained in Chapter 5. The input for vertex partitioning algorithms is in the form of a list of vertices with their neighbours and for edge partitioning algorithms the input consists of an edge list. The output generated from the partitioning algorithms contains vertex or edge IDs along with partition IDs. The framework also provides graph processing applications mentioned in the survey paper P1. Our framework is built using the Gelly-Streaming API[3] which was developed as a part of research by Daniel Bali, Vasikili Kalavri and Paris Carbone [47]. The Gelly-Streaming API

[1] https://github.com/thorsteinnth/road-congestion-detection
[2] https://github.com/Zainab-Abbas/gelly-streaming
[3] https://github.com/vasia/gelly-streaming

combines features of the graph processing API, i.e., Gelly, and the stream processing API of Flink. The author of the thesis also contributed to developing few functionalities of the API.

- Streaming graph partitioning using Graph Convolutional Networks (GCNs): We provide the code[4] of our GCN based partitioning method called as *GCNSplit* as open-source and make all models and experiments publicly available. *GCNSplit* is an ML-based streaming graph partitioning framework that can partition large graph streams keeping very little state in memory and it generated good partitioning quality in terms of few graph cuts. *GCNSplit* work with edge stream as input and generates partitioned graph streams.

1.7.3 Other Papers

Other works published in conferences during the doctoral studies which are not part of this thesis is the following.

1 **Graph Representation Matters in Device Placement**, Milko Mitropolitsky, Zainab Abbas, and Amir H. Payberah. 2020. In Proceedings of the Workshop on Distributed Infrastructures for Deep Learning (DIDL'20). Association for Computing Machinery, New York, NY, USA, 1–6. [48]

2 **Privacy Preserving Time Series Forecasting of User Health Data Streams**, Sana Imtiaz, Sonia-Florina Horchidan, Zainab Abbas, Muhammad Arsalan, Hassan Nazeer Chaudhry, Vladimir Vlassov. IEEE International Conference on Big Data, 2020. [49]

[4] https://github.com/anonymous7CD/anonymous-repo

CHAPTER

2

Background

"Actually you can run faster than you run"

— Vladimir Vlassov a.k.a Vlad when I wrote substantial part of my thesis.

This chapter presents the necessary background by first introducing graph partitioning and graph partitioning techniques that are applicable for both snapshots of graphs or graphs at rest and streaming graphs. Next, this chapter gives information about streaming graph partitioning and its applicability to the load-computer-store model used in various distributed graph processing systems, that include, Pregel [50], GraphX [20] and Giraph [51], and the stream processing model used in systems, such as Flink [30], Storm [52] and Naiad [10]. These systems support various graph partitioning techniques. In this thesis, we focus more on methods and corresponding algorithms and use these systems as tools to implement these algorithms. In the end, we explain ML-driven graph partitioning that leverages graph representation learning. For convenience, the notations used in this chapter and Chapters 5 and 6 are presented in Table 2.1.

2.1 Graph Partitioning

Graph partitioning is the process of dividing a graph into a predefined number of subgraphs. Graph partitioning is essential for graph analysis using parallel and distributed algorithms. Distributed graph processing has been widely adopted in recent years and enables knowledge extraction from large and medium-scale graph-structured datasets using commodity clusters [50, 19, 20, 51, 53]. In such settings, each cluster node operates on one partition in parallel and communicates with other nodes through message-passing. Hence, partitioning quality directly affects communication and computation costs and is crucial for graph application performance [19, 2].

Table 2.1 – The notation used in this paper

Symbol	Description
G	input graph
$m = \|E\|$	number of edges in G
$n = \|V\|$	number of vertices in G
k	number of partitions, $k \in \mathbb{N}$
P_i	set of vertices or edges in a partition i, $i \in [1, k]$
$N(v)$	set of neighbors of a vertex v
$S(v)$	set of partitions containing vertex v
C	partition capacity
$\mathbb{L}$	loss function
z_v	an embedding vector of vertex v
$\mathbf{Y}$	assignment probability vector
$\mathbf{D}$	degree vector of vertices
W^l	embedding model matrices (l layers)

Definition 2.1.1. Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and k machines of capacity C, so that the total capacity kC is sufficient to store the whole graph, a partitioning algorithm splits G into k partitions, P_i, so that $P_1 \cup \ldots \cup P_k = G$ and $P_i \neq \emptyset$.

For convenience, we often refer to each individual partition by its index i. The partitions, P_i, are not always disjoint, they can contain a copy of the same vertex or edge. In definition 2.1.1, $P_i \cup P_j = V_i \cup V_j, E_i \cup E_j$. An *offline* graph partitioning algorithm accepts the complete graph G as input and typically computes the partitioning in multiple passes. For example, iterative clustering and community detection methods are often used to compute high-quality partitions. In contrast, a *streaming* graph partitioning algorithm processes the graph as a *stream*, a sequence of edges or vertices, and maps each element to a partition index i on-the-fly.

2.1.1 Graph Partitioning Techniques

There exist two main approaches to graph partitioning (in a broad spectrum and not limited to stream ingestion), namely *vertex partitioning* also know as *edge-cut partitioning*, and *edge partitioning* also know as *vertex-cut partitioning*. Both approaches aim to minimize cross-partition dependencies by defining a *minimum-cut* optimization objective. In the case of vertex partitioning (edge-cut), the minimal cut optimization objective is to minimise the number of edges crossing partition boundaries, i.e., the edges with one end-vertex placed in a partition different from the other end-vertex; whereas in the case of edge partitioning (vertex-cut), the objective is to minimise the number of vertices crossing partition boundaries, i.e., the vertex which is placed in more than one partition.

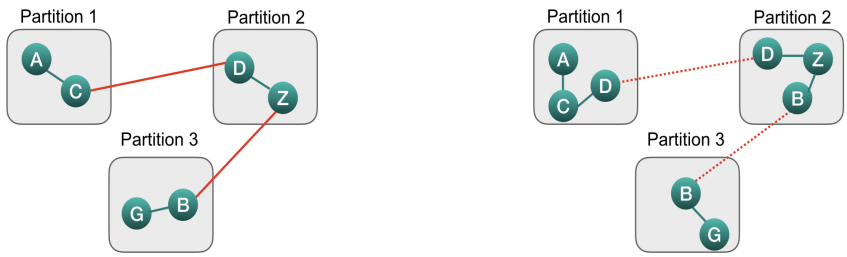


Figure 2.1 – Vertex partitioning (left) assigns vertices to partitions, possibly creating edge-cuts; Edge partitioning (right) places edges to partitions, possibly creating vertex-cuts. Both partitioning techniques are done using a greedy algorithm explained in Chapter 5.

Vertex Partitioning. Vertex partitioning [54] operates on the vertex set V, assigning each vertex to a partition i. Edges can cross partition boundaries, as illustrated in Figure 2.1 (left), where edges (c, d) and (b, z) are *cut* by the partitioning. Therefore, vertex partitioning is also know as *edge-cut partitioning*.

Definition 2.1.2. For a graph G, the *edge-cut* $E' \subseteq E$ is a set of edges, such that $G' = (V, E \backslash E')$ is disconnected.

Here, by disconnected we mean that there is no common edge or vertex between G and G'. The fewer edges crossing partition boundaries the lower the communication overhead, considering distributed graph processing using a vertex-centric model with message-passing along edges. Thus, the main optimization objective of vertex partitioning methods is the **minimum edge-cut**.

Edge Partitioning. Analogously, edge partitioning [55] operates on the edge set E, assigning each edge to a partition i. In the case of edge partitioning, two edges incident with the same vertex can be assigned to two different partitions that cause the vertex to be cut as illustrated in Figure 2.1 (right). Therefore, edge partitioning is also known as vertex-cut partitioning. References to the same vertex in different partitions are also known as *mirrors*.

Definition 2.1.3. For a graph G, the *vertex-cut* $V' \subseteq V$ is a set of vertices such that $V \backslash V'$ along with $E' \subseteq E$, the set of incident edges, make $G' = (V \backslash V', E \backslash E')$ disconnected.

Here, by disconnected we mean that there is no common edge or vertex between G and G'. The fewer mirror vertices the lower the communication overhead, considering a distributed graph processing with an edge-centric programming model. Thus, the main optimization objective for edge partitioning is **minimum vertex-cut**.

2.2 Streaming Graph Partitioning

Apart from the edge-/vertex-cut optimization, graph partitioning can also include load balance optimization, i.e., assigning the same number of edges or vertices to the partitions, by splitting a graph into evenly sized subgraphs to balance the load in parallel and distributed graph processing. *Balanced graph partitioning* is an instance of the graph partitioning problem that tries to optimize for both load balance and minimum cuts and it is an NP-hard problem [56]. Offline graph partitioning methods have access to the entire graph and iteratively refine partitions by re-assigning nodes and edges in each iteration in order to achieve optimal partitioning. Techniques range from exact and slow to approximate and fast (heuristics) [57, 58, 45]. Streaming graph partitioning methods, on the other hand, ingest a graph as a stream of either vertices or edges and partition it in an *online* fashion [2, 59, 23, 60, 24, 61]. As edges and vertices arrive continuously, online partitioners cannot iterate over the entire graph and need to make partition assignment decisions on-the-fly. Thus, they rely on heuristics and *state* of streaming graph partitioner that is a function of the history of earlier partitioning decisions.

A graph input stream can arrive either in the form of edges, as a sequence of interconnecting edges; or in the form of vertices, as a sequence of vertices each with a corresponding adjacency list. The edge-centric stream representation of massive graphs as a stream of edges is more favourable to process than the vertex-centric stream and does not require prior knowledge of graph properties, such as the number of nodes, edges and nodes' degree information. However, all non-trivial methods suffer from an increasing state size which must be kept in memory to perform partitioning. Each time the algorithm processes a new vertex, it must add new data to the state, creating $O(|V|)$ memory complexity. This memory complexity becomes a bottleneck for modern distributed stream processing systems as it increases the communication cost between the parallel partitioning instances since all memory updates are communicated. Thus, neither existing edge-centric partitioning algorithms nor vertex-centric ones can efficiently handle truly unbounded data.

Streaming graph partitioning is applicable to both the *load-compute-store* (batch) graph computational model, used in systems such as Pregel [50], GraphX [20] and Giraph [51], and the stream processing model, used in systems such as Flink [30], Storm [52] and Naiad [10]. Figure 2.2 shows the workflow of staged and pipelined

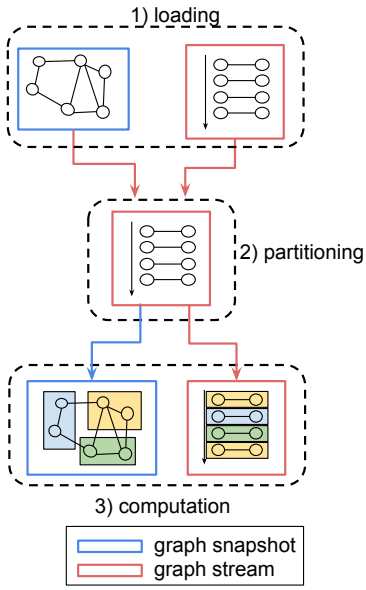


Figure 2.2 – A general workflow for graph snapshot and stream loading, partitioning, and computation.

phases for these models. In the case of batch processing, graph loading, partitioning, and computation, happen in separate consecutive stages. For stream processing, stages are pipelined and data is continuously passed as a stream from one stage to the next.

2.2.1 Loading

During the loading phase, graph is read from the disk or other external source and placed onto the computation cluster. In batch processing, the graph data is bounded and once loaded it represents a graph *snapshot* (e.g. the Facebook social network at a given time). In stream processing, graph data is continuously read from an external source and can be potentially unbounded (e.g. live user interactions on Twitter). A graph stream can be represented either as a sequence of edges (edge stream) or as a sequence of vertices with their adjacency lists (vertex stream). In essence, the streaming model subsumes the batch model, since a graph snapshot is merely a bounded graph stream. Hence, graph properties, such as the number of vertices n, the number of edges m, and the degree distribution can be computed before partitioning for a graph snapshot, while these properties continuously evolve for an unbounded stream.

2.2.2 Partitioning

During the partitioning phase, the partitioner takes a graph stream as input and assigns each vertex or edge to a partition. The decision is made on-the-fly by processing each element only once. In many cases, the partitioning logic can be implemented inside the graph loader, so that loading and partitioning happen in a single phase. Partitioners can base their decision on the current element or they can maintain *state*. Stateful partitioners [21, 22, 19] consider the history of the stream seen so far. For example, in order to properly balance the number of elements per partition, a partitioner might store the current available capacity per partition. In principle, the state can be distributed among parallel partitioner instances, where each instance has a partial view of the stream, or a global view shared across parallel instances. As our analysis reveals, existing stateful streaming partitioning methods require a shared state, which is a feature not available in modern distributed stream processors [24, 61]. Thus, partitioning logic needs to be executed by a single instance. Moreover, many methods often assume that global graph metrics are available before partitioning. These characteristics pose a major challenge in adapting existing methods for distributed processing of unbounded graph streams.

2.2.3 Computation

Computation takes place after loading and partitioning. In the batch model, the computation starts after the whole graph has been loaded and partitioned, in a subsequent stage, and it operates in one or multiple passes (e.g. bulk synchronous model with fixpoint termination). In contrast, in the streaming model, application logic is triggered on-the-fly, per graph element (a vertex or an edge), in a pipelined fashion after the partitioning step. In this case, graph elements are only accessed once. Therefore, applications that employ on-the-fly processing are also referred to as single-pass streaming applications (the term semi-streaming [62] is also used to describe a constant number of graph stream passes).

2.3 ML-driven Graph Partitioning

ML-driven graph partitioning is a new exciting research area, which refers to using machine learning for the purpose of graph partitioning [63, 64]. Recent breakthroughs in representation learning, such as GraphSAGE [65], have enabled effective dimensionality reduction for large graphs and shown promising predictive performance capabilities. The essence of inductive Graph Convolutional Networks (GCN) is to exploit features associated with vertices and edges as well as the graph structure to build convolutional neural networks that summarize the graph. GAP papers [63, 64] make use GCNs to perform graph partitioning offline. Other than the

offline approaches outlined in the GAP papers [63, 64], we are not aware of closely related work. In the remaining part of this section, we explain graph representation learning followed by a popular GCN method GraphSAGE. In the end, we present the use of GraphSAGE for graph partitioning from the GAP paper [63, 64].

2.3.1 Graph Representation Learning

Graph representation learning methods automatically learn to encode graph structure and properties into $n-$dimensional vectors [66, 65, 67]. These low-dimensional *embeddings* can be then handled by task-specific downstream ML algorithms. Embeddings can encode nodes, edges, subgraphs or the entire graph. Node embedding techniques encode graph vertices so that node similarity is preserved in the embedding space. As such, their objective function aligns with that of partitioning algorithms, which aim to assign similar nodes to the same partition.

Due to the streaming nature of our problem, we apply *inductive* graph representation learning based on GCNs. This approach is capable of successful generalization to instances unseen during the training. Convolutional inductive methods represent nodes as functions of their neighborhood while utilizing node features or attributes. Given *certain* neighborhood of an unseen node, such encoders can generate a useful embedding, making them scalable and amenable to parallelism [65].

GraphSAGE. We briefly describe GraphSAGE [65], the inductive GCN that lies at the core of our proposed ML-driven graph partitioning framework, i.e., *GCN-Split*. GraphSAGE has been successfully applied in various real-world scenarios [68] and relies on fixed-sized uniform neighborhood sampling. Restricting the neighbourhood size makes it practical for large and skewed graphs.

GraphSAGE and similar GCN frameworks rely on the notion of *neighborhood aggregation*. Let us consider the generation of an embedding vector z_v for a node v. z_v is first initialized using the raw input features of v and subsequently, its embedding is refined in an iterative manner as follows. At every iteration, v gathers the embeddings of a subset of its neighbors and aggregates them into a single vector. The aggregated vector is further combined with node v's embedding vector z_v that is updated through a fully connected neural network. As iterations proceed, the embeddings capture topological and feature information from distant neighbors [65].

The training procedure can be supervised or unsupervised. In unsupervised base case of GraphSAGE, the aim is to optimize a graph-based loss function which utilizes negative sampling. Such a function leads to representations of nodes which are similar if the nodes of the original network are close and dissimilar if the nodes are distant:

$$\mathbb{L} = -\log(\sigma(z_u^T z_v)) - Q \cdot \mathbb{E}_{v_n \sim S_n(v)} \log(\sigma(-z_u^T z_{v_n}) \tag{2.1}$$

where node v is close to node u during a fixed-length random walk, S_n is a distribution of negative sampling, σ is the sigmoid function, and Q represents the number of negative samples. Here, the loss function can be replaced by a different loss function depending on the training goal.

2.3.2 GCN-based partitioning

The use of GCNs for partitioning is a recent research topic with very promising results. Next, we briefly present GAP [63, 64], the first work that formulated a cut-based loss function that *GCNSplit* relies upon.

Naive formulation of a cut-based loss function can generate unbalanced partitions as it favors the disconnection of small sets of isolated nodes. GAP avoids this situation by adding a normalizing factor to reduce bias. The cut cost is a fraction of the total number of edge connections to all nodes, called *association* [69] or *volume* [70]. The formula for the normalized cut cost with k partitions is as follows:

$$Ncut(P_1, P_2, ..., P_k) = \sum_{i=1}^{k} \frac{cut(P_i, \bar{P_i})}{vol(P_i, V)} \tag{2.2}$$

The $vol(P_i, V)$ is defined as the total number of edge connections between nodes in P_i and V, which can be represented as the total degree of nodes belonging to P_i in graph G.

$$vol(P_i, V) = \sum_{v \in P_i} d_v \tag{2.3}$$

where d_v represents the degree of node v. To utilize the normalized cut for model training, we apply several transformations to Equation 2.2. The output of the model is $\mathbf{Y} \in \mathbb{R}^{n \times k}$. First, we represent the Ncut in terms of $\mathbf{Y}$. In the output of the partitioning network, $\mathbf{Y}_{ai}$ represents the probability that a node v_a is part of a partition P_i. The probability that the node v_a is not part of the partition P_i is equal to $1 - \mathbf{Y}_{ai}$. Therefore, the expected value of a cut is defined as follows:

$$\mathbb{E}[cut(P_i, \bar{P_i})] = \sum_{v_a \in P_i;\, v_b \in N(v_a)} \sum_{j=1}^{k} \mathbf{Y}_{aj}(1 - \mathbf{Y}_{bj}) \tag{2.4}$$

where $N(v_a)$ represents the sampled neighborhood of node v_a. Using adjacency matrix notation (A), Equation 2.4 can be rewritten as follows:

$$\mathbb{E}[cut(P_i, \bar{P_i})] = \sum_{reduce-sum} \mathbf{Y}_{:,i}(1 - \mathbf{Y}_{:,i})^T \odot A \tag{2.5}$$

Equations 2.4 and 2.5 are equivalent, because the element-wise multiplication with adjacency matrix ($\odot A$) guarantees that only the nodes of the sampled neighborhood are taken into account. The result of such a product is a square matrix

whose side length equals the number of nodes in the graph. The final value is a sum over the elements of this matrix.

Although we have managed to incorporate Y into the cut equation we still need the normalizing factor. From Equation 2.3 we know that we need to utilize the degrees of the nodes to perform normalization. Therefore, suppose that $\mathbf{D}$ is a column vector where each value a corresponds to node v_a's degree. Using the product of the matrices $\mathbf{Y}$ and $\mathbf{D}$, we compute the expected value of volume for each partition as $\mathbb{E}[vol(P_i, V)] = \Gamma_i$, where $\Gamma = \mathbf{Y}^T\mathbf{D}$ and Γ_i is the ith element in the vector Γ.

According to Equation 2.2 we have both parts of the normalized cut, i.e., the minimum cut and the volume. Combining both lead to an equation as follows:

$$\mathbb{E}[Ncut(P_1, P_2, .., P_k)] = \sum_{reduce-sum} (\mathbf{Y} \oslash \Gamma)(1 - \mathbf{Y})^T \odot A \tag{2.6}$$

where $\oslash$ represents an element-wise division. Minimizing the loss function of Equation 2.6 could lead to unbalanced partitions and even assign all nodes to the same partition. To address this issue, GAP introduces a balancing term, which acts as *regularization*. Given $|V|$ nodes and k partitions, perfectly-balanced partitions would contain exactly $\frac{|V|}{k}$ nodes. The sums of the columns of Y represent the expected number of nodes in each partition. The equation which considers the perfectly-balanced partition size looks as follows:

$$\sum_{i=1}^{k}(\sum_{a=1}^{n} \mathbf{Y}_{ai} - \frac{n}{k})^2 = \sum_{reduce-sum} (1^T\mathbf{Y} - \frac{n}{k})^2 \tag{2.7}$$

We get the loss function by combining the normalized load equation 2.6 and the equally loaded partition error equation 2.7.

$$\mathbb{L} = \sum_{reduce-sum} (\mathbf{Y} \oslash \Gamma)(1 - \mathbf{Y})^T \odot A + \sum_{reduce-sum} (1^T\mathbf{Y} - \frac{n}{k})^2 \tag{2.8}$$

The model and loss function we have described so far can be used to partition static graphs by assigning nodes to partitions.

CHAPTER

3

Scalable Large Spatial Time Series Forecasting Using Graph Partitioning

"First golden rule of writing; Don't lie..."

— Peter Van Roy during our thesis writing discussion session.

A large amount of data is being generated mainly through social media, the World Wide Web (WWW), and the Internet of Things (IoT) devices. IoT data is usually in the form of time series generated from various spatial locations. Performing a learning task on a large scale spatial time series is a non-trivial task because it is computationally expensive to do on a single machine. Especially if the task involves deep learning, it becomes computationally and memory intensive to train the model. Data parallelism helps to make training and inference parallel, and can in turn improve training time. In this chapter, we work with processing large scale spatial time series data generated by huge complex systems. We use a graph to model these systems. We apply graph partitioning for efficient processing of large spatial time series data since graph partitioning is useful to parallelize and scale the computations over time series data. Partitioning spatial time series is challenging due to the fact that we need to model the spatial and temporal dependencies in the data [13, 16, 17] and perform careful partitioning to preserve these dependencies for accurate time series forecasting. We provide an efficient time series partitioning solution to scale time series forecasting with improved performance.

3.1 Introduction

Deep neural networks (NN) have shown promising results for different machine learning and data mining tasks, such as classification and prediction, in various application domains. Modelling of a large complex system requires a large dataset

to train a deep NN with many parameters [71]. At scale, training deep NNs is computationally and memory intensive. Partitioning and distribution is a general approach to the challenge of scale in NN-based modelling. A number of methods have been proposed to achieve scalability, such as distributed and collaborative machine learning [72, 73] that rely on dividing the problem into smaller tasks. These tasks comprise smaller models working on subsets of data.

In this work, we address the scalability and improve the performance of deep learning models in large-scale spatial time series forecasting. Spatial time series are multiple time series that correspond to different spatial locations [13]. There are dependencies between spatial time series that need to be taken into account when building an NN-based prediction model. Moreover, often there are real-time requirements for traffic data analysis and forecasting that put constraints on the training and inference time. This requires a scalable solution for processing a large amount of data with low latency. Our approach to tackling the scalability problem is to partition time series data while preserving essential dependencies between them in order to perform training in parallel on the partitioned data. To improve the performance in terms of prediction accuracy we aim to place correlated data in the partitions.

Application. Our application domain is the analysis of road traffic data collected by road infrastructure sensors and modelling of traffic flow behavior for the task of traffic prediction. Accurate traffic predictions can further help in route planning, traffic congestion reduction, air pollution reduction, infrastructure planning, and other tasks. We believe that our proposed partitioning technique, to achieve scalability and better performance for traffic prediction, is general and can be applied to partition and model a complex system that can be represented as a directed weighted graph of dependencies between spatial time series generated by components of the system. Other systems where we believe we can apply our partitioning approach include air traffic control systems, fitness trackers' data networks etc.

We work with real-life large data-sets generated by traffic sensors deployed in Stockholm and Gothenburg, Sweden. The number of sensors and the number of measurements from the sensors are increasing. For example, the number of infrastructure sensors in the Motorway Control System (MCS) deployed on highways in Stockholm and Gothenburg, Sweden, has increased from about 800 in 2005 to more than 2000 in 2016. The number of measurements has also increased from 400 million to about 1 billion per year, as shown in Figure. 3.1.

Sensor data are spatial time series, and having a large number of sensors causes a scalability problem in traffic time series forecasting. Due to real-time requirements for traffic data analysis and forecasting, a scalable solution for processing a large amount of data with low latency is required. In some cases, to achieve scalability, the data is partitioned in order to perform mining or modelling tasks in parallel.

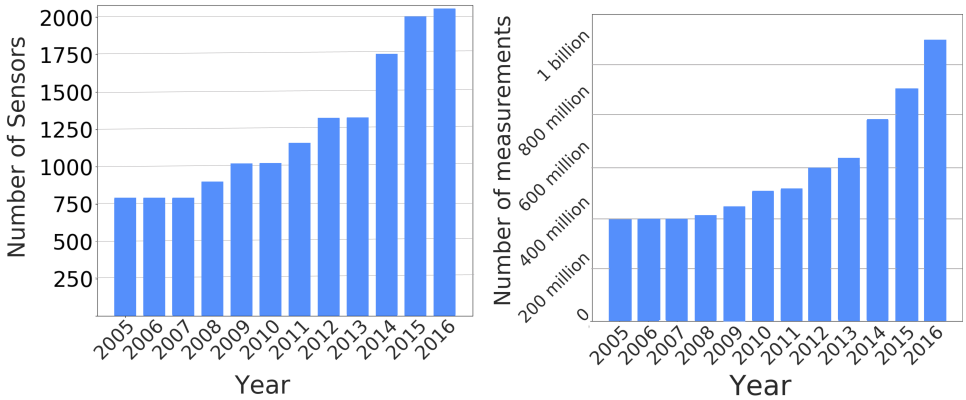


Figure 3.1 – Number of sensors' measurements over years.

Careless partitioning causes degradation in the model accuracy. For example, too small partitions might not provide enough information to train the model; Placing unrelated data, such as traffic data from unconnected road segments, in the same partition might negatively affect the accuracy or complexity of the model. The negative effect can happen because uncorrelated data will make it hard for the learning model to learn trends in the data which will affect the overall accuracy. Also, adding uncorrelated data will increase the dataset size and in the case of large scale data, it will increase the parameters of the learning model, thus making it complex and hard to train. Careful partitioning is especially important for grouping time series data of multiple sensors (spatial time series), because of dependency between the sensor readings. One example of such a dependency is that a moving car counted by one sensor will be counted by the next sensor in the flow direction. Another example is that a traffic queue growing in the opposite direction of the traffic flow causes a slowdown of cars and, as a consequence, a dependency between sensor readings. The dependency is strong between sensors which are closely placed and there is a path between them; the dependency is weak between sensors which are far apart or have no path between them. Taking these dependencies into account is important for the partitioning of sensors, meaning that a partition should include correlated sensors.

We propose a partitioning technique to tackle the scalability problem that enables parallelism in training and prediction: 1) We represent the sensor system as a directed weighted graph based on the road structure, which reflects dependencies between sensor readings, and weighted by sensor readings and inter-sensor distances; 2) We propose an algorithm to automatically partition the graph taking into account dependencies between spatial time series from sensors; 3) We use the generated sensor graph partitions to train a prediction model per partition.

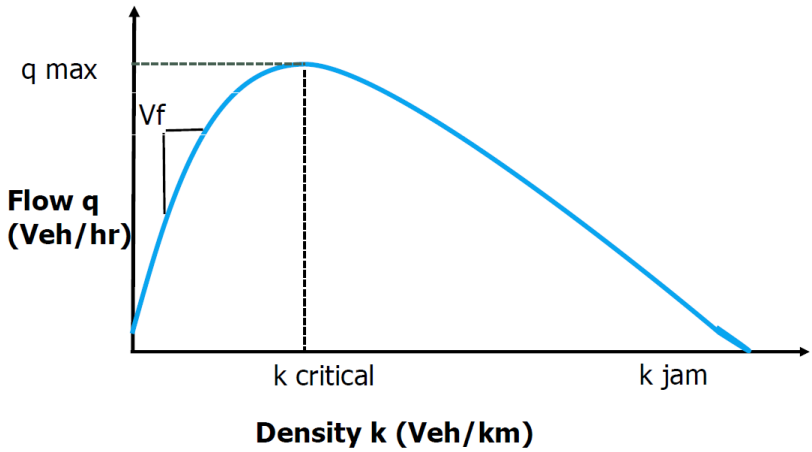


Figure 3.2 – The fundamental curve of road traffic flow

3.2 Preliminaries

In this section, we present the necessary background on the traffic flow theory, along with details of the NN-based prediction model we use in this work for predicting road traffic, and the input representation fed to the prediction model for capturing spatio-temporal dependencies between road traffic data.

3.2.1 Traffic Flow Theory

Traffic flow theory is the study of vehicles' behaviour on road; it helps to explain the vehicle flow and the interaction of vehicles with each other. Mainly three traffic variables are used to explain the vehicles' movements on road, namely [74]: 1) traffic flow q (number of vehicles per unit time) that is the number of vehicles passing a particular point on road, 2) density k (number of vehicles per unit distance) that is the concentration of vehicles on road, and 3) speed v (distance covered per unit time). The three variables are related as:

$$q = k \times v \tag{3.1}$$

The fundamental traffic flow theory diagram, as shown in Figure 3.2, gives us a useful relation between the three traffic flow theory variables. At the start of the curve, the flow q of cars increases along with the density k; during this phase the vehicles move with free-flow speed V_f, represented by a positive slope on the curve. When q increases further the density k reaches its critical value $k_{critical}$. At this point, the flow is maximum, i.e., q_{max}. Beyond $k_{critical}$ the vehicles' movements

become restricted because their concentration is increasing on the road, thus we see a decrease in the speed with a negative slope. k_{jam} indicates the traffic jam density, at this point the speed is very low due to congestion.

3.2.2 Long Short-Term Memory NN-Based Prediction Model

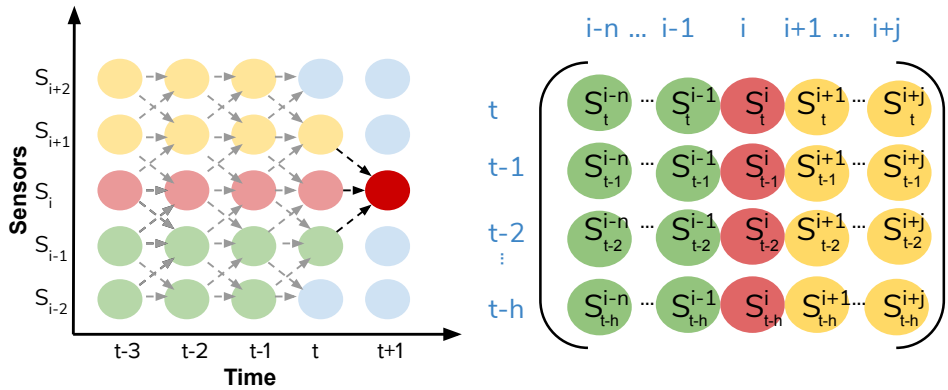
Traditional time series forecasting techniques for traffic prediction, including flow theory-based models and statistical techniques, such as Bayesian analysis [75], Markov chains [76] and ARIMA [77], have been replaced by neural networks. NNs perform better for time series forecasting than the traditional techniques because the latter are incapable of handling missing and multidimensional data. NNs have been employed in forecasting time series data [78, 79]. In particular, Long Short Term Memory (LSTM) networks [80] have shown promising results for traffic prediction due to their deep hierarchical structure which enables the extraction of non-linear and stochastic characteristics of the traffic data [81, 82].

In this work, we use LSTM-based prediction models. LSTM is a type of Recurrent Neural Network (RNN) architecture that is capable of learning long term trends in the data. In our previous work [25], we compared a 2 layered stacked LSTM based architecture with various statistical and neural networks based models. The stacked architecture containing 2 layers gave better accuracy compared to other models.

Complexity. The basic LSTM architecture consists of three layers: the input layer, LSTM (hidden) layer, and output layer. Data from the input layer is fed to the LSTM layer which contains memory blocks comprising of memory cells with self-connections and gates. These self-connections are from the cell's output unit to the input unit and gates. These gates, namely the input, output and forget gate control the flow of data across these cells which represent the state of LSTM. The output units of cells are connected to the output layer. The computational complexity of an LSTM network per time stamp and weight is $O(1)$ [80]. Therefore, the complete learning complexity of LSTM with a total of W number of parameters is $O(W)$. W is computed by the equation [83]:

$$W = n_c^2 \times 4 + n_i \times n_c \times 4 + n_c \times n_o + n_c \times 3 \tag{3.2}$$

Here, n_c is the number of memory cells, n_i is the number of input units and n_o is the number of output units in the LSTM layer. A large number of input, output and memory units can increase the computational complexity of LSTM.



(a) Spatio-temporal dependencies of sensor data

(b) Space-Time window representation of sensor data

Figure 3.3 – Data representation of spatial time series

3.2.3 Data Representation for Spatial Time Series Forecasting

Time series data collected from road traffic sensors contain both spatial and temporal dependencies. To capture these dependencies, we need to take into account the sensor's previous readings, the readings of neighbouring sensors and neighbouring sensors' previous readings. Therefore, we model the sensor data using Equation 3.3, which represents the time series dependencies in space and time. Using this equation, we consider that the predicted reading of a sensor S_i at time $t+1$ is given by the following equation:

$$\begin{aligned} S_{i,t+1} = f(&S_{i-n,t}, S_{i-(n+1),t}, ..., S_{i,t}, S_{i+1,t}, ..., S_{i+j,t}, \\ &S_{i-n,t-1}, S_{i-(n+1),t-1}, ..., S_{i,t-1}, S_{i+1,t-1}, ..., S_{i+j,t-1}, \\ &..., S_{i-n,t-h}, S_{i-(n+1),t-h}, ..., S_{i,t-h}, S_{i+1,t-h}, ..., S_{i+j,t-h}) \end{aligned} \tag{3.3}$$

Here, $S_{i+1}, ..., S_{i+j}$ are readings of the sensors placed downstream, i.e., in the direction of traffic flow, $S_{i-1}, ..., S_{i-n}$ are readings of the sensors placed upstream, i.e., in the opposite direction of traffic flow, and $t, ..., t-h$ refers to h time units in the past from the current time t. Figure 3.3a shows that the prediction of the sensor reading S_i at time $t+1$ depends on the readings from neighbouring sensors placed downstream (S_{i+1} and S_{i+2} shown in yellow), sensors placed upstream (S_{i-1} and S_{i-2} shown in green) and sensor S_i's previous readings in time intervals $t, ..., t-h$ (in red).

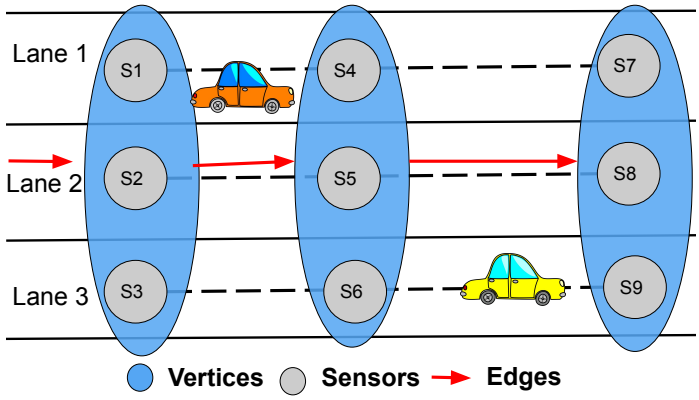


Figure 3.4 – Sensor graph creation

We map Equation 3.3 to a space-time window which is fed as an input to train the prediction model. For a sensor S_i on a particular point in the highway, the space-time window contains sensor readings, i.e., the density of vehicles computed using the flow and average speed recorded by the sensor, from $i - n$ sensors placed upstream and $i + j$ sensors placed downstream. These readings are taken for a history of h time stamps from the current time t. As shown in Figure. 3.3b, the first row of the window contains sensor readings from $S_{i-n}...S_i...S_{i+j}$ at a current time interval t. Similarly, the next row is for readings at $t - 1$, i.e., the previous minute. The rows following this contain previous readings till $t - h$ time interval. Once the input is fed, the prediction model is trained to predict for $t + n$ future time intervals traffic density.

3.3 Graph Representation of Road Traffic Network

We represent the traffic infrastructure sensors in the form of a directed weighted graph to capture the spatio-temporal dependencies between the time series from the sensors. We construct the graph G based on the road paths and the traffic direction between the sensors. Next, we weight the graph using sensor readings and distance between the sensors. Sensors that are at the same location but on different lanes are represented as a vertex in V. The paths between these vertices are represented as edges in E. The direction of the traffic flow determines the direction of the edges. Figure 3.4 shows the sensors on parallel lanes at the same location grouped together as sensor site vertices (in blue) and the road between them are represented as directed edges (in red). Figure 3.5 depicts a high-level view of the sensor graph of the Stockholm centre.

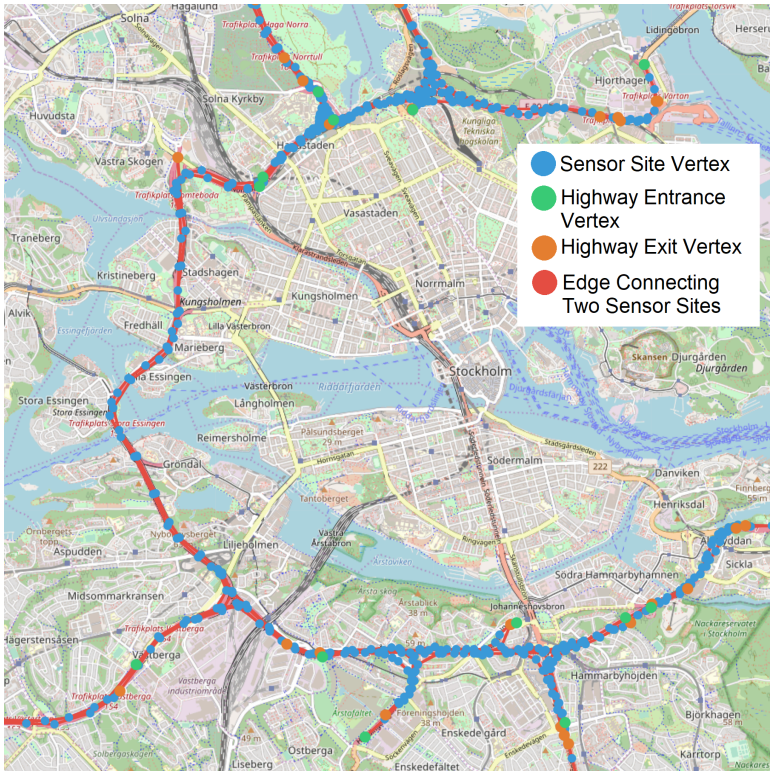


Figure 3.5 – Graph representation of road sensors in Stockholm

Once the graph is constructed, the edges are weighted based on the travel time of the cars. The edge weight is equal to the time it takes a vehicle to travel the road segment corresponding to that edge. The weight $w(i, j)$ represents the weight of an edge directed from the vertex v_i to v_j. $w(i, j)$ is calculated using the average speed v of vehicles recorded by the sensors corresponding to the vertex v_j during rush hours and distance d between the sensors corresponding to vertices v_i to v_j. Equations 3.4 [84] are used to compute the distance d and travel time t for weighting the edges.

$$
\begin{aligned}
d &= 2r \arcsin\left(\sqrt{\sin^2(a) + \cos(\varphi_1)\cos(\varphi_2)\sin^2(b)}\right) \\
t &= d/v \\
\text{where } a &= (\varphi_2 - \varphi_1)/2 \text{ and } b = (\lambda_2 - \lambda_1)/2
\end{aligned}
\tag{3.4}
$$

φ_1 is start latitude in radians, λ_1 is start longitude in radians, φ_2 is end latitude in radians, λ_2 is end longitude in radians, r is Earth radius (6371 km), d is distance between sensor sites.

3.4 Road Network Graph Partitioning

We partition the directed weighted graph, that represents the traffic infrastructure sensors, in order to find groups of correlated sensors. The partitioning algorithm consists of three stages: 1) Creation of base partitions; 2) Creation of the base partitions graph; 3) Addition of partitions from the front and behind to capture the dependencies of sensors located behind (in the direction of traffic flow towards the point of interest) and in front of the selected point of interest (in the direction of traffic queue growing or moving in the opposite direction of traffic flow towards the point of interest) in the graph.

Creation of Base Partitions: The base partitions are created using backward traversal of the graph from the starting point which is a vertex with no outgoing edge. The backward traversal is made to capture the dependencies between sensor readings caused by cars moving in the direction of traffic flow. For example, a moving car counted by one sensor will be counted by the next sensor in the flow direction.

The algorithm starts by taking an input parameter time t, and chooses a starting point, which is a node/vertex with no outgoing edge from it, in the graph. The traversal is made in the opposite direction of the incoming edge towards the node. The weights of edges along the traversal are added up until they reach the threshold t. Once the threshold is reached, the edges are added to a partition and the next partition starts from the last edge visited. The algorithm terminates when all edges are visited. This results in creating base partitions.

Algorithm 1 defines the steps to create base partitions. G is the input sensor graph, v is a vertex and $e(v, w)$ represents an edge. *getStartingVertices(*G*)* returns starting vertices in G. *getBackwardVertices(*v*)* is used to get vertices visited when traversed backwards from v and *getMaxWeight(*v*)* is used to get maximum weight of the partition containing vertex v.

Figure 3.6a shows an example of the directed weighted graph created to represent the road sensors, where the weights are based on travel times of vehicles. Figure 3.6b shows the base partitions created over the weighted sensor graph with an input parameter $t = 2\ min$ and the starting vertex in red.

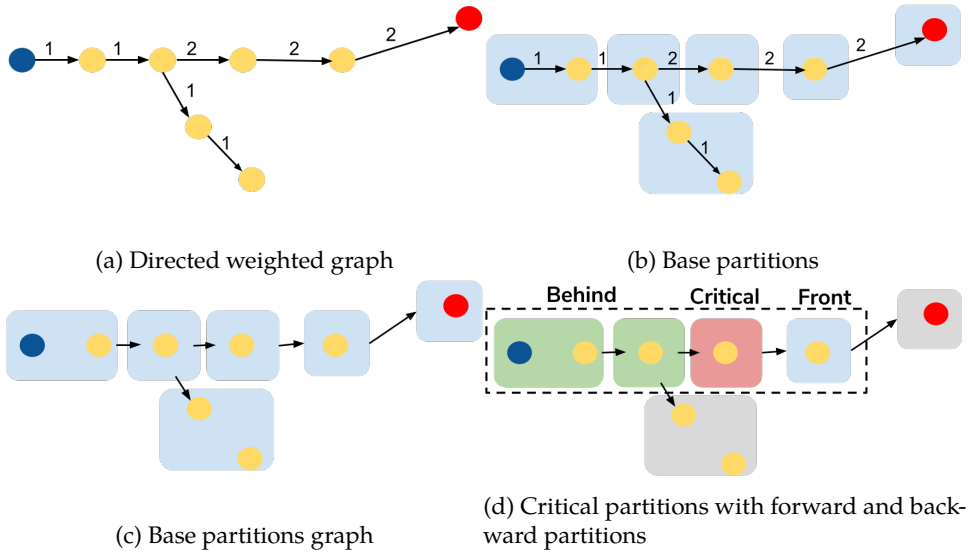
Creation of Base Partitions Graph: Once we have the base partitions, they are connected to form a base partitions graph, where the partitions are vertices and the flow directions between them are edges (Figure 3.6c).

Algorithm 1 Backward Graph Partitioning Algorithm for Creation of Base Partitions

```
1: P={}                                                    ▷ Parititions initialized
2: function main(G, s)                                     ▷ G - graph, t - input time
3:     starting_vertices=getStartingVertices(G)
4:     for each (v in starting_vertices)
5:       partition(G,v,t,0)
6:     end for
7:     collect all partitions in P
8:     return P
9: function partition(G, v, t, sum)   ▷ G - graph, v - vertex, t - input time, sum -
   accumulated sum
10:    if v is in P then
11:        return
12:    if t ≤ sum then
13:        add v to starting_vertices
14:        return
15:    add v to a partition in P
16:    next_vertices = getBackwardVertices(v)
17:    for next_vertex in next_vertices do
18:        weight = weight of e(v, next_vertex)
19:        if next_vertex not in P then
20:            partition(G, next_vertex, t, sum + weight)
21:        else if t+weight <=getMaxWeight(next_vertex) then
22:            merge the current partition with the partition containing next_vertex.
23:        else if sum+weight <= t then
24:            add vertices of partition containing next_vertex to starting_vertices.
25:            partition(G, next_vertex, t, sum + weight)
```

Additions of Partitions from Front and Behind: After we get the base partitions graph, first, we add some partitions from behind to capture the dependencies between sensor readings caused by the cars moving towards the tail of the base partition. Then, we add partitions from the front to capture dependencies between sensor readings caused by the traffic queue, created during a traffic jam, moving towards the head of the base partition. The number of partitions added from front and back affects the size of our final partitions.

A partition size is determined in such a way that it covers all correlated sensors reachable within the prediction time horizon by a car travelling at an average speed in each segment of the partition, i.e., the sensors affected by the same traffic stream are placed in the same partition if the travel time between them lies within the prediction horizon. This allows partitioning of large sensor networks while preserving dependencies between spatial time series generated by the sensors.



(a) Directed weighted graph

(b) Base partitions

(c) Base partitions graph

(d) Critical partitions with forward and backward partitions

Figure 3.6 – Partitioning steps

Table 3.1 – Configuration parameters of the prediction models

Model	**Input units**	**Output units**	**Memory units**
Single Sensor	1	1	50
Entire Sensor Infrastructure	2058	2058	1000
Partitioning-Based	partition size	critical partition size	1000

In our experiments, we make predictions for up-to 30 min. Therefore, we add partitions from the back that cover at least the distance travelled by the cars coming towards the base partition in 30 min time interval. Similarly, we add partitions at the front of the base partition by almost half the number of the ones added at the back, because the traffic queue builds up slowly and few sensors are covered by the traffic queue moving backwards, i.e., towards the head of the base partition. Figure 3.6d shows the addition of partitions from the front (in blue) and behind (in green) of the current base partition or the critical partition (in red).

3.5 Traffic Prediction Models

The prediction models we use for evaluation consist of 1) The single sensor models, 2) the entire sensor infrastructure model and, 3) the partitioning-based traffic network models. In this section, we explain various features of these models, such as the number of input units, the memory units, and the output units. Table 3.1 contains the configuration parameters of the prediction models.

Single Sensor Models. The single sensor models (SSMs) are trained per sensor, where the input is only one sensor's readings and the prediction is also done for the same sensor. These models, when considered individually, have low complexity as the input and output is only one. However, these SSMs result in a very large number of models and they collectively require high computational power for training and serving. The prediction results of this model will be based only on the readings of a single sensor and no information from neighbouring sensors is taken into account. This might result in less accurate predictions because less information is fed to train the prediction model.

Entire Sensor Infrastructure Model. The entire sensor infrastructure model (ESIM) is a huge un-partitioned model trained on readings from all infrastructure sensors and makes predictions for all the sensors. Using all sensors' readings during model training can help the model learn the correlations between the sensor readings because of their spatio-temporal dependencies. The complexity of this model is directly dependant on the number of input and output units of this model. In our case, the number of input units and output units for this model are equal to 2058, which is the total number of sensors in the road sensor infrastructure.

Partitioning-Based Models. The partitioning-based models (Bt) are trained with sensors in the partitions created after partitioning the road infrastructure graph. The partitioning algorithm creates partitions to group the correlated sensors. Creating partitions helps in reducing the number of sensors used for training the prediction model, which results in a less complex model per partition compared to the entire sensor infrastructure model. The number of input and output units for the partitioning-based model depends on the size of the partition. In our experiments, we use various input parameters for creating partitions of different sizes. More details about the input parameters used in our work are given in the evaluation section 3.6.1.

3.6 Experimental Evaluation

We evaluate the accuracy and performance of our proposed approach to predict road traffic density. We compare the proposed partitioning-based prediction models with the entire sensor infrastructure model and the single sensor models.

3.6.1 Experimental Setup

Metrics. We use the following metrics for evaluation:

- **Accuracy:** We measure the accuracy of models in terms of the Root Mean Square Error (RMSE) and Mean Absolute Error (MAE). RMSE and MAE between predicted density values $\hat{k}_{inm}$ and observed density values k_{inm} for i^{th} interval, N number of sensors over M minutes of the days are computed using the following equations:

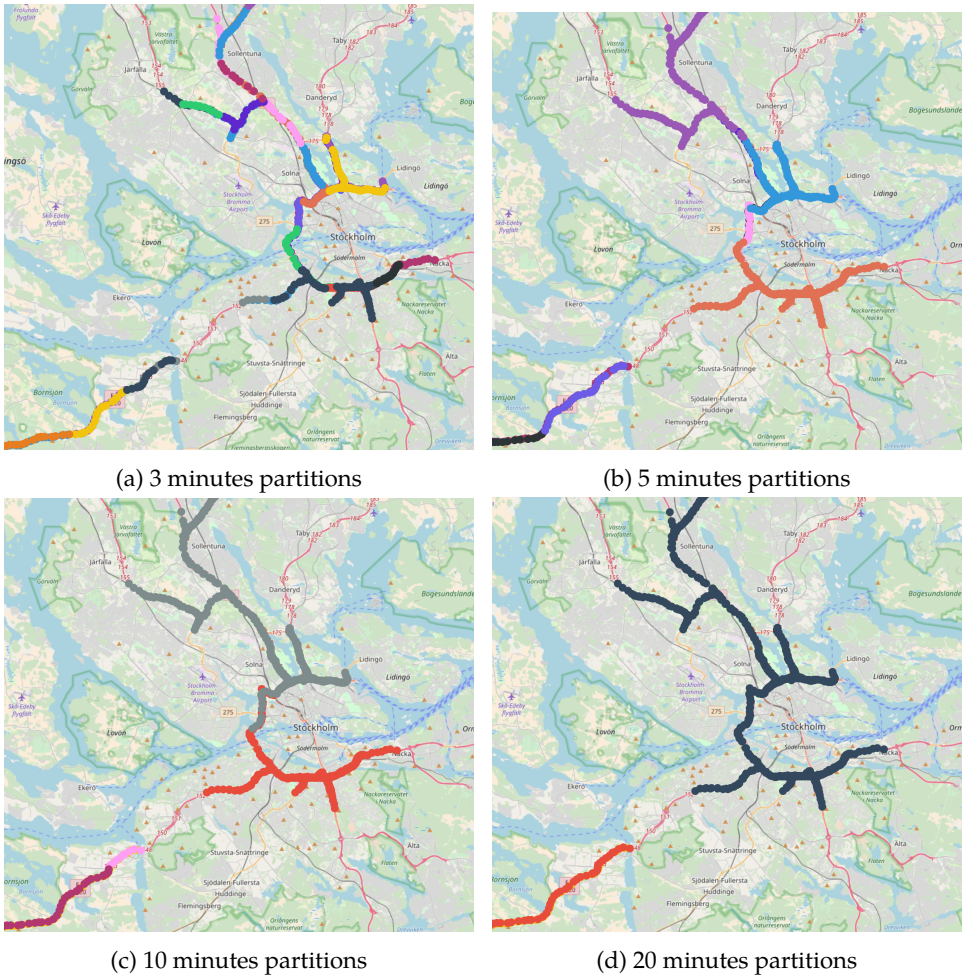
$$MAE = \frac{1}{NM} \sum_{n=1}^{N} \sum_{m=1}^{M} |k_{inm} - \hat{k}_{inm}| \tag{3.5}$$

$$RMSE = \sqrt{\frac{1}{NM} \sum_{n=1}^{N} \sum_{m=1}^{M} (k_{inm} - \hat{k}_{inm})^2} \tag{3.6}$$

- **Performance:** We measure the performance of prediction models in terms of prediction and training time.

LSTM Network-Based Traffic Prediction Model. Our prediction model is based on the stacked LSTM network architecture with two LSTM layers. In our previous work [25], we compared the 2 layer LSTM network-based model with other prediction models including classical baseline statistical models, such as ARIMA, Support Vector Regression (SVR), and neural network-based models, such as RNNs with two layers, Feed Forward Neural Network (FFN) with two layers and LSTM-1 with a single LSTM layer. LSTMs with 2 layers proved to give better prediction accuracy than the aforementioned mentioned models. Therefore, in this work, we use the same stacked 2 layers-based LSTM prediction model and address the scalability problem.

Dataset. The traffic dataset used in this work was provided by the Swedish Transport Administration [85]. The dataset consists of readings from radar sensors on Stockholm and Gothenburg highways during the period 2005-2016. The sensors are placed a few hundred meters apart from each other on each lane. They collect data per minute, that results in a large and microscopic dataset compared to data aggregated per hour or over multiple lanes. The sensor readings include the flow and average speed of vehicles per minute. The dataset used for prediction consists of more than 88 million data points collected by 2058 sensors (Figure 3.1) over a period of one month in 2016.



(a) 3 minutes partitions

(b) 5 minutes partitions

(c) 10 minutes partitions

(d) 20 minutes partitions

Figure 3.7 – Partitioned road graph using different input parameters

Partitioning Parameters. We partition the road sensor network graph to create base partitions for various values of parameter t. Figure 3.7 shows the results of partitioning for values of t = 3 min, 5 min, 10 min and 20 min. Partitions are shown in different colors, where colors of two partitions representing traffic in opposite directions may overlap. The higher the base partition creation input parameter, the bigger the partition size. Also, the total number of partitions becomes less with the increase in the input parameter. In Figure 3.7d, we can see that almost all the highway is covered with the partition shown in black color. This partition further has one more similar partition for the traffic in the opposite direction.

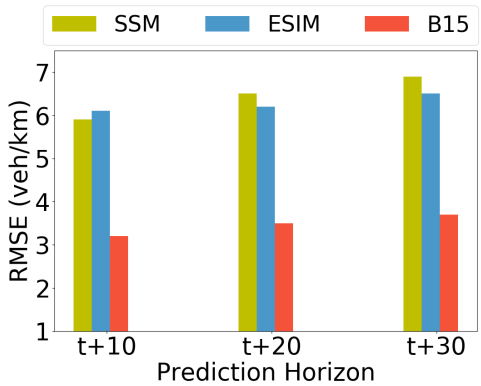


Figure 3.8 – RMSE (veh/km) of traffic density

Table 3.2 – Accuracy of Prediction Models

Model	RMSE 10 min	RMSE 20 min	RMSE 30 min	MAE 10 min	MAE 20 min	MAE 30 min
SSM	5.9	6.5	6.9	3.0	3.2	3.5
ESIM	6.1	6.2	6.5	3.1	3.1	3.2
B2	5.4	6.2	6.5	2.7	3.0	3.1
B5	5.4	5.7	6.2	2.7	2.8	2.9
B10	5.5	6.0	6.1	2.7	2.8	3.0
B15	**3.2**	**3.5**	**3.7**	**1.8**	**1.9**	**2.0**

Experimental Environment. We used on-premises nodes consisting of `Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz`, 256GB of RAM and 16.6 TB disk. We used Apache Spark version 2.4.0. with Python 2.7.15 and Tensor Flow version 1.11.0.

3.6.2 Traffic Prediction

Accuracy. Table 3.2 shows the RMSE and MAE values for the Single Sensor Models (SSMs), the Entire Sensor Infrastructure Model (ESIM) and the Partitioning-Based models (Bt) using base partitions with the input parameter $t = 2$ min, 5 min, 10 min and 15 min, denoted as B2, B5, B10 and B15, respectively. SSMs have lower errors compared to ESIM for short prediction intervals, such as 10 min; whereas, ESIM has slightly less error for 20 min and 30 min intervals. Bt models show better accuracy compared to the base-line SSMs and ESIM. Overall, B15 shows the best accuracy. Figure. 3.8 shows RMSE for the base-line SSMs and ESIM compared to B15. B15 has about 2x lower RMSE compared to the base-line approaches. Furthermore, RMSE and MAE increase with the increase in the prediction horizon.

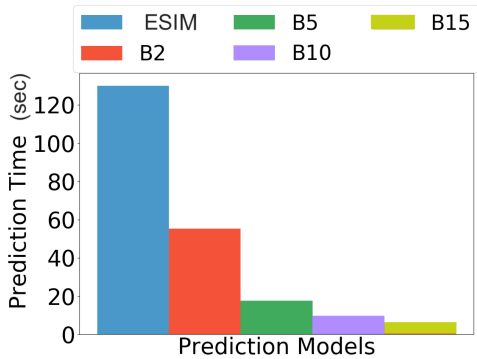
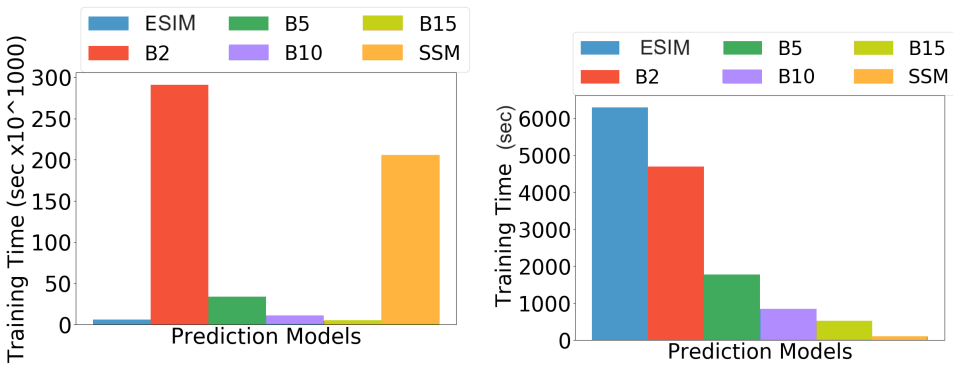


Figure 3.9 – Prediction time (sec) of different prediction models



(a) Training time (sec), if run sequentially

(b) Training time (sec), if run in-parallel

Figure 3.10 – Training time (sec) for the sequential and parallel run of different prediction models

Table 3.3 – Performance of Prediction Models

Model	**Parallel training time (sec)**	**Sequential training time (sec)**	**Prediction time (sec)**	**No. of models**
SSM	100.3	205600	<1	2058
ESIM	6300.2	6300	130.2	1
B2	4696.4	291152	55.3	62
B5	1776.1	33746	17.6	19
B10	838.2	10897	9.6	13
B15	522.1	4699	6.3	9

Performance. The performance of the prediction models is measured by computing the sequential training time, parallel training times, and the prediction time. Figure 3.9 shows the prediction times and Figure 3.10 shows the training times of SSMs, ESIM and Bt models. One SSM has the prediction time < 1 sec, thus it is omitted from the plot in Figure 3.9. SSMs have shorter parallel training and prediction times compared to other models. However, there are 2058 sensors and training/serving SSMs sequentially for all these sensors take long time and makes SSMs inefficient in terms of performance. Therefore, the training and prediction time for SSMs, if run sequentially, is very high.

ESIM has the highest training time if run in parallel, and prediction time because of a large number of sensor data being processing to train this huge network. On the other hand, the Bt models have shorter training time, if run both in parallel and sequentially with B15, and less prediction time compared to ESIM. The training and prediction time for the partitioning-based models decreases with the increase in the input parameter for base partitions creation. Table 3.3 shows the performance comparison of these models. The number of partitions for B15 is only 9 and it has less training and prediction time compared to other models. Hence, it is suitable for large-scale traffic prediction. Overall, partitioning-based models Bt take 2x, if run sequentially, and 12x, if run in parallel, less training time, and 20x less prediction time compared to ESIM. The partitioning-based models take 100x less total sequential training time compared to SSMs.

Findings. Our results for comparing SSMs, ESIM and partitioning-based models show that we can achieve a scalable traffic prediction solution using partitioning-based models. Our experimental results show that: 1) SSMs are better for very short prediction horizons compared to ESIM and become less accurate with the increase in prediction horizon. At scale, the number of SSMs gets large which makes them inefficient for use in terms of overall high training and serving time. 2) ESIM albeit high training time yields better accuracy compared to SSMs for long prediction horizons. 3) Partitioning-based models with B15 shows overall best accuracy compared to SSMs and ESIM. The performance is also better compared to ESIM, in terms of the training and prediction time. Also, compared to SSMs the performance of partitioning-based models is better in terms of the training time if run sequentially.

3.7 Related Work

Fouladgar et.al. [33] worked on scalable deep neural networks for urban traffic congestion prediction. However, they did not focus on addressing the scalability issue. Moreover, they use aggregated data over 5 min for only 58 locations. The issue of scale is still un-addressed. Other works [34, 35] done on large-scale

traffic estimation and prediction do not fully address the issue of growing model complexities at scale.

3.8 Acknowledgments

I would like to thank Ahmad Al-Shishtawy for working with us on this task of road traffic prediction. Ahmad has been a great support throughout this project and has provided immense knowledge on working with big and complex traffic datasets. Furthermore, I would like to thank our master student Jon Reginbald Ivarsson for his hard work in implementing the partitioning techniques. This work was co-supervised by Vladimir Vlassov and Sarunas Girdzijauskas who provided constant useful feedback and also helped us in shaping and writing the papers for this work.

3.9 Summary

In this work, we address the challenge of *CH1 Scaling Deep Learning Models for Time Series Forecasting*, mentioned in Chapter 1. We address the scalability problem in a real-life task of large-scale road traffic prediction using real-life large data sets generated by traffic sensors deployed in Stockholm and Gothenburg, Sweden. We propose a partitioning technique with corresponding algorithms to tackle the scalability problem that enables parallelism in both training and prediction, and hence reduces the training and model serving times while improving the accuracy of LSTM-based prediction models. We represent the road sensor infrastructure as a directed weighted graph to capture the spatio-temporal dependencies between the sensor readings; We propose a graph-based partitioning algorithm to group correlated sensors reachable within a given time horizon into partitions, to capture dependencies between spatial time series of data collected by sensors in partitions, and to train and serve prediction models for partitions in parallel. The partitioning-based prediction models are fast and more accurate compared to single-sensor models and a single non-partitioned model for the entire road sensor infrastructure.

With this real-life case study, we have illustrated that partitioning is feasible and effective to address the scalability challenge in modelling complex systems using deep learning, given that structural partitioning of data sources preserves the data dependencies, e.g., dependencies between sensor readings in the road infrastructure.

Our proposed partitioning technique in general can be applied to partition and model a complex system that can be represented as a directed weighted graph of dependencies between spatial time series generated by components of the system. For example, air traffic control systems generating spatial time series, which are correlated, can be partitioned using our proposed approach.

CHAPTER

4

Streaming Graph Analytics for Large Spatial Time Series Analysis

"Sometimes I'll start a sentence and I don't even know where it's going.
I just hope I find it along the way"
— Michael Scott, The Office.

In the previous chapter, we applied graph partitioning to scale DL models for large spatial time series forecasting generated from components of a complex system, where we modelled the complex system as a graph. Here, we present our work on performing useful analytics on large spatial time series. Large scale spatial time series analysis includes processing multiple correlated time series. These time series often contain spatial dependencies (inter-dependencies) and temporal dependencies (intra-dependencies). It is challenging to model time series data for capturing both spatial and temporal dependencies at scale because these dependencies are dynamic and are both short and long-range [13, 16, 17]. The range here is the closeness in terms of time and space. For example, spatial dependency is not always related to the physical closeness of the data sources generating the time series. Taking into account the aforementioned challenges, we propose to model spatial time series as graph streams because graphs provide a structural representation that enables complex analytics algorithms [86, 87]. We use streaming graph analytics to perform useful analyses of multiple time series at scale.

4.1 Introduction

Streaming graph analytics is an emerging application area that aims to extract knowledge from evolving networks in a timely and efficient manner [11, 12]. *Graph streams* are (possibly unbounded) sequences of timestamped events that represent

relationships between entities: user interactions in social networks, online financial transactions, driver and user locations in ride-sharing services. Graph streams are continuously ingested from external, often distributed, sources and are modeled either as streams of edges or as vertex streams with associated adjacency lists. In this chapter, we illustrate and support the practical importance and usefulness of applying streaming graph analytics to a real-life use case.

We propose to use streaming graph analytics to detect road traffic congestion and control traffic jams by processing road traffic streams as a use case to demonstrate the importance of streaming graph analytics. Streaming graph analytics, being a relatively new field of research, should be explored further for other use-cases. Few such use-cases mentioned in [12] include predicting traffic and demand for a ridesharing service and management of networks online in software-defined network controllers.

Application. Our application domain is road traffic analytics for the task of traffic congestion detection and mitigation using both real-life and synthetic data. Real-life data is collected from a region in one of the largest metropolitan cities in China. We proposed an end-to-end framework built on top of a modern stream processing system, i.e., Apache Flink. Our system comprises of 1) an online traffic jam detection mechanism for detecting jams on streaming data collected from traffic sensors using streaming graph analytics, and 2) a congestion reduction mechanism for reducing the effect of congestion in the congested area by deploying dynamic traffic light policies.

With the plethora of vehicles used to commute every day, traffic congestion has become a common sight. It is important to monitor traffic flows to prevent congestion to avoid a multitude of problems. Some of these problems include: increase in fuel consumption and pollution [88], decrease in economy [89] and traffic safety that is caused by a speed variance between cars in the congested region compared to cars moving freely [90], and harmful effects on the mental and physical health of people [91, 92].

Mitigating congestion is thus an essential task of a traffic control system. Moreover, there are real-time requirements of modern traffic control systems that require a traffic monitoring and congestion control mechanism with low latency. In this work, we investigate on real-time traffic jam detection and congestion reduction over traffic streams. Real-time congestion detection can help in sending safety warnings to drivers approaching the congested region to avoid accidents, to do daily route planning, and to deploy various policies for mitigating congestion. Once congestion is detected in real-time, congestion mitigation can be done by setting up speed limits for the vehicles approaching the congested region and by controlling the traffic lights for limiting incoming traffic towards the congested region. Therefore, an online traffic stream processing based solution is necessary in order to efficiently mitigate traffic congestion by measuring the current traffic conditions.

Most of the existing congestion detection algorithms [37, 38, 39] use historic data and are thus suitable to work offline. An online system is required to process possibly unbounded streams with low-latency, making congestion detection and mitigation challenging. One example of such offline technique is [37], which uses link journey times of vehicles to detect congestion. It requires historic information on past link journey times to detect non-recurrent congestions. Hence, it is not efficient for real-time use. In order to detect traffic jams caused by congestion in real-time, we represent the traffic infrastructure network in the form of a directed weighted graph to capture correlations between traffic sensors based on the dependencies between their generated data streams. An example of such a dependency is that a vehicle detected by one sensor will also be detected by another sensor a few moments later in the traffic flow direction. Another example of a dependency is a traffic queue moving in the opposite direction of the traffic flow during the traffic jam that causes slow down of cars approaching the end of the queue, resulting in dependency between readings of the sensors placed in the opposite direction of the traffic flow. These dependencies are taken into account by us in measuring traffic flow variables that are used for the detection of traffic jams and tracking of traffic jams' propagation.

In this work, we use traffic flow theory [74] combined with graph analytics to detect traffic jams in the streaming traffic data. Next, we develop a streaming graph-based algorithm to find correlated traffic jams in the network. We also propose a congestion mitigation/reduction mechanism by dynamically changing traffic light policies to control the traffic flow moving towards the congested region.

4.2 Preliminaries

In this section, we provide the necessary background by introducing congestion detection using fundamental traffic flow theory (cf. Chapter 3, Section 3.2.1).

4.2.1 Traffic Congestion

Traffic congestion is a state of the traffic on the road in which the vehicles cannot move freely on road, their movements are restricted, i.e., vehicles cannot easily overtake each other, change lane, or move at high speed. Congestion can be caused by a poorly designed road infrastructure that is unable to meet the traffic demand. It can also be caused by external factors, such as rainy weather, accidents, and road repair work. On the contrary to congestion, "free-flow" state is the one in which vehicles can easily overtake, change lane and increase speed [93, 94].

Congestion Detection. The fundamental traffic flow curve can be used to find important measures, that include, maximum free-flow q_{max}, free-flow speed V_f and critical density $k_{critical}$ which are used to differentiate between the free-flow

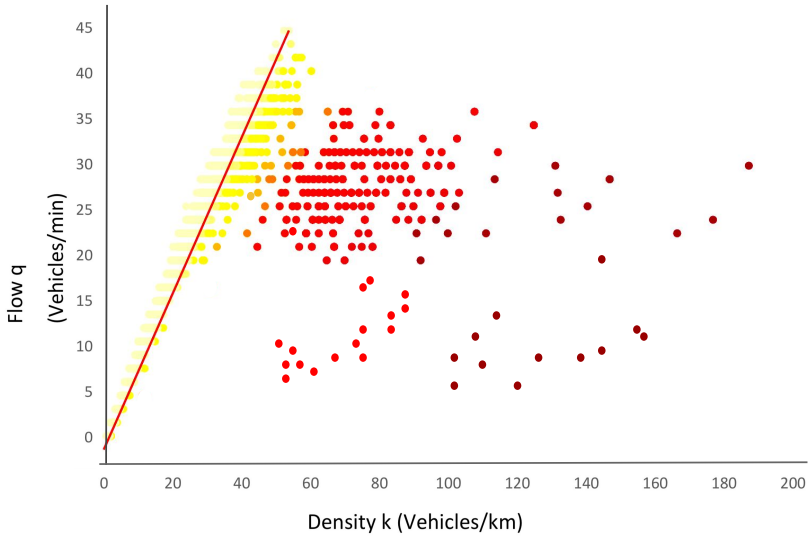


Figure 4.1 – Empirical fundamental traffic flow diagram

traffic and the congested traffic. Figure 4.1 represents the empirical fundamental traffic flow diagram generated using one of our real-life datasets. It shows the classical traffic flow curve behaviour. For estimating a congestion threshold, a line (shown in red) is drawn from the empirically computed q_{max} point to the origin. All the sensor readings on the left side of the line represent free-flow traffic and all the sensor readings on the right side represent congested traffic.

The magnitude of congestion can be determined depending on the distance from the congestion threshold line. The points farther from the line indicate traffic getting more and more congested. Different congestion levels are shown by different shades of red. Dark red color indicates high congestion. High congestion is the state in which traffic jams appear with jam density k_{jam}. Traffic density is an important measure to determine traffic jams on road. We use density in our work as an indicator of traffic jams.

4.3 Traffic Jam Detection

Here, we present an overview of our traffic jam detection and congestion reduction system. Next, we explain the graph representation of the dataset used in our work for computing various traffic metrics over the traffic streaming data and we explain how it is used for traffic jam detection. In the end, we present the streaming graph processing based algorithm to find connected traffic jams from the traffic stream.

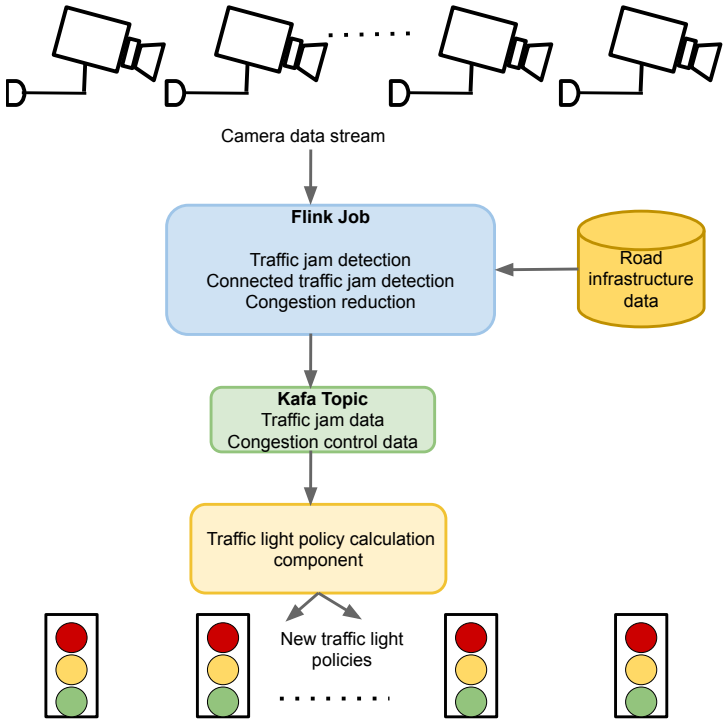


Figure 4.2 – Traffic jam detection and congestion control system

4.3.1 Traffic Jam Detection and Congestion Reduction System Overview

Our traffic jam detection and congestion reduction system, shown in Figure 4.2, is built using Apache Flink [36] and Kafka [95]. First, the input stream from various cameras placed on road intersections is fed to the system. A Flink job then 1) processes the camera data stream along with the road infrastructure information to compute traffic metrics, such as, average speed and density of vehicles; 2) detects traffic jams in the streams using the computed traffic metrics; 3) identifies the traffic jams that are connected; 4) detects the paths containing traffic jams in the traffic graph and their neighboring paths from which the traffic is incoming to the congested paths. These paths' information, i.e., paths with traffic jam and their neighboring paths, is written on the Kafka topics as output. This data helps in creating traffic light policies for congestion reduction in the congested region. We will explain the congestion reduction mechanism in detail in the next section.

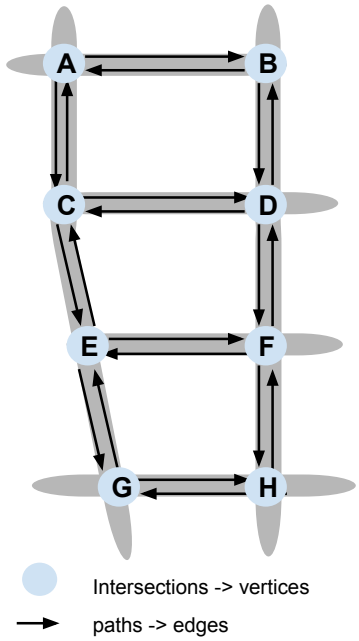
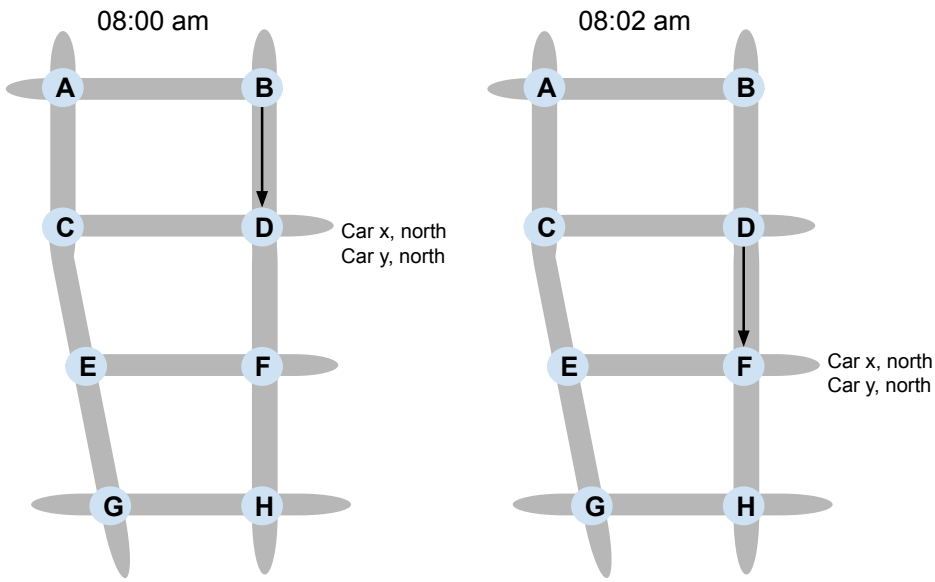


Figure 4.3 – Graph representation of traffic data

4.3.2 Graph Representation of Traffic Dataset

The traffic dataset used in our work is taken from a region of Shenzhen city in China containing traffic cameras deployed across various intersections of the roads in the city. Each intersection contains at least one camera in every direction, which detect the number of vehicles passing that particular part of the intersection. The camera data source is sending traffic data stream per second to the system, making it a high-intensity stream. To detect traffic jams and to find connected traffic jams, the input data is represented in the form of a graph. A graph is useful to represent relationships between various entities in a network. In our case, the relationship exists between various camera sensors that are deployed across the same intersection in the network or are connected with a path in the traffic network graph.

Figure 4.3 presents the road network of one of the regions from a city in China, which contains eight intersections, namely, A, B, C, D, E, F, G and H. Each intersection has at least one camera in every direction depending upon the number of directions from which the vehicles are passing these intersections. In order to create a graph of this network, we represented the camera sensors placed over intersections as vertices of the graph and possible paths that vehicles can take between these intersections as directed edges of the graph.



(a) vehicles detected at 08:00 am (b) vehicles detected at 08:02 am

Figure 4.4 – Vehicles detected across various intersections on the road

The traffic graph is created using the road infrastructure data information. The edges of the graph are then labelled dynamically using the camera data stream, which contains the sensor ID, the timestamp, the number of vehicles passing the sensor in the direction of the directed edge and their number plates.

4.3.3 Traffic Jam Detection

We detect traffic jams on the streams by computing the traffic density. The stream we receive, per second, from cameras contains the number of vehicles that pass the sensor, i.e., the traffic flow, and their number plates. First, we use this information to compute the average speed of the vehicles that are detected by the sensors, then we use the flow and speed values to compute the density of vehicles for detecting traffic jams using equation 3.1.

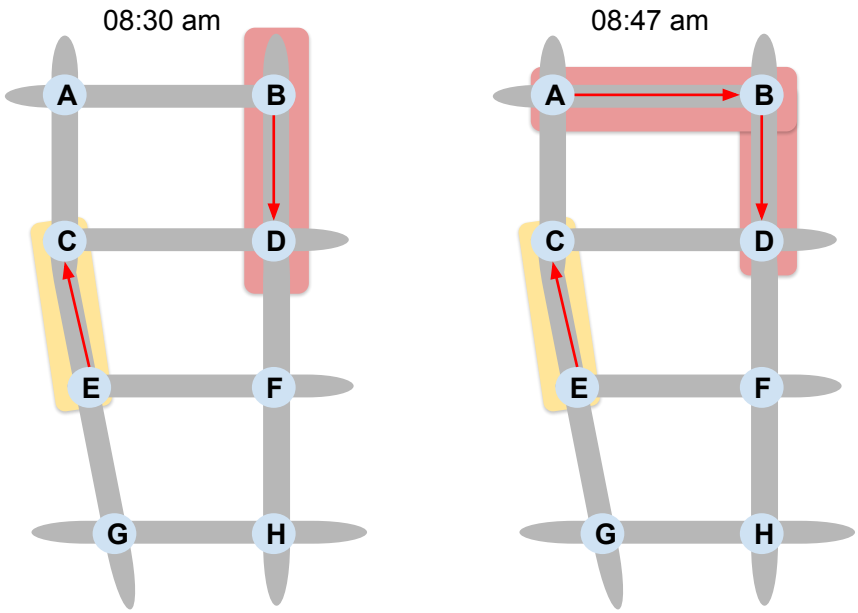
We explain our approach to compute traffic density with a simple scenario given in Figure 4.4, where we assume, for simplicity, that the distance between each intersection connected is 0.5 km. Figure 4.4a shows that two cars, with number plates Car x and Car y, were detected at intersection D coming from the

north direction. Later at 08:02 am in Figure 4.4b, Car x and Car y are detected at intersection F. In order to compute the average speed of cars at the intersections we keep one-hop records of the vehicles that cross the intersections. In the given scenario, Car x and Car y made one-hop from intersection D to F coming from the north direction in 2 min (08:00 am to 08:02 am). The average speed of vehicles crossing the sensor detecting vehicles from the north on intersection F at 08:02 am is computed using the travel time of Car x and Car y coming from D to F and the distance between D to F. In this case, the average speed is ≈ 15 km/h. This average speed value v is then used along with the aggregated traffic flow q value, that is aggregated per minute, to estimate the traffic density at the path from D to F using equation 3.1. Similarly, traffic density values are computed over the stream at various intersections using the one-hop speed of vehicles at a particular time interval. We only keep one-hop trip information for speed computation of vehicles, as soon as the speed is computed we replace this information with the next-hop data to keep the state in memory minimal. Based on our empirical results, the traffic density ≥ 140 veh/km [96] indicates a traffic jam on the road. We use this threshold to identify the paths in the graph containing traffic jams.

4.3.4 Connected Traffic Jams

Once we detect the paths containing traffic jams in the graph, we use a streaming graph-based algorithm to track the propagation of traffic jams across the network and to find the traffic jams that are connected. Figure 4.5a shows two paths in the graph, i.e, BD, and EC, labelled as congested (red edges). These paths indicate the presence of traffic jams across them at 08:30 am. Figure 4.5b shows another path AB labeled as congested at 08:47 am. This path is connected to the previous congested path BD, thus the traffic across them are part of the same traffic jam. We adapted the connected components algorithm [97] over streaming graphs to find the connected traffic jams in our graph of traffic streams and track their propagation in time.

Algorithm 2 is for the connected traffic jam detection on graph streams. A graph stream is created in the form of streaming edges [24], where each edge (x, y) has two-end vertices x and y. In our case, the end-vertices represent the intersection IDs and we create a weighted edge, (x, y, w), with the weight w equal to the traffic density value on the destination vertex. For example, for the path BD (in Figure 4.5), assuming it has a density 150 veh/km at 08:30 am at D, an edge representing this information will be created as (B,D,150). After creating these edges, they are first filtered based on our traffic density threshold, i.e., 140 veh/km. These filtered edges are denoted as congested edges in the algorithm. For each incoming edge in the stream, the end vertices x and y are assigned to a set j with $j.id$, where id indicates the traffic jam id and j represents a set containing vertices that belong to the traffic jam with the assigned id. The end vertices are assigned based on the given rules: 1)



(a) traffic jam at 08:30 am (b) traffic jam at 08:47 am

Figure 4.5 – Traffic jams detected across the network

If all of the sets, $j \in J$, does not contain x and y, then create a new set j with id that is minimum of $x.id$ and $y.id$. 2) If either of x and y and present in j, then add the other vertex to the same j. 3) If both x and y are present in the same j, do nothing. 4) If both x and y and present in different sets then merge the two sets and set id to the minimum value of both set ids.

In the case of the aforementioned scenario in Figure 4.5, our algorithm will result in giving two groups of traffic jams, i.e., group 1 (B, D, A) containing intersection IDs for paths AB and BD, and group 2 (E, C) containing intersection IDs for the path EC. Once we detected the connected traffic jams, they are used in the congestion reduction policies that we discuss next.

4.4 Congestion Reduction

In this section, we present our congestion reduction scheme. We first explain the identification of paths in the traffic network that are selected for deploying new traffic light policies and then explain the changes we make to the traffic lights along these paths in the road network.

Algorithm 2 Algorithm for detecting connected traffic jams

Input: Incoming stream of congested edges (x, y, w)
Output: Traffic Jam IDs for each vertex x and y

for *all edges* (x, y, w) *in set* E **do**
  for all sets $j \in J$ **do**
    if *Set* j *contains both* x *and* y **then**
      ▷ `x and y have been seen before`
      return
    end if
    else if x *and* y *are in different sets* **then**
    merge the two sets and set the id of the new set to minimum of both set ids
    end else if
    else
    if x is in a set, add y to the same set, and vice versa
      ▷ `Only one of the vertices has been seen before`
    end else
  end for
  create new set j with id `min(x.id, y.id)` and assign x and y to it
    ▷ `x and y vertices have not been assigned before`
end for
Return J

4.4.1 Selection of Paths for Deploying New Traffic Light Policies

For reducing the effect of traffic jams, once they are detected in the traffic network, we identify the links that are essential to reduce the overall travel time of the cars in the congested area. The aim of identifying these links is 1) to control the traffic that is moving towards the congested region to avoid further congestion in that region and, 2) to reduce the travel time of cars that are in the congested region. Figure 4.6 shows two traffic jams detected at 08:47 am. Path AB and BD are part of the first traffic jam and the EC is part of the second traffic jam. The traffic moving towards the first traffic jam is coming from path AC (highlighted in green) and the traffic moving towards the second traffic jam is coming from paths FE and GE (highlighted in green). In order to find these paths highlighted in green, we do one-hop backward propagation in the graph from the intersections that are part of traffic jams (highlighted in red and yellow) and get the ids of the sensors located along these backward propagation paths (highlighted in green). Policies for traffic lights placed on these backward propagation paths along with the paths that are in the congested region are then changed to reduce the overall effect the congestion.

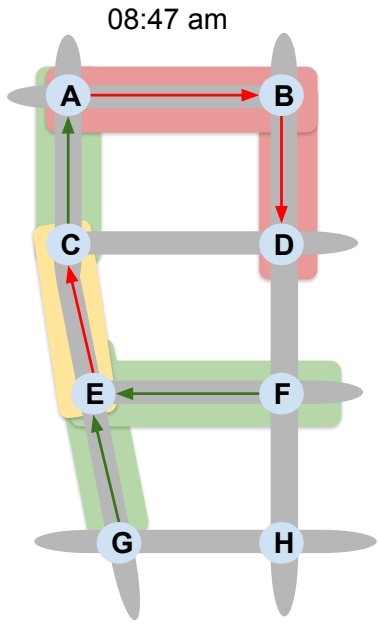


Figure 4.6 – Paths in traffic network containing vehicles moving towards the congested regions

4.4.2 Traffic Light Policies

The traffic light policies are built to control the red and green signal time of traffic lights. Typically the traffic light policies are fixed and do not react dynamically to the changes in traffic. In our work, we dynamically change the traffic light policies to control the green signal time of the traffic lights placed on roads under congestion, marked in red and yellow in Figure 4.6, and the roads containing traffic moving towards the congested area marked in green in Figure 4.6. We change the green signal time either by increasing it with bonus addition or by decreasing it with bonus removal. Adding a bonus gives more green time for the vehicles to pass the intersection. Alternately, subtracting a bonus gives less green time for the cars to pass the intersection. A bonus is added to the traffic lights green signal time in the congested region to allow more cars to pass the intersection to reduce the concentration of vehicles stuck in a traffic jam. A bonus is subtracted from the green signal time of traffic lights placed on roads containing traffic moving towards the congested region to allow fewer cars to move towards the congested regions. The bonus is computed using the given formula:

$$bonus = ((k_{node} - k_{jam}) * bonus\ factor)/100 \quad (4.1)$$

Here, $bonus\ factor$ here is 0.3 times the green time, where green time is the time during which the traffic signal is green. k_{node} is the traffic density value at the node in the graph which selected for bonus application, and k_{jam} is the density value threshold during traffic jam set to 140 veh/km based on empirical results and theoretically suggested traffic jam density threshold [96].

4.5 Experimental Evaluation

In this section, first, we explain the experimental setup and datasets used in our work to evaluate the performance of our proposed traffic jam detection and congestion control framework. Then, we present the experimental results of our work. The aim of our experiments is to measure, 1) the performance of our system in terms of reducing the effect of congestion on roads, and 2) the scalability of system in terms of handling high-intensity streams from a large number of camera detectors.

4.5.1 Experimental Setup

Datasets. Datasets used in our experiments consist of both real-life and synthetic datasets. Information about the datasets used in our experiments is given in Table 4.1. The first dataset, which we refer as Region 1, is a real-life dataset taken from a region of one metropolitan city, namely Shenzhen, in China. This dataset is collected from various traffic sensors deployed across intersections of the roads. The type of sensors from which data is collected is cameras. The cameras send per second information about the vehicles crossing the intersection. This per second stream consists of a timestamp, number of vehicles crossing, their number plates and the direction from which they arrive. Region 1 consists of 8 road intersections containing 28 number of traffic sensors, where each intersection has a camera in every direction. Region 1 dataset consists of a total of $\approx$ 2 million records that are collected over a period of 24 hrs. The records consist of vehicle detections sent per second by the sensors. The second dataset, which is bigger, is used to test the scalability of the system. It is a synthetically generated Grid dataset using SUMO simulator tool [98] based on the traffic behaviour of our real traffic datasets. The road network for this large dataset is a squared grid with 225 intersections. Each intersection is connected to its neighboring intersections with two lanes (one per direction) of 400 meters length. Grid consists of a total of 900 sensors generating per second streaming data. This makes more than 137 million data points in the data for a period of 12 hrs. The size of this dataset is around 50 GB.

Table 4.1 – Datasets with their attributes used in experiments

Attributes	Region 1	Grid
# intersections	8	225
# sensors	28	900
# records	2,419,200	137,721,600
size of storage	1GB	50GB

Experimental Environment. We performed our experiments on a physical on-premises machine. Its specs are `Intel (R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz`, Linux OS and 32 GB of RAM. We used Apache Flink v1.10.0 with a local cluster consisting of one job manager and one task manager consisting of 8 parallel task slots. Furthermore, we used Kafka v2.12. The traffic jam detection and congestion control framework is written in Java 8.

Metrics. We measure the performance of our system in terms of reducing the effect of congestion on road in Section 4.5.2. We compute the travel time of cars after applying changes to the traffic light polices once the traffic jam is detected. We compare this travel time to the base-line travel time of cars with default traffic light policies. Next, we measure the scalability of our framework, in Section 4.5.3, by computing the throughput, i.e, the number of records processed by the system per second with an increasing traffic stream, and the latency, i.e, the time to process one record by the system with an increasing traffic stream.

4.5.2 Congestion Reduction

In this section, we explore the effect of the bonuses that are applied to the traffic lights dynamically as a part of our congestion reduction policies after detecting congestion. We generated three different disruption scenarios in the simulation to capture the effect on travel times of cars. The scenarios were created by blocking two links in the graph of Region 1 for 10 min, 20 min and 25 min. Figure 4.7 is a snapshot taken from the SUMO simulator tool that contains two cars shown in red creating disruption on the road network.

In the disruption scenarios, we measure the average travel time of cars during their trips after applying the congestion reduction policies, denoted as TTR, and compare it to the average travel time of cars during the trips without the congestion reduction policies, denoted as TTC. The normal travel time of cars on the same trips with no disruption is denoted as TTN.

10 min disruption. Figure 4.8 shows the average trip times of vehicles with (TTR) and without (TTC) congestion reduction policies after a 10 min disruption is created. TTN is the average travel time without disruption. A traffic jam is detected after 08:30 am. Overall TTR is lower compared to TTC, resulting in fewer travel times

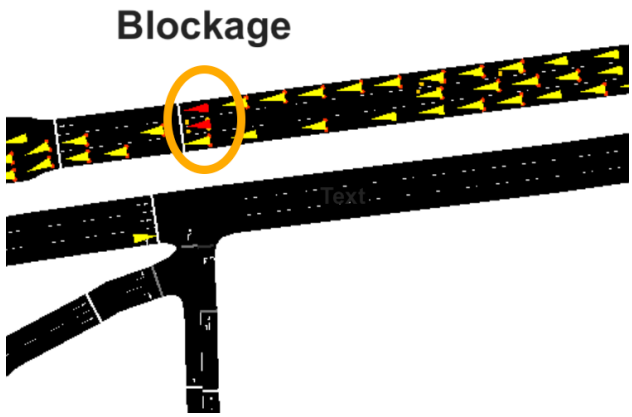


Figure 4.7 – Disruption created using two blocking vehicles shown in red

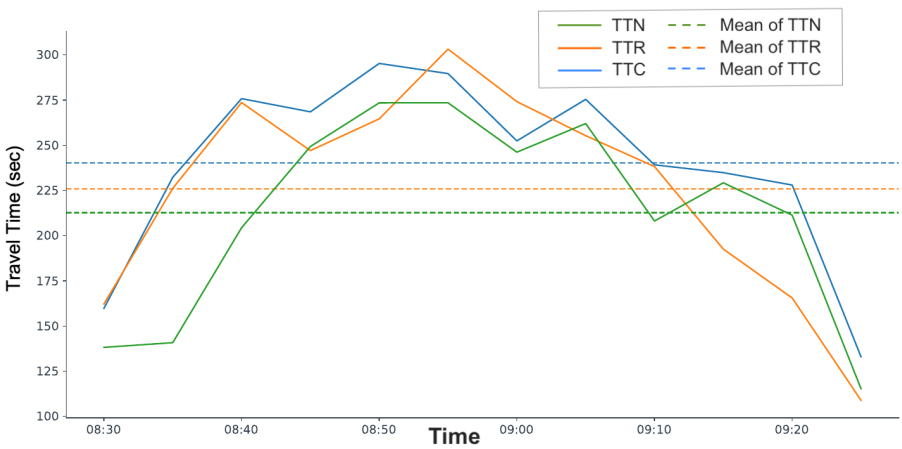


Figure 4.8 – Average travel time of vehicles with 10 min disruption

of all cars with the application of congestion reduction policies. During the traffic jam interval from 08:30 onwards till 09:10, the average travel time of all cars is reduced by $\approx$ 15%, at the best at 08:50. After the traffic jam interval, 09:10 onwards, the reduction in travel time is even more as TTR is lower compared to TTC. The average time reduced after 09:10 is $\approx$ 27%, at the best. Overall during disruption, vehicles have $\approx$ 8% less travel time after bonuses are applied to traffic lights during congestion reduction, compared to travel time with default traffic light policies.

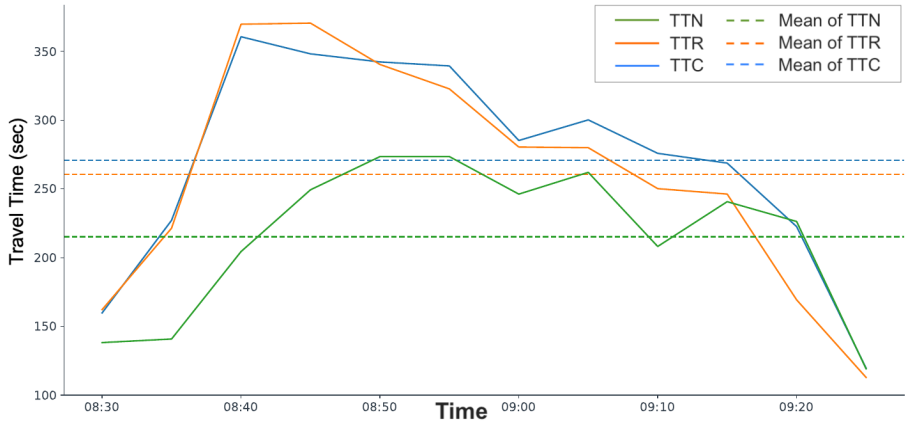


Figure 4.9 – Average travel time of vehicles with 20 min disruption

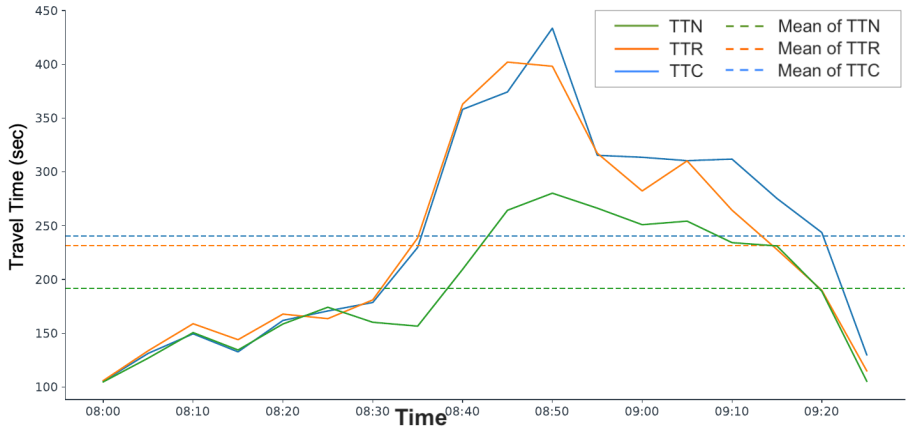


Figure 4.10 – Average travel time of vehicles with 25 min disruption

20 min disruption. Figure 4.9 shows the average trip times of vehicles with (TTR) and without (TTC) applying congestion reduction policies on traffic lights after a traffic jam is detected around 08:30 am for the 20 min disruption scenario. Overall cars take less travel time with congestion reduction policies since TTR is lower compared to TTC. During the traffic jam interval, i.e., from 08:30 onwards till 09:10, the average travel time of all cars is reduced by $\approx 15\%$, at the best at 09:10. After 09:10, TTR continues to be lower than TTC. The average travel time reduced after 09:10 is $\approx 18\%$, at the best. Overall all cars take 5% less travel time after applying congestion reduction policies compared to travel time with default policies.

25 min disruption. Figure 4.10 plots the average trip times of vehicles with (TTR) and without (TTC) applying congestion reduction policies to the traffic lights for the 25 min disruption. A traffic jam is detected after 08:30. Overall TTR is lower compared to TTC indicating fewer travel times of all cars with the application of congestion reduction policies. On average all cars take $\approx 5\%$ less travel time, and at the best $\approx 22\%$ less travel time, after congestion reduction policies are applied to the traffic lights, compared to travel time with default policies.

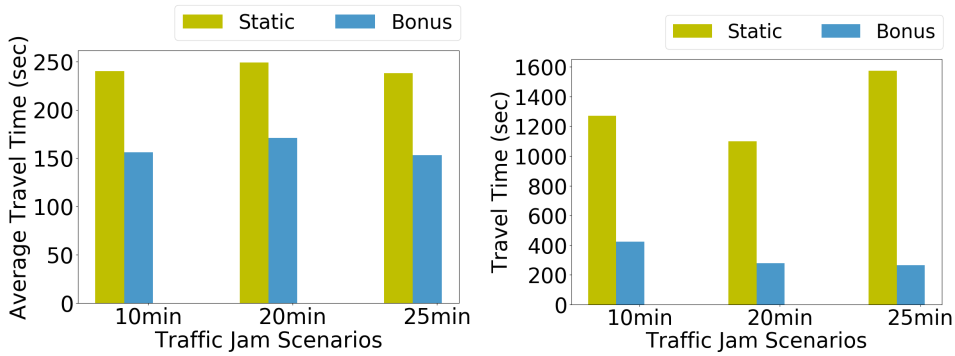
Comparison. We compare the congestion results for all three disruption scenarios by measuring the travel time of vehicles that were only in the congested region during the disruption scenario, i.e., the region where the traffic jam is detected. We take into account the vehicles for which the travel time was reduced after applying bonuses to the traffic lights. Figure 4.11a plots average travel times of the aforementioned vehicles with bonuses applied to traffic lights and the average travel times of these vehicles with the default static traffic light plans. For 10 min disruption scenario, the average travel time of 4242 vehicles is $\approx 35\%$ less with bonuses applied to traffic lights compared to the default static policies. Similarly, for 20 min disruption, the average travel time of 4163 vehicles is $\approx 31\%$ less with bonuses. Lastly, for 25 min disruptions, the average travel time of 6663 vehicles is $\approx$ 36% less with bonuses applied to traffic lights compared to static plans.

Figure 4.11b shows the travel time of cars during the three aforementioned disruption scenarios that had the most travel time gain. Travel time gain is the difference in trip time during static traffic light policies from the trip time during bonus based policies. Our results for the best single vehicle trips show that the vehicle takes almost $3\times$ less travel time for 10 min, $4\times$ less travel time for 20 min, and more than $6\times$ less travel time for 25 min disruption scenario when bonuses are applied to traffic lights compared to the default traffic light policies.

4.5.3 Scalability

We now evaluate the scalability of our traffic jam detection and congestion reduction system by measuring the throughput and latency. The input dataset used for the scalability test is the Grid dataset comprising of total 137,721,600 records. In order to process the dataset in a distributed manner, we used a Taskmanager in Flink with 8 parallel task slots. It took a total of ≈ 40 min to process the complete dataset. We measured the throughput and latency of our system during the complete processing job of the dataset.

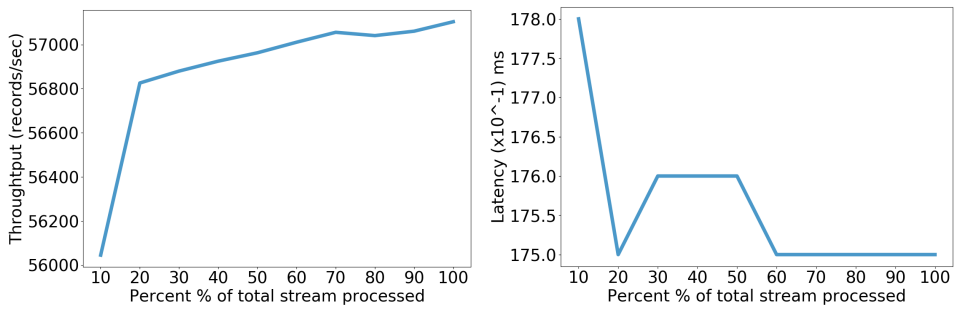
Throughput. Figure 4.12a shows the throughput of our system on the y-axis in terms of records processed per second and the x-axis shows the percentage of stream processed from the total dataset. Throughput of the system slightly increases with the increase in processing of the data stream. Throughput curve starts with around 56000 records/sec when 10% data is processed and goes up till 57056 records/sec at



(a) Average travel time of cars in congested region for different disruption scenarios

(b) Maximum travel time gain in the congested region for different disruption scenarios

Figure 4.11 – Comparison of travel time gains for 10 min, 20 min and 30 min disruption



(a) Throughput

(b) Latency

Figure 4.12 – Throughput and latency of the system

70% processed stream. Finally, the throughput is the highest, i.e., 57104 records/sec, when the complete stream is processed. The throughput curve shows that our system is capable of achieving high throughput during the processing of the stream.

Latency. Figure 4.12b shows the latency of our system on the y-axis in terms of milliseconds and the x-axis shows the percentage of stream processed from the total dataset. Our plot depicts that latency of the system, in general, decreases with the

increase in the percentage of the processed stream. Latency slightly increases at 30% processing of data, then goes low again at 60% processing of data. The throughput curve shows that our system has a very low latency during the processing of a big data stream.

4.5.4 Findings.

Our results indicate that 1) applying dynamic traffic light policies in the congested regions and its neighboring regions yields less travel time of cars. The average travel time of all cars is reduced at the best by 27% in 10 min, 18% in 20 min and 22% in 25 min disruption scenario and, 2) our traffic jam detection and congestion reduction system gives high throughput and low latency by keeping minimal state in memory yielding high-speed processing of large scale streaming data. From these results, we can deduce that considering the real-time requirement of traffic optimization, using our proposed framework, a simple consumer machine would allow us to monitor and help mitigating congestion in large urban areas.

4.6 Related Work

Several interesting research has been done on traffic monitoring, more specifically for congestion detection. One of the most popular and widely used congestion detection systems is Google Maps [99]. It uses probe vehicles and data collected from cellphones with GPS to monitor traffic. However, this method is based on massive data collection and raises privacy concerns according to the General Data Protection Regulation (GDPR) laws [100].

Existing work done on congestion detection by Anbaroglu et al. [37, 38] detects congestion using link journey times of cars. Soylemezgiller et. al [39] proposes a road pricing model for reducing congestion on the road. These approaches make use of historic data that includes past link journey times and other past statistics of the road at a specific hour of the day. In a pure streaming system, this historic data is not available, making these approaches not suitable to be implemented for online processing.

Other techniques include vision-based congestion detection [101, 102, 103, 104]. These techniques require computationally expensive pre-processing steps that include feature extraction, background subtraction etc., thus making them unsuitable for real-time congestion detection system with low latency requirements over large scale streaming data. Recently various studies [105, 106, 107] have been done on using vehicular ad hoc networks (VANETs) for local congestion information propagation in real-time. In this approach, a vehicle collects its surrounding information about speed, position etc, and sends messages to surrounding vehicles. There are several problems with VANETs based techniques that still need to be addressed. Firstly, information disseminating on urban roads is challenging due to

their complex topology. Secondly, cooperation among vehicles is not very effective making congestion detection imprecise and challenging in real-time. Moreover, we use fixed-point dataset in our work, VANETs based approaches cannot work on our dataset. Besides this, several neural network-based techniques [108, 109, 110, 25, 1] are being explored to detect and predict congestion. Since they all require historic data for training, we are unable to use them in our work for building a stream processing based traffic control system.

Another activity related to congestion detection is incident detection. An incident happens due to congestion on unusual times. INGRID [111] and RAID [112], to name a few, are systems developed for incident detection. Both these systems make use of inductive loop detectors. In order to monitor traffic with induction loops, multiple of them need to be installed on the roads for accuracy of traffic metrics measured. Furthermore, they are expensive to install, which makes them not economic to use in large cities. Our congestion detection mechanism works only with a camera installed on the intersections of the road. Which are less in number and cheaper compared to induction loops.

4.7 Acknowledgments

I would like to thank the Lamport team; Stefano Bortoli, Paolo Sottovia, Mohamad Al Hajj Hassan, Daniele Foroni, Cristian Axenie and Alexander Wieder from the Huawei Munich Research Centre, and Stella Maropaki from NTNU, Norway, for their valuable feedback and guidance. Also, I would like to thank my masters student Thorsteinn Thorri Sigurdsson who helped us developing the initial congestion detection mechanism using graphs. Thorsteinn was co-supervised by Ahmad Al-Shishtawy and Vladimir Vlassov who provided useful feed back and guidance throughout the work.

4.8 Summary

In this work, we address the challenge of *CH2 Large Scale Spatial Time Series Analysis*, mentioned in Chapter 1. In particular, we tackle the problem of large scale time series analysis on a practical use-case of traffic congestion detection and mitigation on real-life data collected from a region in one of the largest metropolitan cities in China. We proposed an end-to-end framework built on top of a modern stream processing system, i.e., Apache Flink. Flink is used because of its high throughput and low-latency guarantees. Our framework comprises of two mechanisms: 1) an online traffic jam detection mechanism for detecting jams on streaming data collected from traffic sensors, and 2) a congestion reduction mechanism based on streaming graph analytics for reducing the effect of congestion in the congested area. In our proposed congestion reduction approach, we identify correlated traffic

jams and essential parts in the road network on which new traffic light policies are deployed for congestion control. With the proposed framework, we are not only able to detect traffic jams in real-time, but we also apply dynamic traffic light policies that yield less travel time for vehicles in the congested region. Moreover, our system is capable of processing high-intensity and large scale traffic streams with low latency and high throughput because the memory state is kept minimum.

Our work, which is based on a real-life case study, is effective in giving the desired performance and scalability in traffic congestion detection and mitigation. This work can help in monitoring and reducing congestion considering the real-time requirement of traffic optimization in a large urban area.

For future work, we consider investigating more into dynamic traffic light policy adaption. The policy adaption and removal strategies can be deeply investigated to bring more effective results.

CHAPTER

5

Streaming Graph Partitioning

"This paper needs some love and care and then we go to Rio"

— Paris Carbone, when we were working on our survey paper for VLDB.

5.1 Introduction

Graph partitioning, the process of dividing a graph into a predefined number of subgraphs, is essential for graph analysis using distributed algorithms. Distributed graph processing has been widely adopted in recent years and enables knowledge extraction from large and medium-scale graph-structured datasets using commodity clusters [50, 19, 20, 51]. In such settings, each cluster node operates on one partition in parallel and communicates with other nodes through message-passing. Hence, partitioning quality directly affects communication and computation costs and is crucial for graph application performance [19, 2].

The problem of graph partitioning has been thoroughly studied and several methods have been proposed in the past few decades, each with a particular graph type, ingestion model or application objective in mind. Among existing methods, we study *streaming* graph partitioning algorithms. As opposed to *offline* methods, which first load the complete graph in memory and then divide it into partitions, streaming graph partitioning operates *online*, while ingesting the graph data as a stream [2].

We examine two practical use-cases of streaming graph partitioning. First, in the context of the *load-compute-store* computational model (e.g., MapReduce [113], Spark [114, 20], Giraph [51]), partitioning can be performed in a streaming manner during the *load* phase, by treating the bounded input graph dataset as a stream of vertices or edges. Second, it is appropriate for distributed streaming and semi-streaming algorithms [62, 115, 116, 117] that compute graph summaries and perform

aggregations on possibly unbounded edge streams, and systems supporting native graph-as-a-stream computations [28, 29, 30, 31, 10].

Partitioning methods vary significantly in terms of their heuristics, assumptions, and respective performance, thus making it difficult for developers to compare them and assess their characteristics. Choosing the right technique for the computational problem at hand is non-trivial, especially because each method adopts a limited set of application objectives and constraints. As in offline graph partitioning, streaming partitioning typically defines two main objectives: *load balancing* and *minimum cuts* (vertex or edge). These correspond to aiming for fair load distribution and minimized communication overhead, respectively. Optimizing for both objectives, also known as the *balanced graph partitioning* problem, is an NP-hard problem [54].

Past studies [45, 44] have focused on offline graph partitioning techniques or heuristics used for streaming graph partitioning [2]. However, in the context of the stream ingestion model, the question of identifying factors that influence performance and quantifying their effects is still open. We specifically examine sensitivity to stream ingestion order, the number of partitions, suitability for unbounded processing, and cost amortization of applications, including bulk synchronous iterative processing [50] and stream or semi-streaming graph approximations [115, 62].

The necessary preliminaries for this chapter are presented in Chapter 2. In this chapter, we classify algorithms with regards to their data model, strategy, constraints, complexity, state requirements, and objectives. To provide an unbiased performance comparison, we implement all studied methods on top of a common evaluation framework based on Apache Flink [118, 30], a distributed stream processing engine. Finally, we use bulk synchronous and single-pass graph streaming algorithms to evaluate distributed graph application performance in terms of partitioning cost amortization.

5.2 Online Partitioning Methods

Streaming graph partitioning algorithms are quite diverse in their objectives, assumptions, and runtime complexities. We summarize eight partitioning algorithms that can be used in the streaming model and categorize them based on the following criteria: 1) *data model*, 2) *partitioning strategy*, 3) possible *constraints* regarding input boundness or a priori knowledge etc., 4) *computational and space complexities*, 5) *state* requirements while partitioning, and 6) *optimization objectives*. Table 5.1 summarizes the algorithms across all criteria. In Section 5.2.4 we highlight the main findings.

5.2.1 Model-Agnostic Methods

Data-model agnostic partitioning algorithms can be employed in both vertex and edge-centric models. Hash partitioning is probably the most representative and widely-used method in this category.

Table 5.1 – Features and characteristics of the chosen streaming partitioning methods for this study.

Algorithm	**Data model**	**Strategy**	**Constraints**	**Space**	**Time**	**State**	**Objective**
Hash	Agnostic	Hash	None	None	*O(n)* / *O(m)*	None	Load balance
LDG	Vertex stream	Neighbors	Bounded stream	*O(n)*	*O(kn+m)*	Vertices, partition assignment	Load balance, edge-cuts
Fennel	Vertex stream	Neighbors / non-neighbors	Bounded stream	*O(n)*	*O(kn+m)*	Vertices, partition assignment	Load balance, edge-cuts
Greedy	Edge stream	End-vertices	None	*O(n)*	*O(km+n)*	Vertices, partition assignment	Load balance, vertex-cuts
HDRF	Edge stream	Degree	None	*O(n)*	*O(km+n)*	Vertices, degree, partition assignment	Load balance, vertex-cuts
DBH	Edge stream	Degree and hash	None	*O(n)*	*O(m+n)*	Vertices, degree, partition assignment	Load balance, vertex-cuts
Grid	Edge stream	Hash	Perfect square partitions	*O(n+k)*	*O(m+n)*	Vertices, partition assignment	Load balance, vertex-cuts
PDS	Edge stream	Hash	$p^2 + p + 1 = k$	*O(n+k)*	*O(m+n)*	Vertices, partition assignment	Load balance, vertex-cuts

5.2.1.1 Hash Partitioning

The idea of using a consistent hashing function to map elements with distinct keys to different partitions is widespread outside the domain of graph processing (e.g., for load balancing content in distributed key-value stores [119] and managed stream state [118]). In the context of vertex partitioning, a consistent hashing function can be used to assign vertices with unique identifiers to a physical partition index $V \rightarrow \mathbb{N}$ uniformly at random. Similarly, in the case of an edge data model hashing maps a set of edges to partitions $E \rightarrow \mathbb{N}$. For brevity, if we assume a vertex-centric model, hash-based partitioning can be defined as the mapping function $f(v) = hash(v) \bmod (k)$.

Discussion: Hash partitioning is simple and does not require any a priori knowledge of the graph structure (only the number of partitions needs to be known), making it generally applicable to unbounded streams. Hashing is also stateless since it requires no history synopsis during partitioning, thus it can be trivially parallelized and be used to partition large-scale graphs.

5.2.2 Vertex Partitioning Methods

In the category of vertex partitioning algorithms, we analyze Linear Deterministic Greedy [120, 2] and Fennel [121, 23] as good representatives of partitioning mechanisms that can be applied online on a stream of vertices.

5.2.2.1 Linear Deterministic Greedy (LDG)

Linear Deterministic Greedy partitioning (LDG) tries to place neighboring vertices to the same partition, in order to yield fewer edge-cuts [2]. It uses a greedy heuristic that assigns a vertex to the partition containing most of its neighbors while respecting certain capacity constraints.

More specifically, given a range of partitions in $[1, k] \in \mathbb{N}$, let P_i represent the set of vertices placed in partition $i \in \{1, ..., k\}$. For $N(v)$, the known set of neighbors of v, the LDG heuristic is given by $f(v)$ in the following Equation:

$$\begin{aligned} f(v) &= arg\,max_{i \in [1,k]} \{g(v, P_i)\} \\ g(v, P_i) &= |P_i \cap N(v)| w(i) \\ w(i) &= 1 - \frac{|P_i|}{C} \end{aligned} \tag{5.1}$$

LDG selects the partition that maximizes $|P_i \cap N(v)|$, the number of neighbors already assigned to a partition while enforcing the capacity constraint $C = \frac{n}{k}$.

The algorithm is shown in Pseudocode 1. The heuristic is continuously applied until the load of a partition reaches the threshold $g(v, P_i) < g(v, P_j), j \in \{1, ..., k\}$

Pseudocode 1 LDG

Input: v, $N(v)$, k
Output: partition ID

```
1: procedure PARTITION(v, N(v), k)
2:     for all partitions i = 1 to k do
3:         P_i ∩ N(v)                          ▷ neighbors in partition i
4:         w(i) = 1 − |P_i| / C                ▷ load penalty
5:         g(v, P_i) = |P_i ∩ N(v)| w(i)       ▷ partition scoring
       end for
6:     for all partitions i = 1 to k do
7:         ind = arg max_i {g(v, P_i)}
       end for
8:     Return ind
```

and $j \neq i$. The load penalty enforces load balancing to avoid the extreme case where all vertices end up in the same partition.

Discussion: LDG requires the number of vertices n to be known a priori for calculating the capacity constraint C. Hence, it is generally unsuitable for unbounded processing. The algorithm requires keeping track of all partitioning decisions made so far, saved as the partitioning assignment state, which is accessed for every vertex in the input stream.

5.2.2.2 Fennel

Fennel [23] is a partitioning strategy whose heuristic combines locality-centric measures (low edge-cut) [2] with balancing goals [122]. Fennel's core idea is to interpolate between maximizing the co-location of neighbouring vertices and minimizing that of non-neighbours. Pseudocode 2 presents the Fennel logic in more detail. As in LDG, Fennel computes the number of neighbors present in each partition for every input vertex. In addition, the load limit per partition which sets a threshold for the maximum number of assigned vertices. The score $\delta g(v, P_i)$ is computed according to Equation 5.2 for each partition whose load is below the threshold and the input vertex is assigned to the partition with the maximum score.

$$\begin{aligned} f(v) &= \arg\max_{i \in [1,k]} \{\delta g(v, P_i)\} \\ \delta g(v, P_i) &= |P_i \cap N(v)| - \alpha\gamma |P_i|^{\gamma-1} \end{aligned} \tag{5.2}$$

Here, $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$ and the load limit $= \nu\frac{n}{k}$. The parameters α, γ, and ν are tunable and control the weights associated with maximizing the number of neighbors and minimizing the number of non-neighbors for the input vertex during partition-

ing. In our experiments (Section 5.5) we picked the values used in the original evaluation [23], $\gamma = 1.5, \nu = 1.1$.

Pseudocode 2 Fennel

Input: v, $N(v)$, k
Output: partition ID

procedure *PARTITION(v, N(v), κ)*
  $load\ limit = \nu\frac{n}{k}$ ▷ load limit
  for *all partitions i = 1 to k* **do**
    if $|P_i| <$ *load limit* **then**
      $P_i \cap N(v)$ ▷ neighbors in partition i
      $\delta g(v, P_i) = |P_i \cap N(v)| - \alpha\gamma|P_i|^{\gamma-1}$
    end if
  end for
  for *all partitions i = 1 to k* **do**
    $ind = arg\ max_i\{\delta g(v, P_i)\}$
  end for
  Return *ind*

Discussion: Similar to LDG, Fennel requires the parameters n and m to be known a priori, and maintaining a persistent state of the assigned partitions during execution, that makes it unsuitable for partitioning unbounded streams.

5.2.3 Edge Partitioning Methods

Many partitioning mechanisms operate on the edge-centric model. We select the following techniques that can operate online on an edge stream: Greedy [19], HDRF [21], DBH [22], and Grid [123], which are presented next.

5.2.3.1 Greedy

Greedy is a rule-based partitioning mechanism introduced in PowerGraph [19]. It aims to minimize vertex-cuts while also assigning balanced load across partitions. Let $S(v_i)$ denote the set of partitions containing the vertex v_i. Pseudocode 3 presents the rules employed by the Greedy algorithm, where the $leastLoad(S(v))$ method returns the least loaded partition ID form the set $S(v)$.

For each edge in the input stream, Greedy examines the participation of the endpoint vertices to existing partitions by applying the following rules: 1) Rule 1: If both endpoint vertices have been previously assigned in any common partition pick the least loaded common partition. 2) Rule 2: If both endpoint vertices have been previously assigned in different partitions pick the least loaded from the union of all assigned partitions. 3) Rule 3: In case either vertex has been previously assigned, pick the least loaded partition from the assigned partitions of that vertex. 4) Rule

Pseudocode 3 Greedy

Input: v_1, v_2, k
Output: partition ID

```
1:  procedure PARTITION(v1, v2, k)
2:      if S(v1) ∩ S(v2) ≠ ∅ then
3:          partitionID = leastLoad(S(v1) ∩ S(v2))    ▷ leastLoad() returns
    least loaded partition ID
        end if
4:      if S(v1) ∩ S(v2) = ∅ && S(v1) ∪ S(v2) ≠ ∅ then
5:          partitionID = leastLoad(S(v1) ∪ S(v2))
        end if
6:      if S(v1) = ∅ && S(v2) ≠ ∅ then
7:          partitionID = leastLoad(S(v2))
        end if
8:      if S(v1) ≠ ∅ && S(v2) = ∅ then
9:          partitionID = leastLoad(S(v1))
        end if
10:     if S(v1) = ∅ && S(v2) = ∅ then
11:         partitionID = leastLoad(k)
        end if
12:     Return partition ID
```

4: If none of the vertices has been previously assigned, then pick the least loaded partition overall.

Discussion: Greedy does not require any knowledge of graph properties before processing the stream. Therefore, it can potentially process an unbounded edge stream. However, it requires maintaining the current partition assignment as a synopsis, which, in case of an unbounded stream would also grow without bound.

5.2.3.2 HDRF

HDRF [21] is particularly tailored for power-law graphs. It is based on PowerGraph's heuristic [19], which targets workloads with highly skewed graphs. The key idea that since power-law graphs have few high degree nodes and many low degree nodes, it is beneficial to prioritize cutting the high degree nodes to radically reduce the number of vertex-cuts. Pseudocode 4 summarizes the logic of HDRF.

In more detail, for an input edge $e = (v_1, v_2)$, the partial degrees of its endpoint vertices are recorded as δ_1 and δ_2 . These values are then normalized using:

$$\theta(v_1) = \frac{\delta(v_1)}{\delta(v_1) + \delta(v_2)} = 1 - \theta(v_2) \tag{5.3}$$

Pseudocode 4 HDRF

Input: v_1, v_2, k
Output: partition ID

procedure *PARTITION*(v_1, v_2, k)
  $\delta_1 = \text{getDegree}(v_1)$
  $\delta_2 = \text{getDegree}(v_2)$ ▷ `getting partial degree values`
  $\theta(v_1) = \frac{\delta(v_1)}{\delta(v_1)+\delta(v_2)} = 1 - \theta(v_2)$ ▷ `normalizing the degree values`
  for *all partitions i = 1 to k* **do**
    $C_{BAL}^{HDRF}(i) = \lambda \times \frac{maxsize - |i|}{\epsilon + maxsize - minsize}$
    $C_{REP}^{HDRF}(v_1, v_2, i) = g(v_1, i) + g(v_2, i)$
    $C^{HDRF}(v_1, v_2, i) = C_{REP}^{HDRF}(v_1, v_2, i) + C_{BAL}^{HDRF}(i)$
  end for
  for *all partitions i = 1 to k* **do**
    $ind = \arg\max_i \{C^{HDRF}(v_1, v_2, i)\}$
  end for
  Return *ind* ▷ `returning id of the partition`
procedure *G(V, I)*
  if *partition* $i \in S(v)$ **then**
    Return $1 + (1 - \theta(v))$
    else
    Return 0
  end if else

HDRF works using Equation 5.4. Each edge is assigned to the partition i with highest value of $C^{HDRF}(v_1, v_2, i)$.

$$C^{HDRF}(v_1, v_2, i) = C_{REP}^{HDRF}(v_1, v_2, i) + C_{BAL}^{HDRF}(i) \tag{5.4}$$

$$C_{REP}^{HDRF}(v_1, v_2, i) = g(v_1, i) + g(v_2, i) \tag{5.5}$$

$$g(v, i) = \begin{cases} 1 + (1 - \theta(v)) & \text{if } i \in S(v) \\ 0 & \text{otherwise} \end{cases}$$

$$C_{BAL}^{HDRF}(i) = \lambda \times \frac{maxsize - |i|}{\epsilon + maxsize - minsize} \tag{5.6}$$

Here, $maxsize$ is the size of partition with maximum load, $minsize$ is the size of partition with minimum load, $S(v)$ here is set of partitions containing vertex v, ϵ is a constant value, and λ controls load imbalance [21]. When $\lambda \leq 1$, the algorithm behaves similarly to Greedy. When the input stream is ordered in breadth-first or depth-first search order, each incoming edge is placed in the partition containing most of its endpoint vertices' neighbors. As a result, the algorithm can yield imbalanced partitions. Setting $\lambda > 1$ solves this issue by accommodating for load balance. If λ approaches ∞, then the algorithm behaves like random hash

partitioning. In our experiments (Section 5.5), we set the value of $\lambda = 1$ to optimize for minimum vertex-cuts.

Discussion: Similar to Greedy, HDRF does not require any graph parameters to be computed before partitioning, and thus, it can potentially process an unbounded edge stream. On the other hand, it requires maintaining the state of partitions for computing the load and $S(v)$. HDRF uses degree information for making partitioning decisions, however, instead of pre-computing degrees offline before partitioning, it can maintain partial degree information and update it while processing the input stream.

5.2.3.3 DBH

Degree Based Hashing (DBH) [22] is quite similar to HDRF, because it prioritizes cutting those vertices that have the highest degree. However, unlike HDRF, DBH employs hashing for partitioning. Pseudocode 5 presents the algorithm in more detail. For an input edge e, DBH computes the partial degree of its endpoint vertices v_1 and v_2, δ_1 and δ_2. After that, e is assigned to the partition ID computed as the hash of the vertex with the lowest degree.

Pseudocode 5 DBH

Input: v_1, v_2, k
Output: partition ID

```
1: procedure PARTITION(v1, v2, k)
2:     δ1 = getDegree(v1)
3:     δ2 = getDegree(v2)
4:     if δ1 < δ2 then
5:         Return Hash(v1)mod(k)
6:         else
7:         Return Hash(v2)mod(k)
       end if else
```

Discussion: DBH algorithm keeps partial degree information of vertices as a state synopsis. Since it uses hashing, it can compute the current partitioning assignment on-the-fly, thus reducing the state requirements. DBH can potentially process unbounded streams because no global graph properties are required prior to partitioning.

5.2.3.4 Grid and PDS

The Grid [123] algorithm also uses hashing for partitioning. Prior to employing hashing, all partition IDs are arranged in a square matrix, termed the *Grid*. For

each incoming edge $e = (v_1, v_2)$, a constrained set of partitions $S(v)$ for each end vertex v is formed by taking all the partitions in the row and column of the partition where v hashes to in the grid. The edge is assigned to the least loaded partition in the set $S(v_1) \cap S(v_2)$. The main limitation of Grid is that it limits the possible number of partitions to logarithmic degrees for constructing a square matrix ($rows \times columns = N$), where N is the total number of partitions. An alternative algorithm to Grid is PDS [123] that computes the set of partitions using Perfect Difference Sets. It requires $(p^2 + p + 1)$ number of partitions, where p is a prime number.

Discussion: Both Grid and PDS place a constraint on the number of partitions, but, they require no pre-computation on the input graph and they can both potentially sustain unbounded streams. With regards to the state, both algorithms need to maintain the partitioning assignment to compute the least loaded partitions.

5.2.4 Comparison Summary

The streaming graph partitioning algorithms presented so far in this section have various objectives and characteristics. We summarize their main features and design choices and point the reader to their categorization in Table 5.1.

Strategy: Except for hash-based partitioning which is the only stateless algorithm we consider, the strategy used by a partitioning method generally also defines the state it needs to maintain during partitioning. The current partitioning assignment and degree information are used across algorithms to reduce cuts. Vertex partitioning algorithms check the partitioning assignment to compute the number of existing neighbors and non-neighbors of a vertex, while few edge partitioning algorithms also use degree-based strategies.

Constraints: All stateful vertex partitioning algorithms considered are designed for partitioning bounded streams of vertices, while edge partitioning algorithms are more flexible in general. Greedy, HDRF, and DBH have no constraints whatsoever and could potentially process unbounded edge streams. However, this is challenging in practice, due to state requirements discussed next.

State: The state kept and accessed for decision making by the partitioning algorithms affects their computational and space complexities, as well as their applicability to processing unbounded graph streams. All stateful algorithms considered require a way to inspect the current partition assignment or degree, whether for load balancing or for minimizing cuts. As a result, they need to maintain a synopsis that can be queried for vertex or edge membership and current partition size at the very least. Such synopses can grow beyond memory limits for unbounded streams, and also complicate distributed implementations, as they need to be consistent and accessible by all parallel instances of the partitioner.

5.3 Applications

We evaluate the partitioning methods with applications that operate on graph snapshots (batch) and graph streams. We have chosen bulk iterative algorithms and single-pass streaming summaries which we briefly describe next.

5.3.1 Iterative Applications

Connected Components: The connected components algorithm identifies subgraphs within which every vertex is reachable from every other vertex [124]. In the iterative, vertex-centric implementation of this algorithm [125, 126], each vertex is initially assigned a value equal to its own ID. Then, in every iteration, the vertex gathers values from its neighbors and picks the lowest value, which it then scatters back to its neighbors. When the algorithm converges, vertices with the same ID belong to the same component.

PageRank: The PageRank algorithm is an iterative vertex ranking algorithm that assigns weights to vertices based on their importance and their connectivity to other well ranked vertices [127]. The algorithm assigns an initial uniform rank to all vertices. Then, in each iteration, a vertex updates its rank by summing up the partial ranks from its incoming neighbor vertices. The new rank is then evenly distributed across outgoing edges to outgoing neighbors. The algorithm converges when the difference between the vertex rank form the current iteration and the rank in the previous iteration is less than a specified threshold.

Single Source Shortest Paths: The SSSP algorithm finds the shortest path between the source vertex and all connected vertices [128]. It initially assigns a zero value to the source vertex and ∞ to all other vertices. Then, each vertex updates its distance or path length to the source, until it does not change anymore across two consecutive iterations.

5.3.2 Single-Pass Applications

Bipartiteness: The bipartiteness algorithm continuously checks whether a graph stream forms a bipartite graph [129]. As long as the vertices seen so far can be divided into two groups such that there are no edges within those groups, then the graph is bipartite. The single-pass implementation maintains the current groups as state and assigns them a positive or negative sign. Then, the algorithm tries to place the vertices of each incoming edge to the existing groups by maintaining or flipping the signs. The distributed implementation maintains a partial state per processing instance and periodically merges the states into a combined state reflecting the history of the graph stream. The number of operations during the

merge of partial state corresponds to cross-partition communication and thus we expect good partitioning algorithms to result into smaller partial states and more efficient merging.

Connected Components: The single-pass Connected Components [97] algorithm, also known as union-find, operates online over an edge stream. The algorithm maintains a disjoint set data structure to keep track of components. For each incoming edge, it checks whether the endpoint vertices belong to an existing component and merges components accordingly if the endpoints already exist in disjoint components. The distributed implementation maintains a partial disjoint set per partition and periodically merges states similarly to the bipartiteness check implementation.

5.4 Evaluation Methodology

We design our experimental analysis by separating concerns among the system runtime, partitioning algorithms, and application on top. With that goal, we implement a comparison framework that can isolate partitioning costs and application performance under different iterative and pure streaming workloads. More concretely, in our experiments, we seek answers to the following questions:

Q1 What are the benefits, if any, of using more complex, data-centric partitioning methods compared to a generic hash-based strategy?

Q2 What is the partitioning overhead for an application using each partitioning algorithm?

Q3 How does the partitioning quality affect the application performance?

Next, we describe the implementation, datasets, parameters, metrics and experimental setup we use.

5.4.1 Implementation

Apache Flink [30, 118] is a streaming-first distributed analytics platform which executes complex applications by generating a DAG of logical operators and connecting the data streams to it. Bounded computation in Flink (in this case a graph snapshot) is also ingested internally as a stream, which makes Flink a convenient platform to build any graph as a stream ingestion scenario and implement complex partitioning logic. For the purposes of our evaluation framework, we have effectively implemented on Flink the general graph processing workflow presented in Figure 2.2. For staged and iterative tasks, we use the *DataSet* API to implement all respective transformations and application logic (i.e. iterative algorithms). Similarly, we use the *DataStream* API to implement all pipelined workflow steps. That includes all partitioning algorithms presented, as well as the single-pass stream processing applications.

Table 5.2 – Datasets information

Dataset	Vertices	Edges	Category
RMAT	500,000	9,127,486	Synthetic
RMAT	1,000,000	18,540,007	Synthetic
RMAT	1,500,000	28,181,948	Synthetic
RMAT	2,000,000	37,547,390	Synthetic
RMAT	2,500,000	47,411,497	Synthetic
RMAT	4,000,000	80,000,000	Synthetic
RMAT	5,000,000	100,000,000	Synthetic
DBLP	317,080	1,049,866	Collaboration
Flickr	1,715,255	15,551,250	Social
Skitter	1,696,415	11,095,298	Computer
MovieLens	80,555	10,000,054	Rating
Twitter	41,652,230	1,468,365,182	Social
Friendster	65,608,366	1,806,067,135	Social

5.4.2 Datasets

Table 5.2 shows the characteristics of the datasets we use in our experiments. We have generated synthetic graphs of varying sizes using the RMAT (Recursive Matrix) model [130] implemented in Gelly [131], Flink's graph processing API. RMAT produces skewed graphs that follow a power-law degree distribution. Such graphs commonly appear in social network problems [19] and are thus interesting for our analysis. We also use real-world datasets, including the Flickr graph [132] from the Online Social Networks Research web portal [133] produced by [134], several graphs from SNAP [46] (DBLP and Skitter), the MovieLens 10M datasets from GroupLens [135], large Twitter graph [136] and Friendster [137].

5.4.3 Order

For the real datasets, we have used the original order in which they were generated by their source, usually ordered by IDs. Otherwise, we consider three stream orderings for iterating through our datasets: 1) *BFS* [138], in a breadth-first search traversal, a vertex of the graph is selected at random, then the neighbors of that vertex are processed first. After that, the next level neighbors (the neighbors of the neighbors) are processed. 2) *DFS* [138], similar to *BFS*, after selecting a vertex at random, depth-first search is performed starting from that vertex. 3) *Random* [139], this order assumes that the vertices or the edges arrive at random from the streaming source. All partitioning algorithms behave similarly for *BFS* and *DFS* orderings, thus we only present results with *DFS* in Section 5.5.2.1.

5.4.4 Metrics

We use the following metrics for evaluation:

Partitioning Performance. We measure the **throughput** of a partitioning algorithm as the number of edges or vertices it can process (assign to a partition) per second.

Partitioning Quality. We evaluate partitioning quality using three metrics. **Load balancing** indicates how well the computation load is divided across partitions. Specifically, we calculate the normalized load for the highest loaded partition using the following formula:

$$\rho = \frac{\text{Load on highest loaded partition}}{\frac{n}{k}} \tag{5.7}$$

where n is the input size (number of edges for edge stream partitioning or number of vertices for vertex stream partitioning) and k is the total number of partitions.

Edge-cut measures the fraction of edges cut from the resulting partitions. We calculate it using the following formula:

$$\lambda = \frac{\text{No. of edges cut by the partitions}}{\text{Total no. of edges}} \tag{5.8}$$

This metric applies to vertex partitioning algorithms only.

Finally, the **replication factor** indicates how many vertex copies an edge partitioning algorithm creates. We calculate it as follows:

$$\sigma = \frac{\text{Total vertex copies}}{\text{Total no. of vertices}} \tag{5.9}$$

Application Performance. We evaluate the partitioning quality and performance for both vertex and edge partitioning methods, but the application performance is evaluated using edge partitioning because it partitions power-law graphs better in terms of low communication cost than vertex partitioning. Also, some vertex partitioning algorithms require a priori knowledge, i.e, $|V|$ and $|E|$, making them unsuitable for processing continuous streams.

Next, we evaluate the effect of partitioning algorithms on the performance of graph analysis applications. Particularly, for iterative applications we measure complete application **execution time** which consists of partitioning time spent during the stream ingestion phase and computation time spent during the compute phase of the staged workflow. We report the ratio of partitioning time over total application execution time. This metric provides a good indication of the impact a partitioning method can make to the performance of an application. We also report the ratio of total application execution time when using a partitioning method over the execution time when using hash partitioning as a baseline. It is a meaningful

metric to infer the cases where the partitioning cost is amortized. In the case of single-pass stream processing applications, we measure **the number of edges processed** during the whole pipelined workflow. This indicates which algorithm yields lower end-to-end latency. Additionally, for both these applications, we measure the **communication cost** as the ratio of the network traffic when using a partitioning method over the network traffic when using hash partitioning as a baseline.

5.4.5 Environment Setup

We deployed our experiments both on-premises on a university cluster as well as using a virtualized environment at Amazon EC2. The specs of the physical on-premises nodes are `2x Intel(R) Xeon(R) CPU @ 2.80GHz`, 44GB of RAM and Linux OS. This setup applies to all experiments in Sections 5.5.1 and 5.5.2. For the experiments in Section 5.5.3 we used up to 17x `r3.2xlarge` EC2 instances. The exact number of virtual instances in the latter case is experiment-specific and depends on the dataset size and the application type.

For our on-premises deployment, we set up Flink with one Job Manager (master node) and two Task Managers (workers). We further shared equally the amount of slots (allocated tasks) throughout workers. Regarding the virtual EC2 deployment we used one instance as the JobManager and the rest as TaskManagers. Finally, we used Flink v1.2.0 with Java 8 (Oracle JVM).

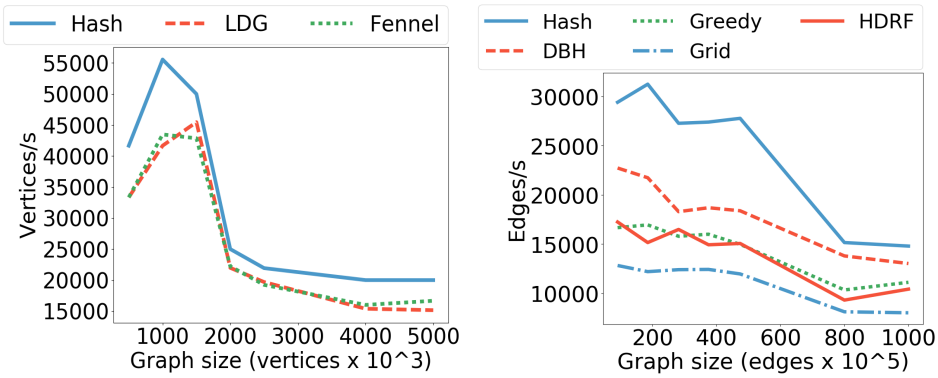
5.5 Evaluation Results

In order to answer **Q1** proposed in Section 5.4, we present partitioning performance results in Section 5.5.1 and partitioning quality (edge-cuts, replication factor and load balancing) results in Section 5.5.2. We give answers to **Q2** and **Q3** by results related to application performance in Section 5.5.3.

5.5.1 Partitioning Performance

We measure the throughput of vertex and edge partitioning algorithms with varying graph sizes. We use synthetic RMAT graphs and set the number of partitions to 4.

Vertex Partitioning. Figure 5.1(a) shows throughput measurements in vertices processed per second for vertex partitioning methods. Throughput initially increases with the graph size for all algorithms and then drops sharply for graphs with 20×10^5 vertices or larger. Overall, Hash partitioning demonstrates superior performance, while Fennel and LDG behave worse but similar to each other.



(a) Vertex partitioning

(b) Edge partitioning

Figure 5.1 – Throughput of partitioning algorithms using 4 partitions. Input: RMAT graphs.

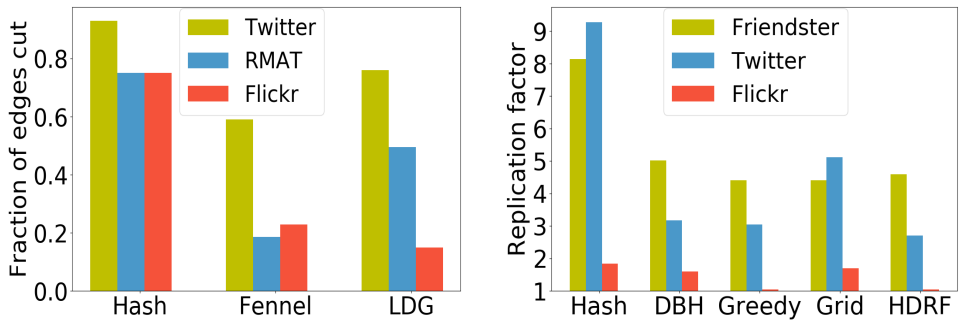
Edge Partitioning. Figure 5.1(b) plots throughput for edge partitioning algorithms. Hash partitioning shows the highest throughput as compared to all other methods. DBH has the second best throughput, followed by Greedy and HDRF which show almost identical performance. Finally, the Grid partitioner ranks last in terms of throughput.

Findings. Our results so far demonstrate that Hash partitioning outperforms all other evaluated methods in terms of throughput. However, the difference in performance is not dramatic. In both experiments, Hash shows at most 2x higher throughput than that of the second best partitioning method and the gap shrinks for larger graphs.

5.5.2 Partitioning Quality

A good partitioning method is not only fast but also produces high-quality partitions. We evaluate partitioning quality by first measuring the edge-cut for vertex partitioners and the replication factor for edge partitioners using different datasets. Then we measure the load balancing for these datasets. Next, we evaluate how stream order affects these results in Section 5.5.2.1. Finally, we examine how the number of partitions affects these results in Section 5.5.2.2.

We measure the partitioning quality using different input graphs. We use the Friendster, Twitter, largest RMAT, and Flickr graphs. We set the number of partitions to 16 for Friendster and Twitter and to 4 for the two smaller graphs. We stream all graphs in the order in which they are generated by their source. The power-law exponent that controls skewness is 1.7 for Flickr and has a very low value for RMAT making it highly skewed.



(a) λ for vertex partitioning

(b) σ for edge partitioning

Figure 5.2 – Fraction of edges cut λ and replication factor σ for different types of graphs.

Table 5.3 – Normalized maximum load ρ for vertex partitioning algorithms.

Dataset	Hash	Fennel	LDG
Twitter	1.0	1.1	1.13
RMAT	1.0	1.1	1.5
Flickr	1.0	1.0	1.0

Edge-Cuts. Figure 5.2(a) shows the fraction of edges cut, i.e., λ for Twitter, RMAT, and Flickr. Hash has the highest λ value for all three datasets, since it does not take into account vertex locality. In fact, more than 70% of edges are cut for RMAT and Flickr, while on Twitter Hash generates 90% edge-cut. Fennel has the lowest cuts for Twitter and RMAT and it is only slightly outperformed by LDG for Flickr. In the case of RMAT, LDG produces more cuts than Fennel because RMAT is highly skewed compared to the other datasets.

Replication Factor. Figure 5.2(b) shows the replication factor, σ, for Friendster, Twitter, and Flickr. Hash has the highest σ of all methods across all datasets. The rest of the algorithms perform quite similarly, with Grid performing worst for Twitter and DBH for Friendster. We also observe that overall Greedy and HDRF perform better than other algorithms for all the datasets by giving lower σ. Specifically, for Flickr, Greedy and HDRF have $\sigma \approx 1$.

Vertex Partitioning Load Balance. Table 5.3 shows the normalized load, ρ, defined in Section 5.4.4, using vertex partitioning algorithms. Hash gives perfectly balanced partitions because of its random placement. Fennel and LDG have nearly perfect load balance for Twitter and Flickr. However, LDG does not balance RMAT well compared to Fennel because LDG uses a greedy placement of vertices, and RMAT is highly skewed.

Table 5.4 – Normalized maximum load ρ for edge partitioning algorithms.

Dataset	Hash	DBH	Greedy	Grid	HDRF
Friendster	1.0	1.001	1.0	1.0	1.0
Twitter	1.0	1.001	1.0	1.0	1.0
Flickr	1.001	1.002	3.98	1.0	3.98

Edge Partitioning Load Balance. Table 5.4 shows the results for ρ using edge partitioning algorithms. All algorithms yield almost perfectly balanced partitions with $\rho \approx 1$ for all graphs except Flickr. When partitioning Flickr with HDRF and Greedy the result has a lower replication factor compared to others, hence generating unbalanced partitions.

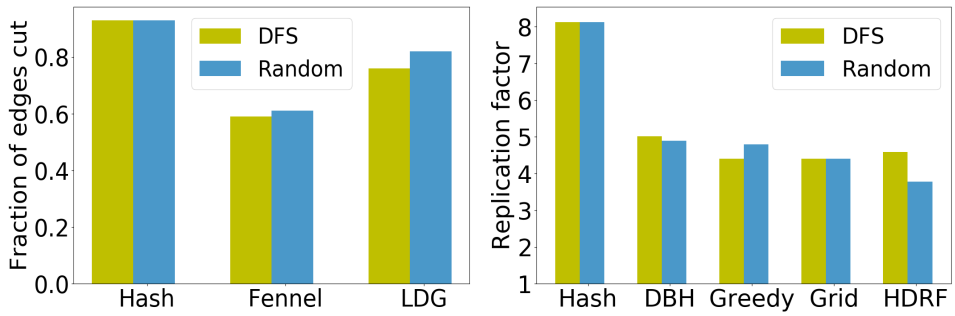
5.5.2.1 Sensitivity to Order

We now investigate how stream ordering affects the partitioning quality in terms of cuts and load balance. Some algorithms are particularly sensitive to the order in which they receive edges and vertices for processing. For example, *BFS* order is bad for Greedy because all neighboring vertices that arrive together might end up in the same partition. As a matter of fact, neighboring nodes do often arrive together in a graph stream. For instance, nodes grouped based on location in social graphs and links connecting web pages. We simulate this locality using *BFS* and *DFS* ordering and also use *Random* order as a baseline. To measure the effect of order on the partitioning quality in terms of λ, σ and ρ, we stream the Twitter and Friendster graphs in different orders and set the number of partitions to 16.

Edge-Cuts. Figure 5.3(a) shows the results for λ using ordered streams. Hash performs worst with highest λ for all orders. Moreover, λ remains the same using Hash despite of the change in order. LDG has higher λ for *Random* order compared to the *DFS* order. In the case of Fennel, we see that λ is not affected by the order. This can actually be controlled by changing the γ parameter values. In this experiment, we have set $\gamma = 1.5$, which according to [23] makes the algorithm less sensitive to the order. Overall, Fennel also has lower λ values than both Hash and LDG.

Replication Factor. Figure 5.3(b) contains the results for σ using ordered streams. Hash has the highest σ, while the other algorithms perform better and similar to each other. Hash and Grid are unaffected by order. HDRF has lower σ using *Random* order compared to the *DFS* order.

Vertex Partitioning Load Balance. Table 5.5 displays results for ρ using vertex partitioning algorithms on different stream orders. Hash and Fennel give well balanced partitions for all orders. LDG has slightly better results for *Random* order. Overall, we conclude that none of the studied algorithms is highly sensitive to order when balancing partitions.



(a) λ for vertex partitioning (b) σ for edge partitioning

Figure 5.3 – Fraction of edges cut λ and replication factor σ for different input orders, using 16 partitions. Input: Twitter (vertex partitioning) and Friendster (edge partitioning).

Table 5.5 – Normalized maximum load ρ values for vertex partitioning algorithms using 16 partitions. Input: Twitter.

Order	**Hash**	**Fennel**	**LDG**
Random	1.0	1.1	1.12
DFS	1.0	1.1	1.13

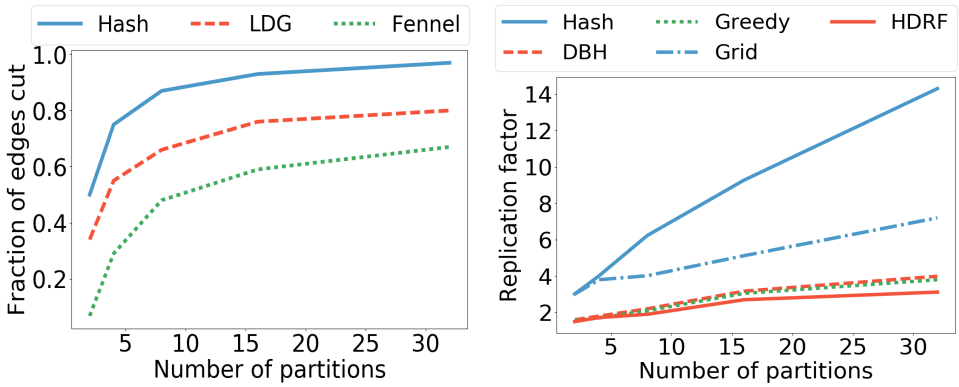
Table 5.6 – Normalized maximum load ρ values for edge partitioning algorithms and 4 partitions. Input: MovieLens.

Order	**Hash**	**DBH**	**Greedy**	**Grid**	**HDRF**
Random	1.001	1.005	1.0	1.0	1.0
DFS	1.001	1.006	4.0	1.0	4.0

Edge Partitioning Load Balance. Table 5.6 contains the result for ρ using edge partitioning algorithms on different stream orders for the MovieLens graph. We omit the results for the Friendster graph, as they were almost identical for both orders and all algorithms produced well-balanced partitions. For MovieLens, Greedy and HDRF show imbalance because they greedily place the neighboring edges arriving in the stream, together in the same partition.

5.5.2.2 Sensitivity to The Number of Partitions

We evaluate the effect of the increase in the number of partitions on λ, σ and ρ by taking the Twitter graph and partitioning it across a range of partitions from 2 to 32.



(a) λ for vertex partitioning

(b) σ for edge partitioning

Figure 5.4 – Fraction of edges cut λ and replication factor σ for different number of partitions (2 to 32). Input: Twitter.

Edge-Cuts. Figure 5.4(a) plots λ for vertex partitioning algorithms with increasing number of partitions. λ increases with more partitions for all the algorithms. While Hash performs poorly for a high number of partitions, LDG approaches 0.8 for 32 partitions and Fennel produces few edge-cuts even with many partitions, with λ slightly above 0.6.

Replication Factor. Figure 5.4(b) plots σ for edge partitioning algorithms using different number of partitions. For all algorithms, σ increases with the increase in the number of partitions. Hash shows a steep increase, while Grid shows the second worst behavior. σ for DBH, HDRF, and Greedy does not exceed a value of 4 even for 32 partitions.

Vertex Partitioning Load Balance. When examining how the number of partitions affects load balancing, we find that both Hash and Fennel have $\rho \approx 1$, thus we omit the results for these methods. Figure 5.5 plots ρ for LDG on Twitter, which is the only algorithm with different behavior. We see that LDG is affected by the number of partitions and its load factor increases, reaching a value of 1.15 for 32 partitions.

Edge Partitioning Load Balance. Increasing the number of partitions does not significantly affect load balancing for the Twitter graph regardless of the edge partitioning algorithm. All methods produce almost perfectly balanced partitions with $\rho \approx 1$.

Findings. We can summarize our results so far into the following observations: 1) Hash gives well-balanced partitions in all cases but produces many edge-cuts and high vertex replication. It is not sensitive to order and it behaves worse in terms of cuts when increasing the number of partitions. In the next section, we investigate

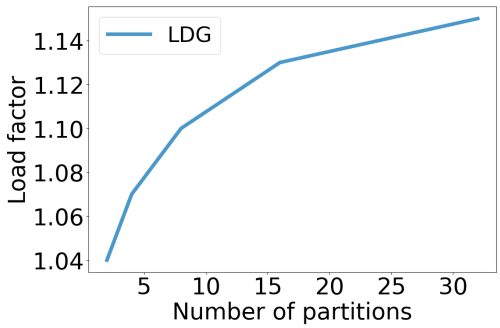


Figure 5.5 – Normalized maximum load ρ for different number of partitions (2 to 32). Input: Twitter.

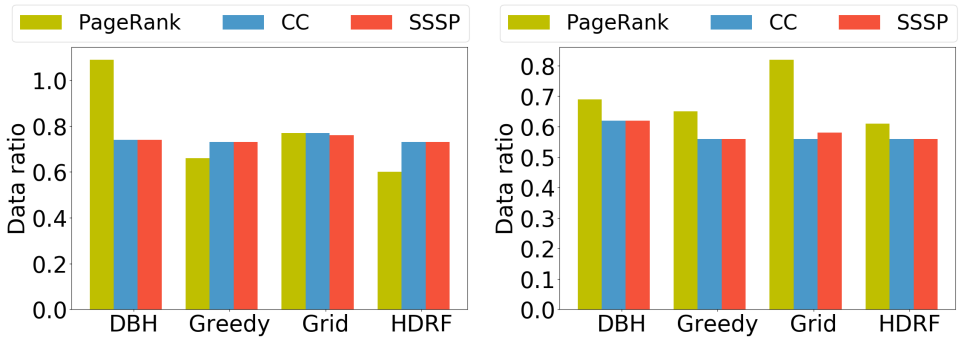
how the low partitioning quality it provides in some cases affects application performance and when its perfect load balancing proves to be beneficial. 2) Fennel and LDG provide low edge-cuts; LDG is sensitive to order; Fennel is tunable. 3) HDRF and Greedy give low vertex-cuts, but both are sensitive to order. 4) Grid and DBH give moderate vertex-cuts; Grid and DBH give almost perfectly balanced partitions.

5.5.3 Application Performance

Considering our previous observations regarding partitioning algorithms, we would like to understand how the partitioning quality of different algorithms affects the performance of applications. We examine two factors that can have an impact: (a) the partitioning performance, i.e, the one-time overhead of the partitioning algorithm when the graph stream is ingested and (b) the partitioning quality, i.e, the load balance and cuts that the partitioning method produces. Good load balancing is important for distributed execution because it lowers the probability of straggler workers and computation skew. On the other hand, a low number of cuts usually enables distributed algorithms to perform as much computation as possible locally and minimize cross-partition communication. To evaluate these factors we use the Twitter and Friendster graphs and partition them across 16 partitions. We examine the effects of partitioning on performance both on the batch, iterative applications, as well as on single-pass distributed streaming applications.

5.5.3.1 Iterative Applications

We first measure the ratio of network traffic as the communication cost for different partitioning methods. Next, we measure the application execution time and report the ratio of partitioning time over the total application execution time. We also



(a) Twitter (b) Friendster

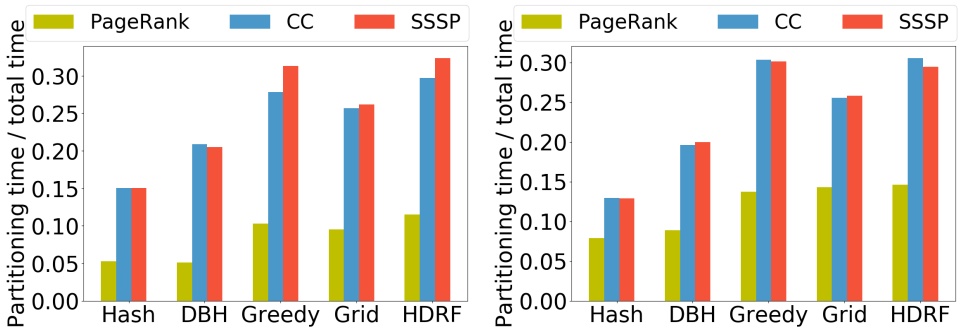
Figure 5.6 – Communication cost of partitioning algorithms as compared to Hash for iterative applications on Twitter and Friendster.

compare the total application execution time for different partitioning methods with that of Hash. We run 10 iterations for the considered applications.

Communication Cost. Figure 5.6 plots the results for the ratio of network traffic using a partitioning method over the network traffic using the baseline Hash partitioning. In the case of SSSP and Connected Components (CC), data exchanged between the partitions using other partitioning algorithms is lower compared to Hash. All algorithms perform similarly with Greedy and HDRF being slightly better. With regards to PageRank on Twitter, DBH shows poor behavior and even exchanges more data than the baseline.

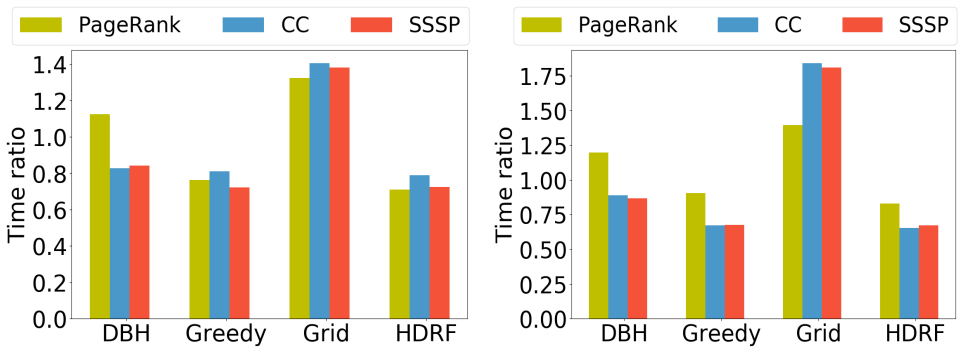
Execution Time. Figure 5.7 plots the ratio of partitioning time over the total execution time (partitioning time and computation time) for iterative applications. Here, we want to compare the effect of different partitioning algorithms on the execution time. Greedy, Grid and HDRF have high ratio for all applications; whereas the ratio of DBH is almost as low as of Hash. The ratio is much lower for PageRank than for SSSP and CC across all methods.

Figure 5.8 shows the ratio of total execution time for applications using different partitioning algorithms over the total execution using the baseline Hash. Here, we want to examine whether the partitioning time for expensive partitioning methods can be amortized by improved application performance. For both datasets, Grid results in the highest total execution time for all applications. After Grid, Hash yields higher execution time compared to others followed by DBH, for SSSP and Connected Components; whereas, for PageRank, the total execution time using DBH is higher than that of Hash. Finally, Greedy and HDRF result in lower total execution time for all applications. Overall, HDRF and Greedy improve the performance of iterative applications by reducing computation times.



(a) Twitter (b) Friendster

Figure 5.7 – Partitioning time over execution time ratio for iterative applications on Twitter and Friendster.



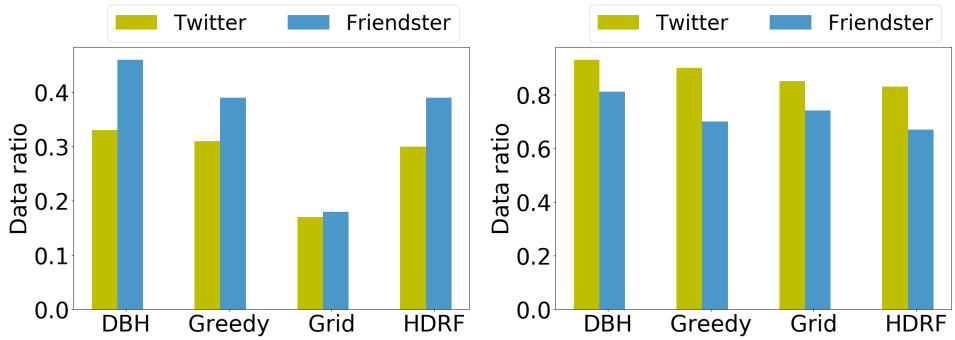
(a) Twitter (b) Friendster

Figure 5.8 – Total application execution time of partitioning algorithms as compared to Hash for iterative applications on Twitter and Friendster.

5.5.3.2 Single-Pass Applications

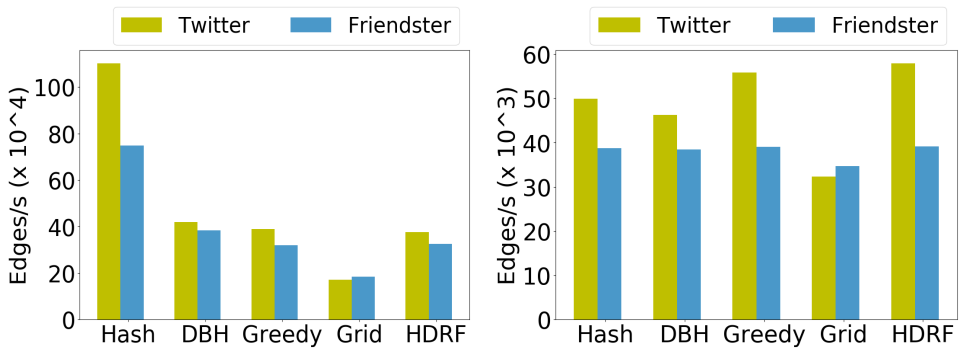
We ingest the Twitter and Friendster graph streams without using any a priori information to emulate the behaviour of unbounded streams and partition them. Next, we run the applications over the incoming partitioned streams for an interval of 15 min. After this, we measure: 1) the communication cost as the ratio of network traffic, and 2) the number of edges processed per second during the execution of the application to indicate which partitioning algorithm improves the latency of edges.

Communication Cost. Figure 5.9 shows the ratio of network traffic using a partitioning method over the network traffic using the baseline Hash. Grid minimizes network traffic for Bipartiteness check, while none of the partitioning algorithms



(a) Bipartiteness check (b) Connected components

Figure 5.9 – Communication cost of partitioning algorithms as compared to Hash for streaming applications on Twitter and Friendster.



(a) Bipartiteness check (b) Connected components

Figure 5.10 – Average throughput (edges/s) for streaming applications on Twitter and Friendster.

provides impressive results for Connected components. For this application, HDRF has low communication cost for both Friendster and Twitter.

Number of Edges Processed. Figure 5.10 plots the average throughput in edges per second for Bipartiteness check and Connected components. Hash results in significantly superior performance for Bipartiteness check, where the state requirements are lower. In the case of Connected components, HDRF and Greedy either match or exceed the performance of Hash. Grid partitioning results in poor throughput for both applications and both datasets.

Findings. Our last results demonstrate that HDRF and Greedy, which have lower replication factor, yield higher partitioning cost that is amortized by lower compu-

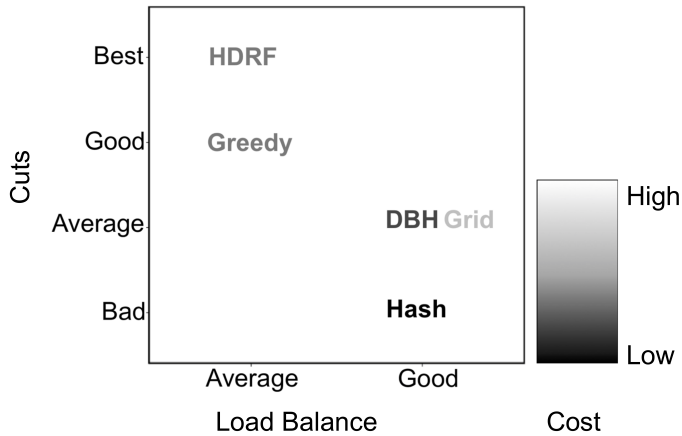


Figure 5.11 – Cuts, load balance and partitioning cost comparison of edge partitioning algorithms.

tation time for iterative applications and better end-to-end latencies for single-pass stream processing applications where data locality significantly affects the communication cost. Hash, with lowest partitioning time, improves the performance of applications when data locality does not significantly affect communication cost. In most cases, Hash results in higher communication cost; however, its partitioning overhead is negligible. HDRF and Greedy are effective at minimizing communication costs; however, they are more beneficial for computation and communication-intensive applications, since their partitioning costs cannot be easily amortized.

5.5.4 Discussion

We have arranged the partitioning algorithms based on our findings in Figure 5.11. The partitioning cost is indicated by shades of gray. Hash has the lowest partitioning cost; it is shown in the darkest color. Grid with highest partitioning cost is shown in lightest shade. For load balancing, DBH, Grid, and Hash give well-balanced partitions. Finally, HDRF provides the lowest vertex-cuts. Overall, the trade-off between balancing and reducing cuts remains. None of the studied algorithms provides both low cuts and perfect load balancing.

Among these algorithms, the low-cut partitioning methods, such as HDRF and Greedy, improve the performance of iterative applications, which have frequent communication between partitions during the compute phase. On the other hand, the impact seems to be less significant for streaming graph applications. Low-cost, partitioning algorithms, such as Hash appear more beneficial, probably because they have a pipelined compute phase and no state requirements.

5.6 Related Work

A number of surveys [40, 41, 42, 43] have focused on offline partitioning algorithms in the past. Besides, several online partitioning methods have been surveyed in the context of the *load-compute-store* model and bounded graphs (snapshots) [45]. A survey by Guo et.al. [44] exclusively covers vertex partitioning algorithms. Our work is, to our knowledge, the first dedicated study to online graph partitioning methods that includes stream-specific properties (e.g., ingestion order) as well as considering single-pass graph stream aggregations, an emerging application domain with increasing system support [28, 29, 30, 31, 10].

5.7 Acknowledgements

This work is an extension of my master thesis [24], which was supervised by Vasikili Kalavri, Paris Carbone and examined by Vladimir Vlassov. I am very thankful for their support. It would not have been possible without their guidance, constant feedback and help. Special thanks to Seif Haridi for helping us run the experiments on amazon machines, and Leila and Amira for reviewing this work.

5.8 Summary

In this chapter, we address the challenge of *CH3-Streaming Graph Partitioning*, mentioned in Chapter 1 and made contributions in this domain by providing *C3*-a formal definition of streaming graph partitioning task as an optimization task at the early stage of graph processing pipeline/workflow, and *C4*-a comparison framework and an experimental evaluation of streaming graph partitioning algorithms. We have studied streaming graph partitioning algorithms and we have empirically compared them using a framework based on Apache Flink [30]. We have evaluated the partitioning quality and performance and we have measured the effect of partitioning on application performance, using both iterative and streaming applications. We conclude that algorithms aiming for optimal cuts, such as HDRF and Greedy, exhibit higher online partitioning cost that is otherwise amortized throughout the graph computation when that computation is sensitive to data locality and associated communication costs. Otherwise, when the computation is not directly affected by data locality, it is preferred to use online partitioning algorithms that aim for load balancing and performance (e.g., Hash and DBH). Several open challenges remain in the area of streaming graph partitioning. We highlight the need for developing new, scalable online partitioning algorithms, with relaxed constraints on the graph properties and fewer state requirements. The next chapter focuses on addressing the aforementioned challenges. We believe that the development of

such algorithms is crucial for making graph partitioning practical for applications ingesting continuous streams on top of modern distributed streaming engines.

CHAPTER

6

Partitioning Un-bounded Graph Streams

"I need a meme for when your paper gets rejected twice with the comment
'starts a new line research' yet the meta-review claims low novelty"
— Vasiliki Kalavri a.k.a Vasia on Twitter @vkalavri.

6.1 Introduction

Online graph partitioning methods [21, 22, 23, 2, 24] process graph streams and assign edges or vertices to partitions on-the-fly. To make high-quality partitioning decisions on streaming graphs, state-of-the-art algorithms either accumulate growing states or optimize for load balancing, sacrificing data locality. Existing solutions for online partitioning fall in one of two extremes, as depicted in Figure 6.1. On one end, stateful methods, such as HDRF [21], yield the best-known performance in terms of minimum IO cost and good balance but have $O(|V|)$ state complexity or higher, where $|V|$ is the number of unique vertices. The size of the accumulated state, as well as the necessity to frequently access and update it, become a bottleneck for high-throughput streams. However, due to their unbounded model size complexity and lack of ability to infer partitions on unseen data, such methods are impractical for applications that continuously process unbounded graph streams. At the other end, stateless approaches such as the hash-based partitioner achieve good balance at the expense of degraded partitioning quality.

Instead, we propose *GCNSplit*, the first in a new class of graph partitioning algorithms that can operate online under unbounded executions. Motivated by the recent work of GAP [63, 64] that leverages GCNs for offline graph partitioning, we examine how ML-aided graph partitioning can be applied in an edge streaming setting. To achieve this goal several challenges need to be addressed. First, as GAP

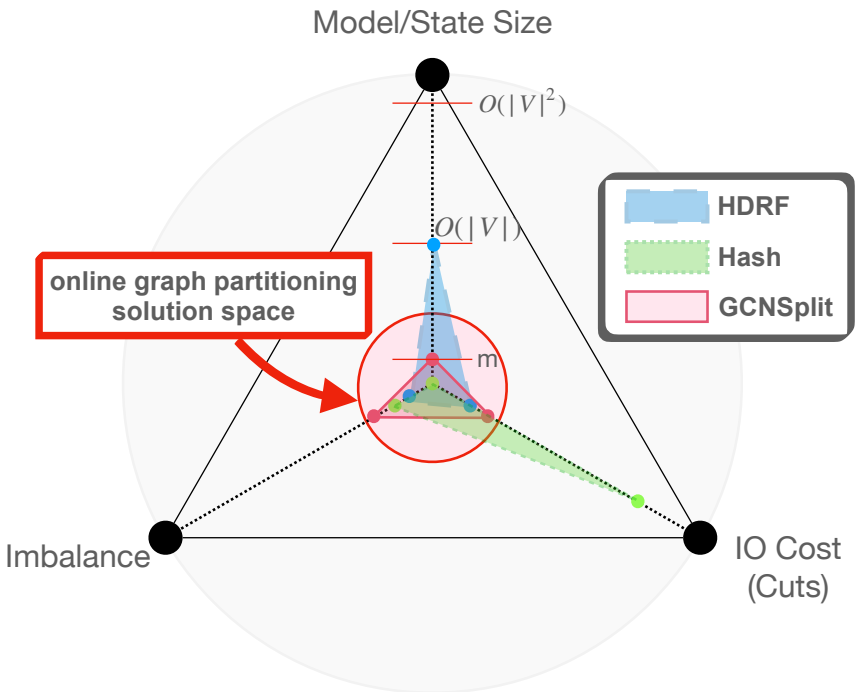


Figure 6.1 – Objectives of graph partitioning and corresponding methods. *GCNSplit* uses a cut-minimization loss function to reduce I/O alongside a load balance constraint. At the same time, it relies on bounded partitioning models whose size is independent of the graph stream's length.

requires prior knowledge of the full graph during the training phase, it cannot be directly used on continuously ingested graph streams. Further, GAP is a vertex partitioning method and applying its loss function to edge partitioning leads to high load imbalance. *GCNSplit* overcomes these limitations by providing (i) offline training on a small graph sample coupled with continuous and scalable online inference, (ii) two assignment heuristics that combine vertex embeddings to make edge assignment decisions and (iii) load constraints to ensure good load balance across partitions. The ability of GCNs to effectively encode graph characteristics into fixed-size models allows *GCNSplit* to partition graph streams with a bounded state. By employing inductive graph representation learning, *GCNSplit* produces high-quality partitions for parts of the graph stream unseen during the training phase, as well as for entirely unseen graphs.

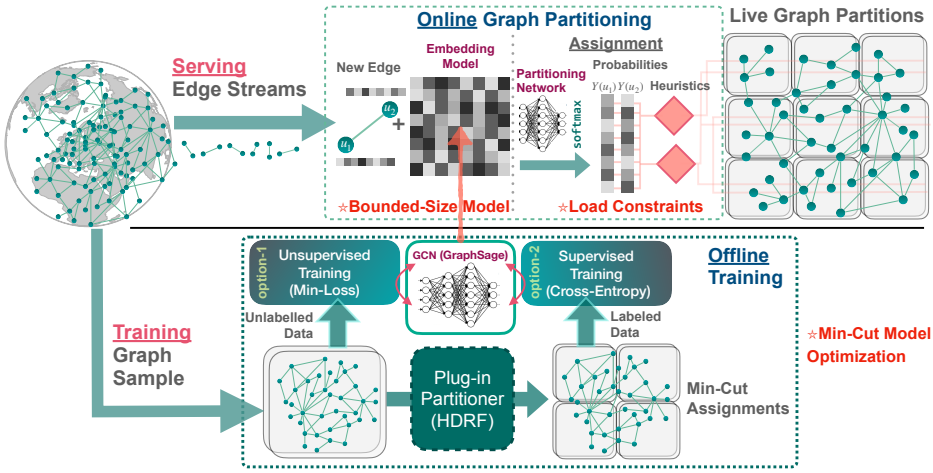


Figure 6.2 – GCNSplit Overview - Training and live Partitioning

6.1.1 Approach Overview

Recent breakthroughs in representation learning, such as GraphSAGE [65], have enabled effective dimensionality reduction for large graphs and shown promising predictive performance capabilities. The essence of inductive Graph Convolutional Networks (GCN) is to exploit features associated with vertices and edges as well as the graph structure to build convolutional neural networks that summarize the graph.

GCNSplit targets attributed graphs encountered in several domains, including social networks, IoT applications, natural and medical sciences, citation networks, and online transaction systems [140, 65, 141, 142, 143]. Figure 6.2 summarizes its design and application setting. The framework we describe in this paper consists of 1. an offline training pipeline and 2. an online partitioning pipeline.

Training. The *GCNSplit* training pipeline supports both unsupervised and supervised training from a sample or snapshot of the target graph or another graph that exhibits the same features and similar structure. Unsupervised training is supported through a multi-objective loss function. Whereas, supervised model training builds upon the assignments instrumented by a given offline partitioning method (e.g., HDRF). In both cases, the resulting model is optimized to provide high-quality partitioning decisions by minimizing cuts.

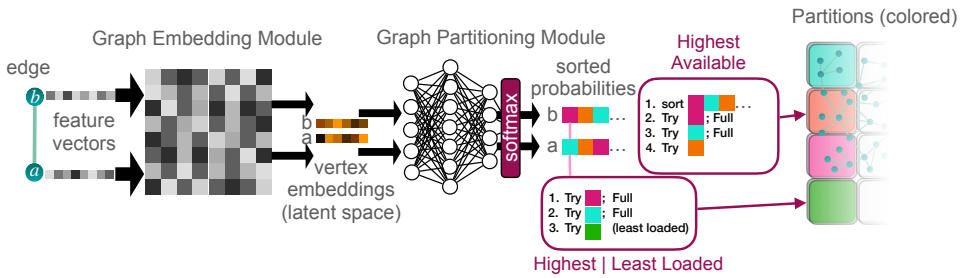


Figure 6.3 – Online graph partitioning framework components and example

Serving. The networks resulting from the training pipeline are then used as a fixed-size serving model to perform online partitioning decisions on unseen data (i.e., attributed edge streams). The size of the model is determined by the number of vertex features and layers of the GCN model in use while being independent of the size of the graph, as we explain in Section 6.2. For each input edge, *GCNSplit* first extracts two vectors in the embedding space, one for each endpoint vertex. These embeddings are then used as an input to the partitioning network, which has been trained using the same cost function. Next, *GCNSplit* determines the target partition for the edge as follows. It extracts vertex assignment probabilities for each endpoint vertex of an edge from the partitioning network (softmax function) and ranks them according to a pre-selected heuristic that ensures the fulfillment of a load balancing constraint.

6.2 Design of *GCNSplit*

GCNSplit is a GCN-based partitioning framework for graph streams that can scale to unbounded inputs. In this section, we explain how *GCNSplit*'s design addresses GAP's limitations to make GCNs applicable to the streaming setting. We provide an overview of *GCNSplit*'s functionality and architecture in Section 6.2.1. Section 6.2.2 describes how the offline training phase generates fixed-size models that can be used for partitioning unbounded edge streams. Finally, we discuss *GCNSplit*'s online operation and heuristics in Section 6.2.3. The necessary preliminaries for this chapter are presented in Chapter 2 Section 2.3 and related work is highlighted in Section 6.7.

6.2.1 Framework Overview

To extend GAP's model to the edge streaming setting, *GCNSplit* involves two core modules in taking online partitioning decisions: a 1. graph embedding and 2. graph partitioning module, as shown in Figure 6.3. The *Graph Embedding Module* is

responsible for the continuous encoding of the non-euclidean input graph stream into vectors of defined size in latent space. The *Partitioning Module* consists of an ML-based partitioning algorithm that assigns incoming edges to partitions. The partitioning objective function aims to minimize the replication factor while taking into account load balancing constraints via its assignment heuristics.

Both models used in the embedding and partitioning of the graph stream are trained jointly offline on a sample of the graph stream using the same objective function. For generality, the *GCNSplit* framework supports two distinct methods of training the two models, a supervised and an unsupervised one. Each method results in a different objective function, however, both of them have the same goal: to lead to an embedding representation and online partitioning model which minimizes cuts while not overloading partitions. After discussing the components, we discuss the two corresponding training approaches in detail. It is worth mentioning that in certain scenarios, the node embedding and partitioning modules could potentially be trained separately or replaced without the need to adjust other parts of the framework. At the final partition selection step, a set of heuristics is used in order to incorporate load metrics known at runtime to ensure good long-term balance while avoiding partition overloading constraints during unbounded executions.

Graph Embedding Module. The graph embedding module uses GraphSAGE to encode input edges and their corresponding vertices into numerical, vector representations. It is therefore important that the input graph stream is annotated with features for our scheme to work efficiently. *GCNSplit* uses an element-wise max function to aggregate vertices' vector representations and vector concatenation as the combine function. The trainable parameters of the node embedding generation module include the W^l matrices of all the l layers of GraphSAGE. The dimension of the matrix from the first layer of the network (W^1) is defined by the size of the feature set used by vertices in the training dataset. The sizes of the matrices from the following layers are equal to the embedding size, which is controlled by a hyperparameter. Thus, the number of the trainable parameters depends on the dimensions of the feature set and the chosen embedding size while being independent of the size and scale of the data it is served. The last layer of the embedding model yields a vector representing a particular vertex in a latent space that is passed as an input to the partitioning module. The list of parameters, including the number of network layers and the embedding size can be found in the project repository [144].

Partitioning Module. The partitioning module converts graph input data from its vector representation in the embedding latent space to vectors $\mathbf{Y} \in \mathbb{R}^{n \times k}$, where $\mathbf{Y}_{ij}$ represents the probability that a corresponding vertex v_i belongs to the partition $j \in \{1, 2, \cdots, k\}$; n is the number of items and k is the total number of partitions. This module is implemented as a fully connected neural network containing a

`softmax` function at its end that turns the k-dimensional vector into a probability distribution over the k partitions. For each element i, the softmax function is represented as follows:

$$Softmax(x_i) = \frac{\exp x_j}{\sum_{j=\{1,2,\cdots k\}} \exp x_j} \tag{6.1}$$

The trainable parameters of the partitioning module are the weights of the connections between neurons of consecutive layers in the neural network. Whereas, the number of parameters depends on the number of layers, the size of the vector and the number of partitions. The first layer of the partitioning network consumes the output of the embedding module, thus, the number of neurons in the first layer is equal to the vector length. The number of neurons in the last layer of the network is equal to the number of partitions.

6.2.2 Model Training

GCNSplit allows training a partitioning model offline in either supervised or unsupervised mode. In the *unsupervised* mode, a loss function drives the model training towards edge-cut minimization. Whereas, in *supervised* mode, a given existing graph partitioner is used to produce labels that correspond to partition assignments. Despite the mode of operation *GCNSplit* generates both embedding and partitioning networks in the training process (loss function re-use), while not restricting the training and application graphs to be of the same origin, as long as they have matching feature sets.

Loss Function Re-use. The training module produces two models using the same loss function: an embedding and a partitioning network. These can potentially be trained separately depending on the encoder used. However, we chose to execute the training of the two components jointly for simplicity and performance using the same loss function to obtain their parameters. As a result, we could exploit GraphSAGE's ability to optimize its parameters based on a differentiable loss function.

Training Graph. Training in *GCNSplit* is executed offline on a snapshot or sample of a streaming graph. The training data is extracted using a fixed-size window or snapshot of the graph stream but it can also be a random sample. Furthermore, the training data does not need to come from the same origin graph that is partitioned. An entirely different graph can also be used for training as long as it shares a similar feature set with the graph that the model is applied for partitioning. In Section 6.2.4 we provide further insights on the generalization to unseen graphs.

Given the differences across the two respective modes of operation, such as the loss function used to optimize the models, we further explain the most distinct parts of each approach in detail.

Algorithm 3 Unsupervised Partitioning Model Training

Input: dataset $X \in R^m$, embedding size d, number of partitions k, batch size b

1: **while** Loss not converged (eq 6.2) **do**
2: b=sample(X), $\mathbf{D} \leftarrow$degree matrix, $A \leftarrow$adjacency matrix ▷ `Generate embedding by passing the feature vectors through the GraphSAGE network`
3: $z_1, ..., z_b; z_i \in R^d \leftarrow \text{GraphSAGE}(v_1, ..., v_b; v_i \in R^m)$ ▷ `Obtain assignment probabilities by passing the embeddings through the partitioning neural network (PNN)`
4: $\mathbf{Y} \leftarrow \text{PNN}(z_1, ..., z_b), \mathbf{Y} \in R^k$ ▷ `Minimize the expected normalized cuts (eq 6.2)`
5: $\Gamma = \mathbf{Y}^T\mathbf{D}, \mathbb{L} \leftarrow update(\mathbf{Y}, \Gamma, A)$
6:

6.2.2.1 Unsupervised training

The main challenge in the unsupervised mode is that there is no indication of what a good partition assignment can be in any given unlabeled graph data. The training function itself is responsible for automatically discovering the partition assignments that provide good cuts and balance.

In Algorithm 3, we summarize the unsupervised training execution logic which is based on mini-batch gradient descent for fast convergence. A set of configurable coefficients (α, β) were used to regulate the importance of 1. a normalized cut and 2. cross-partition balance. This yields the following loss function, after applying necessary transformations :

$$\mathbb{L} = \alpha \sum_{reduce-sum} (\mathbf{Y} \oslash \Gamma)(1 - \mathbf{Y})^T \odot A + \beta \sum_{reduce-sum} (1^T\mathbf{Y} - \frac{n}{k})^2 \qquad (6.2)$$

The resulting embedding module (GraphSAGE) generates a single node embedding while the partitioning network (PNN in Algorithm 3) is used to generate the assignment probabilities (Algorithm 3 Line 4). The unsupervised loss is based on vertex-based partitioning (edge-cut), however, our target use-case of edge-partitioning targets optimal vertex-cut. As explained in detail in Section 6.2.3, we derive edge assignments by applying this model to the endpoint vertices of every edge in a stream.

Neighborhood Sampling. The sampling method used on each step of the mini-batch execution (Algorithm 3 Line 2) is instrumental to the model training. Due to the multiplication with the adjacency matrix, if nodes were sampled arbitrarily, the probability of choosing non-adjacent nodes would be high. Therefore, for each vertex within a mini-batch, we also ensure to include its direct neighborhood in the sample. This way, we update the model effectively without nullifying the normalized cut of the loss function.

6.2.2.2 Supervised Training

The supervised model bases model training on edge assignments that are generated using a provided algorithm on the training dataset. In this mode of operation, the embedding module receives edges and generates embeddings for both endpoint vertices. Then, embeddings are merged and processed by the partitioning module which outputs the assignment probabilities. In our prototype and evaluation section 6.4 we used the HDRF algorithm (described in Section 6.4.3) due to its best-known minimum cut performance, however, the reference algorithm is pluggable and could be replaced by another online or even offline partitioner.

The loss function for supervised training is based on the well-studied cross-entropy loss. Cross-entropy measures the performance of the model's inferences according to the known labels. The further the inferences are from the labeled attributes, the higher the value of the cross-entropy. In our case, the loss value decreases if the model gives the assignments equal to the assignments of HDRF. The formula for the supervised loss for each item is as follows:

$$\mathbb{L} = -\sum_{i=1}^{k} e_i \log(\mathbf{Y}_i) \tag{6.3}$$

e_i is a binary indicator for the particular element if it belongs to the partition i and $\mathbf{Y}_i$ is equal to the probability that this element belongs to the partition i inferred by the model.

As with the unsupervised approach, it is possible to train this model using a mini-batch execution. However, in this case, the loss function only requires training edges and their labels. Hence, every training step consists of a chosen number of edges without any neighborhood sampling. For each mini-batch, we take the average of the loss of all items in the mini-batch.

6.2.3 Model Serving

Since both the supervised and unsupervised training approaches result in two different models, their application methods to partition edge streams are also different. Nevertheless, for both approaches, we introduce the notion of *maximum load*. Enforcing load constraints is a critical extension we made to GAP's model. As we show in Section 6.5.4, GAP's loss function leads to 50-226% higher normalized load (load imbalance) compared to *GCNSplit* when partitioning the same graph. To ensure that all partitions sizes stay within limits during continuous inference, we control their sizes using assignment heuristics.

6.2.3.1 Assignment heuristics

We propose two heuristics namely HighestOrLeastLoaded and HighestAvailable to partition the edge stream using the partitioning models.

HighestOrLeastLoaded. The HighestOrLeastLoaded heuristic first tries to assign an edge to the partition with the highest assignment probability. If the assignment exceeds the load limit, we assign the edge to the currently least loaded partition.

HighestAvailable. The HighestAvailable heuristic handles the maximum load constraint differently. First, the edge assignment probabilities are sorted in descending order. Then, we go through the sorted list of partition indices one by one, until we find the first partition where we can place the new edge without exceeding the load limit.

6.2.3.2 Model Application

In online partitioning, edges are processed one at a time, or window-by-window, where a window is a group of consecutive edges. Thus, each edge assignment is performed independently of the other assignments. In Algorithm 4, we show the steps of the model serving process for the unsupervised-trained model. To put everything together, the model application scheme of *GCNSplit* receives unbounded edge streams and partitions edges by first converting their counterparts (vertices) into latent space vectors (Algorithm 4 Lines 2-3) and then feeding them into the partitioning network (Algorithm 4 Lines 4-5). Moreover, during the model's application to graphs (which are evolutions of a training dataset) training graphs can be used for inference. Namely, for the time of partition inference, an edge (or a window of edges) can be added to the training graph and later removed, as if it was a part of the graph from the beginning (Algorithm 4 Lines 1 and 6). As a result, the embedding network is able to generate more informative embeddings because it has information about the node's neighborhood rather than only the attributes of the edge endpoints.

In the case of the unsupervised-trained model, inference derives from corresponding assignment probabilities of the endpoint vertices generated from the partitioning network which takes two separate embeddings of the endpoint vertices as input and generates two corresponding assignment probability vectors. Whereas, in the supervised-trained model assignment probabilities are made directly to the input edge by combining the two embeddings of the endpoint vertices into one embedding that is fed to the partitioning network for generating an assignment probability vector.

Algorithm 4 Model Serving with Heuristic

Input: training graph G, number of elements in the partitions $S_1, S_2, ..., S_k$, vertices v_1, v_2, maximum load M
Output: partition ID i

procedure APPENDEDGE(G, v_1, v_2) ▷ `getting embeddings`
  $z_1 = \mathrm{GraphSAGE}(v_1)$
  $z_2 = \mathrm{GraphSAGE}(v_2)$ ▷ `getting assignment probabilities`
  $\mathbf{Y}_1 = \mathrm{PNN}(z_1)$
  $\mathbf{Y}_2 = \mathrm{PNN}(z_2)$
  $\mathrm{removeEdge}(G, v_1, v_2)$
  $i = \mathrm{applyHeuristic}(\mathbf{Y}_1, \mathbf{Y}_2, S, M)$

We have modified the assignment heuristics to operate on both vertices (unsupervised model) and edges (supervised model) as follows: In the case of the unsupervised model, $\mathit{HighestOrLeastLoaded}$ first attempts to assign the edge to the partition with the highest assignment probability of the endpoint vertices and then to the least loaded partition if the maximum load exceeds with the current assignment. For the supervised model, the same heuristic attempts to assign the edge to the partition with the highest assignment probability and if such an assignment exceeds the maximum load, the edge is assigned to the least loaded partition. $\mathit{HighestAvailable}$ sorts the assignment probabilities in descending order and iterates through the partitions until it encounters one with the capacity to accommodate the new edge. For the unsupervised model, the assignment probabilities of endpoint vertices are merged and then sorted.

Example: In Figure 6.3 we summarize the partitioning logic in an end-to-end example of model application from the ingestion of an edge to its final assignment. For an edge with corresponding vertices, a and b, a set of vectors are produced by the embedding module. Each vector is then passed to the PNN that outputs a $\mathbf{Y}_i$ vector of probabilities, corresponding to the likelihood of assigning vertex i to each target partition. The example demonstrates the assignment choices using the respective heuristic. With a $\mathit{HighestAvailable}$ policy a sorted vector is first created out of $\mathbf{Y}_a$ and $\mathbf{Y}_b$. Then, each partition is tested against the load constraints and the

first partition that does not violate the constraint is selected for assignment (orange). Similarly, with the HighestOrLeastLoaded strategy only the partitions with the top-most probabilities are first selected across $\mathbf{Y}_a$ and $\mathbf{Y}_b$. In this example, both of these (purple and blue) violate the load constraint so the least loaded partition is chosen for assignment (green).

6.2.4 Partitioning Quality and Applicability

Like any ML-driven application, *GCNSplit*'s effectiveness depends on the quality of the training data and the characteristics of the examples it will be applied on. The graph embedding process is tuned so that *similar* vertices are represented by vectors that are close to each other in the embedding space. Similarity refers to the structural position of nodes in the graph, as well as the statistical similarity of their associated features. As a result, we expect *GCNSplit* to be particularly effective on graph streams whose structural characteristics and feature distribution remain relatively stable over time. Nevertheless, if *GCNSplit* is applied on a graph stream with major concept drift, it will—in the worst case—behave like hash partitioning. The partitioning classifier will be assigning vertices to partitions at random, yet it will still be guaranteed to produce balanced partitions, due to the enforcement of the load balance constraint. We empirically verify this claim in Section 6.5.2.

With regards to generalization, *GCNSplit* is applicable to any unseen graph with the same feature set as the graph used for model training. As in the case of partitioning unseen vertices of the same graph, high structural and feature similarity between the target and training graphs is instrumental to high partitioning quality. Our evaluation results (cf. Sec. 6.5.3) indicate that a richer set of features leads to better generalization, however, we believe this requires a further investigation that is beyond the scope of this work.

6.3 Implementation

We now briefly outline the implementation of the *GCNSplit* framework. *GCNSplit* operates in two modes. Training is a batch process that takes place offline, before partitioning. The user needs to provide a training graph and select either the supervised or the unsupervised method. In the former case, they also need to provide a plug-in partitioner, which is set to HDRF by default. Partitioning is an online process that ingests a graph stream either edge-by-edge or in a micro-batch fashion. The ingestion window size is configurable. In this mode, the user needs to set the number of target partitions and select the partitioning model and heuristic to use.

Model implementation. We implement our GCN-based partitioning models using PyTorch. Our partitioning models are comprised of two components: 1) the GCN component, which generates node embeddings of the incoming graph stream using GraphSAGE and, 2) the partitioning component, which generates a probability vector for assigning the incoming edges to given partitions based on the previously generated embeddings using a 3-layered neural network. We adapted the GraphSAGE implementation from an existing code-base[1]. Both the GraphSAGE network and the partitioning network are built using PyTorch building blocks that provide support for implementing neural networks. Furthermore, PyTorch strong support for GPU makes operations such as matrix and vector multiplication fast for our neural network-based models.

Parallel model serving. Since models are immutable, *GCNSplit* can leverage data parallelism to allow for scalability and sustain high-throughput streams. In contrast to stateful algorithms like HDRF, *GCNSplit* does not need to remember past partitioning decisions and partitioner processes can operate independently of each other. The serving pipeline of *GCNSplit* is implemented as a set of processes that communicate via queues. At the top of the pipeline, a *stream producer* process ingests edge streams from a streaming source and pushes them into an ingestion queue. Next, a set of parallel *stream consumer* processes pull edges from the queue and invoke the partitioning method on them, going through the steps shown in Figure 6.3. Partitioning decisions are then written into an output queue that can be consumed by a downstream graph streaming application. These partitioning decisions are based on load values that are kept local by each partitioning process to ensure that the load does not exceed the set load limit. To implement the multi-process system and inter-process communication we utilized the *torch.multiprocessing* package, which is a wrapper around the Python's *multiprocessing* package optimized towards working with *torch.Tensor*. To ensure equal distribution of computing power between the processes running on the same machine, we limit each process to a single core.

State size configuration. Stateful streaming graph partitioning algorithms like HDRF keep partial vertex degrees and the previous assignments of the processed nodes so far as in-memory state. This state grows as more distinct vertices appear in the input stream. Instead, *GCNSplit* keeps a constant state in memory which depends only on the size of the machine learning model it produced during training. The model size depends on the number of training parameters (which depend on the number of layers in the model's network), the embedding size, and most importantly on the dimensions of the feature set. A dataset with a rich feature set will have a model size larger than that of another dataset with a less rich feature set. In all of our experiments, we found that effective models are no larger than a couple of MBs, and in most cases, they took only a few KBs of space in memory.

[1]https://github.com/twjiang/graphSAGE-pytorch

Table 6.1 – Graph datasets used for evaluation

Dataset	Nodes	Edges	No. of features
Twitch PTBR	1.9K	31K	2.5K
Twitch ENGB	7.1K	35K	2.5K
Twitch RU	4.3K	37K	2.5K
Twitch ES	4.6K	59K	2.5K
Twitch FR	6.5K	112K	2.5K
Deezer RO	41K	125K	84
Twitch DE	9.4K	153K	2.5K
Deezer HU	47K	222K	84
Bitcoin	203K	234K	165
Deezer HR	54K	498K	84
Reddit	230K	5.9M	602
Synthetic	930M	1.3B	64
Papers100M	110M	1.6B	128

6.4 Evaluation Methodology

We evaluate *GCNSplit*'s efficiency, scalability, and partitioning quality in various scenarios. Before presenting the results, we first describe our experimental setup and evaluation methodology in this section. We present the datasets and baseline algorithms we use for our experiments and the partitioning quality evaluation metrics. The configuration parameters used during model training can be found in the Appendix [144].

6.4.1 Environment Setup

We trained our models using an on-premises physical machine consisting of a `Nvidia RTX 2070 Super GPU` with `8GB` of internal memory. We served the models using a machine comprising of an `AMD Ryzen Threadripper 2920X 12-Core` processor with `128GB RAM`. All modules of the system are implemented using `Python 3.8`. We used `Pandas 1.0.2` [145] to read the CSV files containing graph data and `NumPy 1.16.2` [146] to leverage its support for working with arrays containing the node's features. We used `PyTorch 1.4` [147].

6.4.2 Datasets

Table 6.1 shows the characteristics of the datasets we use to evaluate *GCNSplit*, ordered by their number of edges. *GCNSplit* expects an input graph with associated features, as well as timestamps. Of those publicly available, we have selected graphs from different domains to evaluate partitioning quality and of various sizes to verify that the quality of partitioning does not degrade for large graphs. We have also chosen graphs with different feature sets to study how the state size changes.

Twitch: The Twitch graph dataset [148] represents a user-to-user network of the platform where streamers broadcast their activities live. The dataset consists of six different networks based on the user's language. The node attributes consist of a user's location, streaming habits, and activity information and all networks have the same feature set representation. This dataset is interesting because it has a usually large number of features. Further, we can use one of the networks for training and the rest to evaluate the generalization performance of *GCNSplit*.

Deezer: The Deezer graph dataset [149] represents the user's friendship network on a music streaming service. The dataset contains three different networks based on the user's country and the features consist of users' preferred music genres. This dataset can also be used to evaluate the generalization of *GCNSplit*'s models by selecting one network for training. As it has a much smaller set of features than Twitch (84 vs. 2.5K), it can provide insights on the effect of features on the generalization quality.

Bitcoin: The Bitcoin graph dataset [143] represents Bitcoin transactions mapped to real entities. A node in the graph represents a transaction and edges represent the flow of Bitcoins between these transactions. The features of the node contain information, such as the time-stamp associated with each transaction, the number of inputs/outputs and transaction fees etc.

Reddit: The Reddit graph dataset [65] represents a post-to-post network of Reddit. Reddit is a social media platform where users post and comment on topics of their interest. Edges represent comments between posts and node features consist of the post title and the number of comments.

Papers100M: The Papers100M graph dataset [150] represents the citation network between computer science arXiv papers. Each node has a feature vector of 128 dimensions created by embedding the paper's title and abstract. We order the edges of the dataset using their publication year to have a time-based set of edges.

Synthetic: To evaluate the performance of *GCNSplit* in a more challenging scenario, we generate a large random graph with 1.3B edges and 930M nodes. The graph has 64 random synthetic features and we use a subset of 10K edges for model training. We use this graph to demonstrate that *GCNSplit*'s throughput and state

size is independent of the graph size. Materializing its partitioning mapping exceeds the available memory of the machine we use for evaluation. Further, we use this synthetic graph to empirically confirm that *GCNSplit* will perform like hash partitioning in the worst case (cf. Section 6.2.4). The randomly generated features do not capture any useful information about the graph structure or vertex similarity, thus, they present an adversarial example for *GCNSplit*.

6.4.3 Baseline partitioning methods

We compare *GCNSplit* with two baseline methods that prioritize different quality metrics. A hash-based stateless method that favors load balancing and a stateful method that optimizes cuts. Our goal with *GCNSplit* is to strike a balance between the two, providing both well-balanced and high-quality partitions.

Hash Partitioning: The first baseline we consider is the stateless hash partitioning method. It uses a consistent hashing function to map items, vertices or edges, with distinct identifiers to partitions. Hash partitioning leads to highly balanced partitions but it does not optimize for the number of cuts.

HDRF: The second baseline we consider is the HDRF [21] state-of-the-art edge-centric streaming graph partitioning method. HDRF yields the best results amongst similar methods [24] and performs especially well with power-law graphs. HDRF prioritizes splitting high degree nodes first, leading to low vertex-cuts. HDRF uses degree information of the vertices and the partition information of the already assigned vertices to make partitioning decisions. Information regarding already seen vertices has to be stored in memory and constantly updated. HDRF is parametrized by λ and ϵ, where λ controls the extent of partition imbalance in the score computation, while ϵ is a small positive constant that ensures a positive denominator in the score formula. We set $\lambda = 1$ and $\epsilon = 10^{-5}$. We use the original implementation of HDRF available in the public repository [2].

6.4.4 Quality Metrics

We evaluate the quality of partitioning methods with the following metrics. To evaluate load balance, we use the **normalized load**(ρ) on the highest loaded partition. It is defined as

$$\rho = \frac{\mathit{load\ on\ the\ highest\ loaded\ partition}}{n * k^{-1}} \tag{6.4}$$

[2]https://github.com/fabiopetroni/VGP

where n is the number of edges in the graph and k is the number of partitions. $\rho \approx 1$ indicates that the load is well-distributed across the partitions.

To evaluate partitioning quality, we use the **replication factor**(σ), which indicates how many vertex copies are created by the partitioning algorithm. It indicates the resulting vertex-cuts. The lower the σ, the better the partitioning quality. It is defined as

$$\sigma = \frac{\text{Total number of vertex copies}}{\text{Total number of vertices}} \tag{6.5}$$

6.4.5 Model Parameters

In our experiments, we trained models using different configurations for each dataset. Appendix [144] contains model's training configurations. The training sets are created based on dataset's granularity that includes, years, epochs and time-stamps etc. For example, the Reddit dataset models were trained on the first 10,000 edges based on timestamp order and Bitcoin models were trained on the first 9,164 edges based on the first dataset epoch. All models used Adam optimizer [151] with a learning rate equal to 0.0001. The embedding size is 64. The number of layers of the embedding network is 2. The number of layers in the partitioning network is 3 and the number of neurons in a hidden layer of the partitioning network is 64.

In most of the experiments, the maximum load parameter is set to 1.01, keeping the ratio between the number of edges in the highest loaded partition and the *ideal* ($\frac{n}{k}$) partition always below 1.01. This setting is different for experiments regarding the trade-off between the maximum load value and the replication factor. Lastly, the HDRF algorithm has the λ parameter set to 1 and the $\epsilon = 10^{-5}$, based on its default configurations.

6.5 Evaluation Results

We organize the evaluation section in the following three parts. First, we evaluate **partitioning quality** and show that *GCNSplit* is on par with HDRF in terms of replication factor, while it maintains well-balanced partitions (Sec. 6.5.1.). Second, we present **performance** results that demonstrate how *GCNSplit* provides high-throughput partitioning and scales with the number of parallel processes while having small and constant state requirements (Sec. 6.5.2.). Third, we evaluate the **model generalization** and demonstrate that *GCNSplit* not only produces high-quality partitions for unseen graphs but also outperforms HDRF in multiple instances (Sec. 6.5.3). Last, we provide **comparison with GAP** result where we compare *GCNSplit* partitioning quality with GAP (Sec. 6.5.4). For convenience,

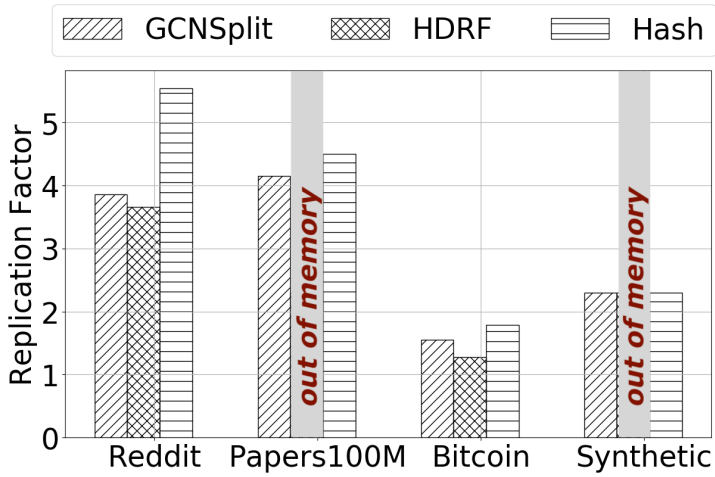


Figure 6.4 – Partitioning quality of GCNSplit and baselines. Load limit = 1.01 , k = 6.

we refer to the unsupervised model as *GCNSplit* and to the supervised model as *GCNSplit*-s. In several of the experiments we omitted the results of *GCNSplit*-s since the observed performance was equal or sometimes worse than that of *GCNSplit*.

6.5.1 Partitioning Quality

We evaluate *GCNSplit*'s partitioning quality in terms of replication factor and load balance and compare it with HDRF, hash partitioning, and a baseline approach that uses GAP's loss function. We also study how the replication factor is affected by the maximum load constraint, the number of edges in the graph stream, and the number of partitions. Finally, we evaluate how the heuristics we introduce in Section 6.2.3 affect the replication factor.

Replication factor. We use HDRF, Hash, and *GCNSplit* to partition the Reddit, Papers100M, Bitcoin, and synthetic graphs into 6 partitions. We set the maximum load limit for *GCNSplit* to 1.01, effectively forcing the creation of well-balanced partitions. We measure the replication factor of the resulting partitions and plot the results in Figure 6.4. *GCNSplit* outperforms Hash on all real graphs and exhibits marginally higher σ than HDRF on the Reddit and Bitcoin graphs. The HDRF baseline ran out of memory before completing the partitioning of Papers100M and Synthetic graphs. The synthetic graph represents a worst-case scenario for *GCNSplit*, having random structure and features, yet, its worst-case performance falls back to that of hash partitioning (cf. Section 6.2.4).

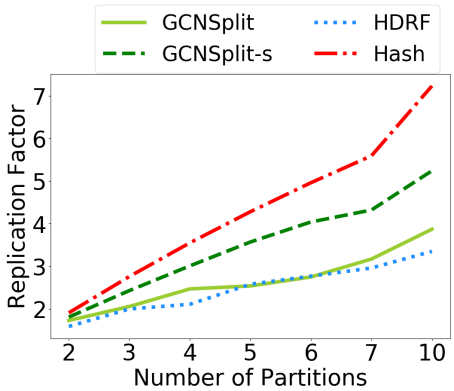


Figure 6.5 – Reddit ($p = 16$).

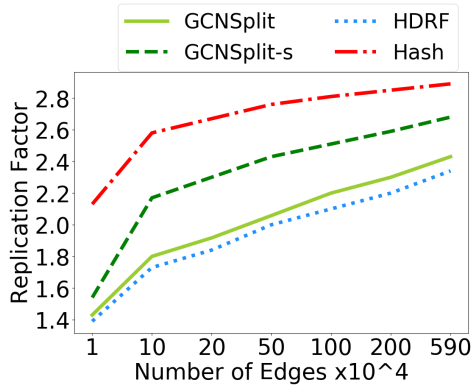


Figure 6.6 – Reddit ($k = 3$).

Effect of the number of partitions. Next, we evaluate the sensitivity of σ to the number of partitions. We partition the Reddit graph into an increasing number of partitions, using *GCNSplit*, HDRF, and Hash, and we measure the replication factor of the resulting partitions. Figure 6.5 shows that σ increases with the number of partitions across approaches. Hash produces the highest σ value, while HDRF and *GCNSplit* (unsupervised) perform similarly with the best cut ratio and lowest σ. *GCNSplit*-s, which is trained using HDRF, has a slightly greater σ compared to *GCNSplit*.

Effect of the number of edges. We now study how the replication factor changes over time, as the streaming algorithms partition continuously arriving edges from the graph stream. We stream the Reddit graph in timestamp order and fix the number of partitions to $k = 3$. We measure the σ after certain intervals based on the x-axis ticks as shown in Figure 6.6. As expected, σ increases with the number of edges for all algorithms. However, once again, *GCNSplit* performs as well as HDRF, producing the lowest number of cuts.

Load Balance. In our experiments so far, we set the maximum load limit constraint to 1.01. We now study how much further we can improve *GCNSplit*'s replication factor by increasing the maximum load constraint to control the normalized load, ρ. Hash partitioning exhibits $\rho \approx 1$, while HDRF achieved $1 \leq \rho \leq 1.0314$, in our experiments, depending on the dataset and number of partitions. For this experiment, we use the Reddit graph and we set the number of partitions to $k = 3$ for *GCNSplit* and *GCNSplit*-s. We measure the effect of changing the maximum load limit constraint on the replication factor σ and plot the results in Figure 6.7. Overall, σ decreases with an increasing maximum load limit. Further, we see that the effect is greater for the unsupervised mode of *GCNSplit*.

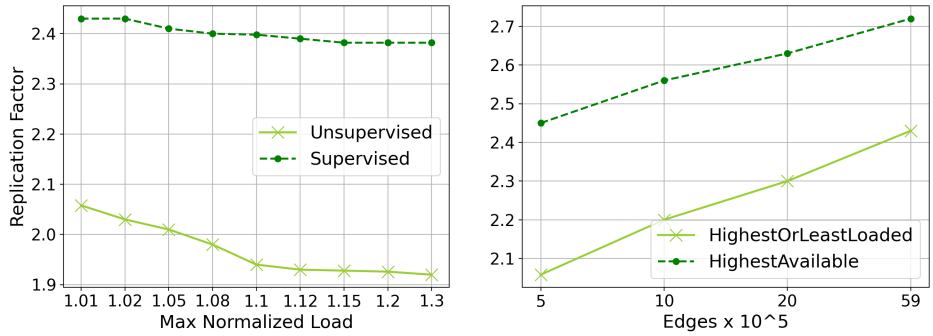


Figure 6.7 – Effect of the maximum load constraint (left) and the heuristic (right) on the replication factor. Reddit unsup., $k = 3$.

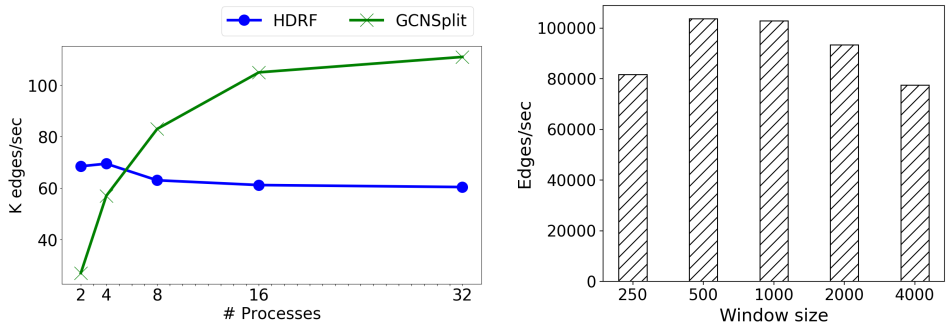
Effect of heuristics. Our experiments so far have used the *HighestOrLeastLoaded* heuristic. In this section, we evaluate how this choice compares with the *HighestAvailable* heuristic and how both methods behave on different graphs. Figure 6.7 plots the replication factor on the Reddit graph in a streaming scenario where edges arrive continuously and are assigned to 3 partitions. This experiment uses *GCNSplit* in unsupervised mode. *HighestOrLeastLoaded* performs consistently better than *HighestAvailable* throughout the duration of the experiment. We obtained similar results for 6 partitions on the Twitch graph and the supervised mode of *GCNSplit*.

6.5.2 Partitioning Performance

We evaluate performance in terms of throughput and state size and compare *GCNSplit* with HDRF using graphs from Table 6.1.

Throughput. Any efficient streaming graph partitioning algorithm needs to be capable of making decisions *online*, as edges arrive at its input. A traditional stateful algorithm, like HDRF, makes decisions by performing state lookups and computing a heuristic to rank partitions. For *GCNSplit*, however, the partitioning decision relies on model inference and entails producing graph embeddings for both edge endpoints before computing the heuristics. Nonetheless, as *GCNSplit*'s state is *immutable*, we can leverage data parallelism to increase its throughput by adding partitioner processes.

For this experiment, we partition the 0.2B edges of Papers100M dataset into $k = 6$ partitions with HDRF and *GCNSplit* and measure the number of edges processed per second. The reason to select 0.2B edges was that HDRF runs out of memory with more edges. We set *GCNSplit*'s input window size to 1K edges and increase



(a) Scalability. Input: Papers100M

(b) Effect of window size. Input: Reddit

Figure 6.8 – Partitioning throughput on with $k = 6$.

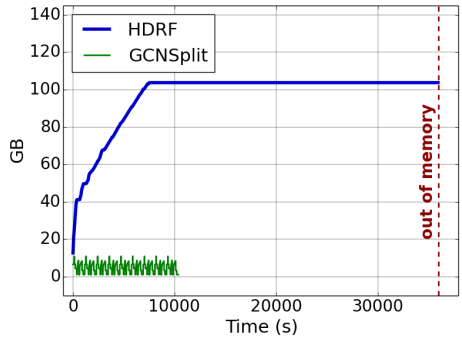
Table 6.2 – Partitioning state size (k=6) and training times.

Dataset	*GCNSplit* state	HDRF state	Training (min)
Twitch DE	1.6MB	4.1MB	22
Deezer RO	126KB	5.4MB	38
Bitcoin	166KB	19MB	10
Reddit	385KB	47MB	36
Papers100M	147KB	>116GB	233
Synthetic	115KB	>116GB	13

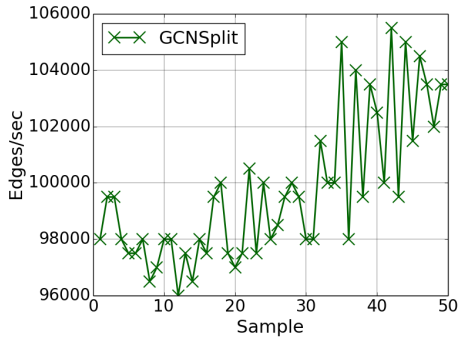
the number of parallel partitioner processes from 2 up to 32. Figure 6.8a shows how *GCNSplit* scales with the number of parallel partitioners while outperforming HDRF with 8 or more processes. *GCNSplit*'s throughput peaks at 111K edges/s with 32 processes, while HDRF cannot scale to more than 70K edges/s.

Effect of window size. The throughput of streaming applications largely depends on the input batch size of the streams they process. We examined *GCNSplit*'s throughput in respect to window size, using the Reddit graph with $k = 6$ partitions and $p = 16$ processes. In Figure 6.8b it is observed that in this setting throughput caps at 1K edge windows, which we adopt as a constant in all experiments.

State size. We compare *GCNSplit* and HDRF model state size for various graphs. Table 6.2 shows the results for 6 partitions. We only include numbers for *GCNSplit*'s unsupervised model since the supervised models were of similar size. As expected, HDRF accumulates much larger state than *GCNSplit*, proportional to the size of the graphs. *GCNSplit*, on the other hand, has orders of magnitude smaller state requirements, requiring just 115KB for Synthetic(1.3BM edges) and 147KB for Papers100M(1.6BM edges) to store the models corresponding to the biggest graphs.

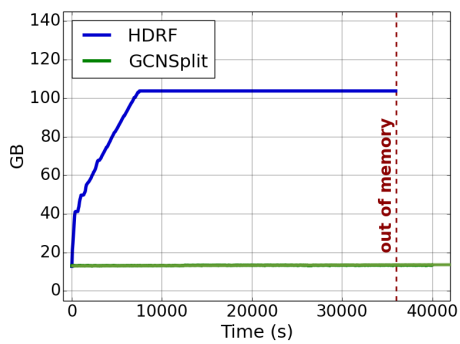


(a) Memory usage.

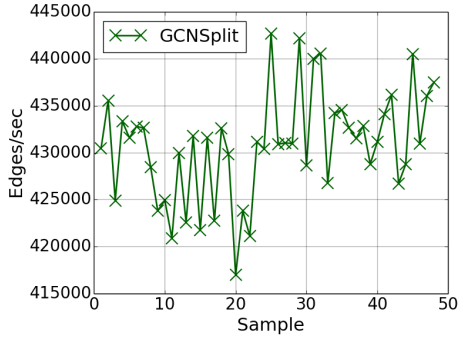


(b) Throughput.

Figure 6.9 – Partitioning performance for HDRF and *GCNSplit* on the large synthetic graph.



(a) Memory usage.



(b) Throughput.

Figure 6.10 – Partitioning performance for HDRF and *GCNSplit* on the Papers100M graph.

Figure 6.9a and Figure 6.10a plot the system memory usage over time while using 32 processes to partition the synthetic graph and the Papers100M graph, respectively. *GCNSplit* successfully partitions the entire synthetic graph using up to 11GB of memory, while maintaining a throughput of 100K edges/s, as shown in Figure 6.9b. Whereas, HDRF's state requirements outgrow the available memory during ingestion and the partitioning fails before completion for both datasets. Figure 6.10b shows that *GCNSplit* partitions the Papers100M dataset with a throughput of 430K edges/s, using up to 14GB of memory. Note that Figure 6.10a

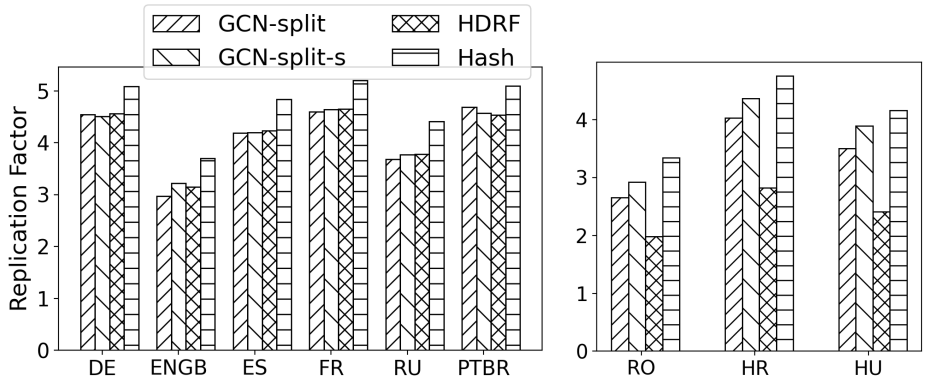
includes disk access time along with the partitioning time thus, the time to partition Papers100M exceeds 40K sec. Whereas Figure 6.10b does not include disk accesses, it is only the throughput of the partitioning processes. Similarly, for the synthetic graph, Figure 6.9a and Figure 6.10a do not include disk accesses because the synthetic graph is being generated on the fly. It is also noteworthy to attribute the effect of the graph structure on the partitioning throughput. As detailed in Table 6.1 the Papers100M dataset contains 9 orders of magnitude fewer nodes compared to the highly dense synthetic graph. Combined with the fact that Papers100M is streamed in time windows, the probability of encountering duplicate graph nodes in the same window is much higher than that of the synthetic graph, therefore, leading to less unique embedding lookups per batch and higher throughput. In a matter of fact, the lower computational cost in the case of Papers100M (Figure 6.8a) even amortizes the read disk access involved in this scenario, contrary to the synthetic graph which is generated on the fly. Evidently, the synthetic graph prescribes a worst-case throughput performance scenario due to its sparsity and randomized edge ordering. Finally, to compare *GCNSplit* against HDRF under equal state usage we conducted an additional experiment on the Reddit graph with $k = 6$. HDRF's memory was throttled and reset at $\sim$ 400KB which led to a significant increase in σ from $\sim$ 3.66 to 4.93 (Fig 6.4), and reduced quality compared to *GCNSplit*.

Training time. *GCNSplit* performs training offline on a snapshot of the streaming graph and re-training is not required when partitioning graphs with the same feature set (c.f. Section 6.2.2). Table 6.2 reports the training times for the models we use in this work. We found that the training performance primarily depends on the snapshot size rather than the resulting model size. Training the largest model (Reddit) takes 36 minutes on a snapshot of 16K nodes. Training the Papers100M model takes 6.5$\times$ longer for a 62.5$\times$ bigger snapshot with 1M nodes.

6.5.3 Generalization to Unseen Graphs

So far we have shown *GCNSplit*'s capability of producing high-quality partitions for unseen nodes of a streaming graph, using a fixed-size model. Here, we evaluate *GCNSplit*'s ability to effectively partition completely unseen graphs. Specifically, we train *GCNSplit* on a subset of a graph and then use its model to partition a different graph stream, albeit with a common feature set.

For this experiment, we use the Twitch and Deezer networks and set $k = 6$. We train the models on the complete Twitch-DE graph and 10K edges of the Deezer-RO graph and use the rest of the networks for inference. Figure 6.11 plots the replication factor of *GCNSplit* in supervised and unsupervised mode, alongside that of HDRF and Hash. *GCNSplit* generalizes well to unseen graphs having matching feature sets and has a low replication factor across experiments. More importantly, *GCNSplit*



(a) Twitch (k = 6).

(b) Deezer (k = 6).

Figure 6.11 – Generalization: partitioning quality on unseen graphs.

outperforms HDRF for all but one instance of the Twitch network. A comparison across graphs also provides an insight into the effect of the feature set, suggesting that the richer the features the better the partitioning quality. Recall that Twitch contains 2.5K features, while Deezer exhibits a mere 83 features per node.

Our observations showcase an important finding. *GCNSplit* is capable of producing high-quality partitions for entirely unseen graphs whose feature set matches that of the training graph. Whereas, the state of the art algorithms accumulate state tailored to a single graph and cannot be re-used in other partitioning instances.

6.5.4 Comparison with GAP's loss function.

We now compare *GCNSplit*'s partitioning quality to a baseline that represents an adaptation of GAP to the streaming setting. As vanilla GAP is an offline vertex partitioning method, we cannot directly use it to partition edge streams. Instead, we incorporate its loss function into our framework and compare its load balance to that of *GCNSplit*. Figure 6.12 shows the results for the Twitch (T-*), Deezer (D-*), and Reddit graphs. The GAP baselines exhibit significant imbalance, up to 226% and 205% higher than *GCNSplit* for unsupervised (GAP-base) and supervised (GAP-base-s) models, respectively

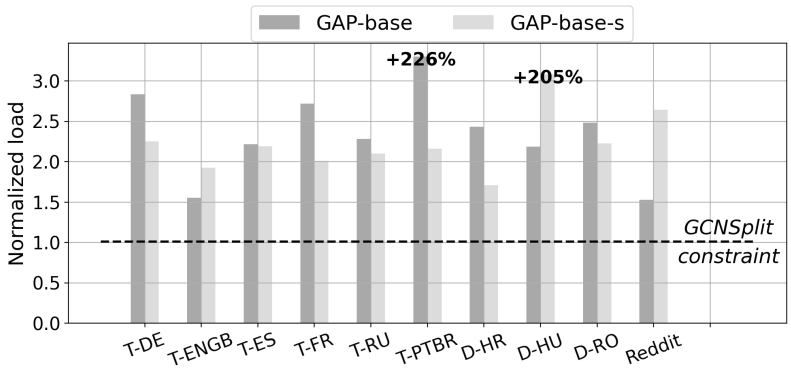


Figure 6.12 – Maximum normalized load of the GAP-baseline approach for supervised and unsupervised models. In all cases, the baseline models produce highly unbalanced partitions, greatly exceeding *GCNSplit*'s 1.01 load constraint.

6.6 Discussion and Limitations

The partitioning quality of *GCNSplit* shows good performance results compared to the state-of-the-art partitioning methods. It uses node features and neighborhood information to create node embeddings that are used for partitioning. Due to the window based mechanism, only neighbors that are part of the window are considered during partitioning.

The partitioning quality of *GCNSplit* depends on the training sample size. For large datasets, i.e., Papers100M with $\approx$ 1B edges, the partitioning model trained on a sample containing only $\approx$ 0.6M edges generates partitioning quality that degrades with time, but still better than Hashing. Online training with periodic updates will improve the partitioning quality.

Since *GCNSplit* uses GraphSAGE for embedding, there are some limitations that exist due to the aggregation technique of GraphSAGE. GraphSAGE cannot work well for featureless graphs. In the worse case when random features are added for a featureless graph *GCNSplit* partitioning quality is similar to Hashing. Also, GraphSAGE cannot distinguish between certain structures of graphs. For example, if two nodes in the graph have the same neighbourhood structure, GraphSAGE would end up generating the same embeddings for these nodes [152].

6.7 Related Work

ML-driven graph partitioning is a new exciting research territory. To the best of our knowledge, *GCNSplit* is the first attempt to leverage the power of inductive graph representation learning for online partitioning. Other than the offline approaches outlined in the GAP papers [63, 64], we are not aware of closely related work.

Within the broader context of the recent ML-enabled data management and systems research, our work is similar in essence to learned indexes and data structures [153, 154, 155]. *GCNSplit* relies on learned characteristics of the input data (the graph) to improve the performance of a data management task, which in our case is partitioning rather than search, as in existing work.

Finally, we summarize work from two adjacent areas we believe will be instrumental in future research: (1) streaming graph partitioning algorithms that can serve as ground truth for supervised learning, and (2) graph representation learning approaches that can serve as alternatives to GraphSAGE.

Streaming graph partitioning. The performance of streaming graph partitioning algorithms has recently been studied in experimental surveys [24, 156]. Among vertex partitioning methods, Linear Deterministic Greedy partitioning [2] and Fennel [23] could aid the supervised learning phase of *GCNSplit*. The first approach relies on a greedy heuristic that assigns a vertex to the partition containing most of its neighbors while respecting certain capacity constraints. The latter interpolates between maximizing the co-location of neighboring vertices and minimizing that of non-neighbors. Among edge partitioning methods, while HDRF performs best in a recent study [24], we believe that training with Degree Based Hashing (DBH) [22] is also worth exploring. DBH is a hybrid approach that employs hashing for partitioning while also prioritizing cutting high-degree vertices.

Other graph representation learning approaches. Transductive methods [142, 141, 157, 158, 159] are of little use in the streaming context, as they can only generate representations for nodes used during the training phase. Nevertheless, they could be leveraged in static graph scenarios and when node features are absent. On the other hand, inductive methods are more suitable for a streaming setting and numerous methods exist that could replace GraphSAGE in *GCNSplit*. The algorithms differ in the way they aggregate and sample neighbors, as well as in their approach to generating and combining representations. With regards to the problem of partitioning, we believe that graph attention networks [160] and position-aware graph neural networks (P-GNN) [152] are particularly interesting. The first approach utilizes the *attention strategy* to conduct neighborhood aggregation. Trainable parameters are applied to a node's neighbors to weight contributions from neighbors differently. In P-GNNs, nodes are represented using their distances to an *anchor set*, which consists of randomly chosen one or more nodes serving as point of reference for other nodes.

6.8 Acknowledgements

I would like to thank our master student Michał Zwolak for helping us in implementing the graph neural network based partitioner. He worked very hard to build *GCNSplit*. I would also like to thank Vasiliki Kalavri and Paris Carbone for their co-supervision and great ideas to build *GCNSplit*. Their constant feedback helped us making this work successful. Lastly, I would like to thank Sonia-Florina Horchidan for her efforts in the experiment revision.

6.9 Summary

In this chapter, we address the challenge of providing a *CH3a-Scalable Graph Partitioning* and *CH3b-Partitioning Graph Streams Using Bounded State* mentioned in Chapter 1 by providing *C4*-a novel streaming graph partitioning algorithm for partitioning unbounded streams. We presented and evaluated *GCNSplit*, an ML-driven streaming graph partitioning method that overcomes the problem of increasing state size for unbounded streams without sacrificing quality. *GCNSplit* bases its partitioning decisions on the attributes of the nodes. Thus, in real-world applications, the feature set should be carefully analysed in order to increase the quality of the data, and consequently, the quality of the partitioning. Moreover, thanks to the capabilities of GCNs, the model incorporates the structural context of every node as well. As a result, it benefits from the information about the connections between the items. Hence, it is important to produce a representative sample of the whole network in the training dataset.

Our results show that *GCNSplit* exhibits partitioning quality as good as state-of-the-art streaming partitioning algorithms, while it maintains well-balanced partitions. Further, it sustains high throughput and scales linearly to the number of parallel processes, while having small and constant state requirements. Finally, *GCNSplit* generalizes well and can effectively partition unseen graphs, as long as they share feature set characteristics with the samples used for model training.

CHAPTER

7

Conclusion

"Shout-out to all PhD students dissertating during the pandemic
and the typos that still survived after multiple proof readings"
— Me.

Throughout the course of this thesis, we proposed graph partitioning techniques to improve the scalability and performance of graph-based and machine learning applications. First, we developed a graph partitioning algorithm to partition spatial time series data for large-scale time series forecasting. Second, in order to study and evaluate the use of streaming graph processing algorithms for analysis of a complex large-scale distributed system, we applied streaming graph analytics for a practical use-case of road traffic congestion detection and mitigation at a large scale with low latency. Third, we surveyed, evaluated and compared state-of-the-art streaming graph partitioning algorithms and highlighted their shortcomings. In the end, we proposed a machine learning-based scalable streaming graph partitioning algorithm that uses a small and constant in-memory state (bounded state) to partition (possibly unbounded) graph streams.

7.1 Summary of Results

In the first part of this thesis presented in Chapter 3, we address the scalability in training deep learning models in a real-life task of large-scale road traffic prediction using real-life large data sets generated by traffic sensors deployed in Stockholm and Gothenburg, Sweden. We developed a graph-based partitioning technique that resulted in enabling parallelism for both training and prediction and improved the scalability and performance of deep learning models. Our partitioning-based models using LSTMs take 2x if run sequentially, and 12x, if run in parallel, lower training time, and 20x lower prediction time compared to the unpartitioned model

of the entire road infrastructure. The partitioning-based models take 100x lower total sequential training time compared to single sensor models, i.e., one model per sensor. Furthermore, the partitioning-based models have 2x lower prediction error (RMSE) compared to both the single sensor models and the entire road model. Since the sensors network topology remained constant during the time frame of our dataset, we assume that the sensor network is static. Although, in real-life sensors are added or removed in the network over time.

In our next work presented in Chapter 4, we provide a practical and useful real-life application of streaming graph analytics on graph steams. We addressed the problem of traffic congestion detection and mitigation using real-life data collected from a region in one of the largest metropolis cities of China. We developed an end-to-end framework built on top of a modern stream processing system, i.e., Apache Flink. With the proposed framework, we successfully detected traffic jams in real-time and deploy new traffic light policies which result in 27% less travel time at the best and 8% less travel time on average compared to the travel time with default traffic light policies. Our scalability results show that our system is able to handle high-intensity streaming data collected from 900 sensors every second with a throughput of 57K records/sec at best. The traffic jam detection on streams is done using single-pass connected components algorithm [97], which works only for edge additions in a graph stream. Dynamic graph stream processing algorithms with edge deletions are beyond the scope of this work.

Next, we present our work in Chapter 5 on streaming graph partitioning algorithms. We have studied streaming graph partitioning algorithms and we have empirically compared them using an evaluation comparison framework that we developed based on Apache Flink [30]. We have evaluated the partitioning quality and performance of the partitioning algorithms and we have measured the effect of partitioning on the performance of iterative and streaming applications. Our experimental results show that in terms of performance, Hash shows 2x higher throughput than the second-best partitioning method and can generate up to 90% edge-cuts keeping no state in memory. Whereas, all studied algorithms are stateful, i.e., they base their partitioning decision using an in-memory state that contains information, such as current vertex assignment, partition capacities, or vertex degree distributions. Stateful algorithms produce 20% edges-cuts and very low vertex cuts compared to Hash and they require the state to be shared among parallel partitioning instances, for a global view of state shared across parallel instances. Due to their shared state requirements, they are *unsuitable* for continuous processing of unbounded streams. We note the advantage of low-cut partitioning methods for iterative applications, while low-cost (cost refers to the partitioning latency) partitioning mechanisms seem preferable for streaming applications. We highlight the need for developing new, scalable online partitioning algorithms, with relaxed constraints on the graph properties and fewer in-memory state requirements.

We believe that the development of such algorithms is crucial for making graph partitioning practical for applications ingesting continuous streams on top of modern distributed streaming engines. The streaming applications are based on the semi-streaming model [97] and only edge insertions are considered in this work. We do not consider edge deletions in this work, they are beyond the scope of this work and a subject to future work.

Finally, we propose and evaluate a novel, graph convolutional network (GCN)-based graph partitioning algorithm, *GCNSplit*, presented in Chapter 6. *GCNSplit* is an ML-driven streaming graph partitioning method that overcomes the problem of increasing state size for unbounded streams without sacrificing partitioning quality. *GCNSplit* learns the structure of the graph by using node features and neighboring nodes' features to make partition assignment decisions and benefits from recent advances in GCNs. It supports both supervised and unsupervised training and generalizes to unseen nodes and graphs, as long as they bear common features. Our results demonstrate *GCNSplit* generates high-quality and well-balanced partitions with small and constant state requirements. It exhibits a replication factor that matches that of the state-of-the-art HRDF algorithm while storing three orders of magnitude smaller partitioning state. At the same time *GCNSplit* provides a high-throughput of 400K edges/s compared to HDRF that provides a throughput of 70K edges/s. We also compared *GCNSplit* to a GCN-based baseline GAP. GAP exhibited up to 226% higher imbalance compared to *GCNSplit*. Further, owing to the power of GCNs, we show that *GCNSplit* can efficiently and effectively partition entirely unseen graphs with the same feature set as the trained model. When applied on unseen graphs with a rich feature set, *GCNSplit* outperforms HDRF in multiple cases. Limitations of *GCNSplit* are that it cannot work for featureless graphs and a dynamic number of partitions.

We conclude from our aforementioned results that graph streams and graph partitioning have helped improve the performance and scalability of spatial time series forecasting and analytics, and streaming graph applications. Also, machine learning-based graph partitioning methods are useful for efficiently processing possibly unbounded streaming graphs. Our solutions have addressed the challenges mentioned in Chapter 1, Section 1.4.

7.2 Generalization to Other Application Areas and Graph Streams

In this section, we discuss how our proposed solutions related to large scale time series analysis and forecasting can be applied to other application areas. Also, how our streaming graph partitioning solution can be generalised to partition unseen graph streams with a similar feature set.

Our proposed graph partitioning techniques to scale deep learning-based time series forecasting models albeit been applied to road traffic use-case, in general, can

be applied to other application domains. The graph representation and partitioning technique we proposed can be applied to other complex systems containing sensors generating correlated spatial time series. One example is the air traffic control systems containing geographically distributed sensor-based infrastructure [161]. The sensors monitoring the air traffic produce a correlated stream of data. Another example is a spatially distributed fitness tracker network, where closely located users generate correlated time series of health activities [49].

Streaming graph analytics is a general data processing domain that can be applied to analyse graph streams in a variety of applications. Graph streams comprise of continuous timestamped events modelled either in the form of a stream of edges or vertices with associated adjacency lists [11, 12]. The events can be user interactions in social networks, online financial transactions, driver and user locations in ride-sharing services. One of the application areas presented in Chapter 4 is traffic congestion detection in a road traffic stream. Other prominent use-cases include prediction of traffic and demand for a ridesharing service and management of networks online in software-defined network controllers.

Our proposed graph convolutional network-based streaming graph partitioning algorithm, i.e., *GCNSplit* (Chapter 6) is applicable to all timestamped graph datasets containing node embeddings. *GCNSplit* generalizes to partition unseen graphs with similar structure and matching feature set dimensions of the training graph. The generalisation capability of graph convolutional networks makes them a powerful tool for creating a general partitioner which is capable of partitioning completely out of domain graphs without re-training.

7.3 Social and Environmental Aspects

We consider the following social and environmental aspects in our research work.

- **Privacy.** Our work does not use any kind of private or sensitive information in the datasets. The datasets used in the traffic analytics work were anonymised by the domain experts who collected them and contained no information that can harm society. Also, the graph datasets that we used are publicly available and are anonymised by the providers. In some cases the data providers mention the privacy preserving techniques used to hide sensitive information and in other cases they do not. Therefore, we cannot say much about the effectiveness of the privacy preserving techniques that have been applied to the data used in this thesis.

- **Environmental Sustainability.** We aim to contribute towards the advancement of stream processing systems. Stream processing systems contribute to low-latency mission-critical applications that can also save compute and

memory-based resources. The low compute and memory requirements exist because generally in a stream processing model the data is visited once and only the aggregates are stored in memory compared to other offline approaches where complete data is pre-processed in memory. Our work presented in Chapter 3 and 4 makes use of graph partitioning and streaming graph analytics to efficiently forecast, detect and mitigate traffic jams. Traffic congestion reduction has a great amount of positive impact because nowadays with the increasing number of vehicles used to commute every day, traffic congestion has become a common sight. It is important to monitor traffic flows to prevent congestion to avoid a multitude of problems. Some of these problems include: increase in fuel consumption and pollution [88], decrease in economy [89] and traffic safety that is caused by a speed variance between cars in the congested region compared to cars moving freely [90], and harmful effects on the mental and physical health of people [91, 92].

7.4 Future Work

In future work, we would like to address the limitations of our work by extending our streaming graph partitioning algorithms and applications to work for dynamic graph streams, which not only include edge additions but also edge deletions. Another dimension is towards the improvement of graph stream processing frameworks. Our proposed graph convolutional network-based partitioning can be exploited to develop distributed graph query engines and databases with stream ingestion. Furthermore, the quality of our techniques can be improved via feature analysis or online learning methods to allow the periodic update of the model to learn recent trends in the data stream when significant changes are detected (new features and structural properties). An interesting direction is exploring how to dynamically change the number of partitions over time. Elasticity could be achieved by training with a large number of *virtual partitions*. An additional layer could then be used to map physical partitions to virtual ones without updating the model.

Our work on machine learning-based partitioning with the potential to partition unseen graphs gives us an interesting future work directed towards a perfect graph partitioner. In a hypothetical world, we think of an oracle that suggests the best partitioning strategy to partitioner for any input graph dataset. Some useful characteristics of this oracle can be: 1) Processing input graphs from various application domains. 2) Finding similarities between the recent input graph features and the graphs partitioned in the past. 2) Suggesting the best partitioning strategy based on feature set similarity measure of the input graph. 3) Guiding the partitioner with a different partitioning strategy in the case of a drift in the input graph stream or a drop in the partitioning quality. 4) Keeping track of the most used partitioning strategies and least used ones and give priority to the most frequently used ones. 5)

Learn from recent history and predict a partitioning strategy. 6) Take into account application performance requirements, such as low latency and high throughout etc., and update the partitioning strategy based on the application's requirement.

The main expectations from a perfect partitioner itself are of high partitioning quality, adaptability in terms of support for dynamic number of partitions, low computational and memory cost, scalability and ability to partition possibly unbounded streams.

Bibliography

[1] Z. Abbas, J. R. Ivarsson, A. Al-Shishtawy, and V. Vlassov, "Scaling deep learning models for large spatial time-series forecasting," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 1587–1594.

[2] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 1222–1230. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339722

[3] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *arXiv preprint arXiv:1204.6078*, 2012.

[4] Z. Fu, Y. Xian, R. Gao, J. Zhao, Q. Huang, Y. Ge, S. Xu, S. Geng, C. Shah, Y. Zhang, and G. de Melo, "Fairness-aware explainable recommendation over knowledge graphs," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 69–78. [Online]. Available: https://doi.org/10.1145/3397271.3401051

[5] J. Li, K. Zhang, and Z.-P. Meng, "Vismate: Interactive visual analysis of station-based observation data on climate changes," in *2014 IEEE Conference on Visual Analytics Science and Technology (VAST)*. IEEE, 2014, pp. 133–142.

[6] A. Greiner, W. Semmler, and G. Gong, *The forces of economic growth: A time series perspective*. Princeton University Press, 2016.

[7] D. Laney, "3d data management: Controlling data volume, velocity and variety," *META Group Research Note*, vol. 6, p. 70, 2001.

[8] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, 2015.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[10] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 439–455.

[11] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz *et al.*, "The future is big graphs! a community view on graph processing systems," *arXiv preprint arXiv:2012.06171*, 2020.

[12] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2651–2658.

[13] P. Zhang, Y. Huang, S. Shekhar, and V. Kumar, "Correlation analysis of spatial time series datasets: A filter-and-refine approach," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2003, pp. 532–544.

[14] K. J. Friston, P. Jezzard, and R. Turner, "Analysis of functional mri time-series," *Human brain mapping*, vol. 1, no. 2, pp. 153–171, 1994.

[15] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," *arXiv preprint arXiv:1707.01926*, 2017.

[16] Y. Li and J. M. F. Moura, "Forecaster: A graph transformer for forecasting spatial and time-dependent data," in *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, ser. Frontiers in Artificial Intelligence and Applications, G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds., vol. 325. IOS Press, 2020, pp. 1293–1300. [Online]. Available: https://doi.org/10.3233/FAIA200231

[17] Y. Li, D. Wang, and J. M. Moura, "Gsa-forecaster: Forecasting graph-based time-dependent data with graph sequence attention," *arXiv preprint arXiv:2104.05914*, 2021.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. ACM, 2010, pp. 135–146.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[20] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 599–613. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[21] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 2015, pp. 243–252.

[22] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Advances in Neural Information Processing Systems*, 2014, pp. 1673–1681.

[23] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. ACM, 2014, pp. 333–342.

[24] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: an experimental study," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1590–1603, 2018.

[25] Z. Abbas, A. Al-Shishtawy, S. Girdzijauskas, and V. Vlassov, "Short-term traffic prediction using long short-term memory neural networks," in *IEEE International Congress on Big Data*, July 2018, pp. 57–65.

[26] Z. Abbas, T. T. Sigurdsson, A. Al-Shishtawy, and V. Vlassov, "Evaluation of the use of streaming graph processing algorithms for road congestion detection," in *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications (ISPA)*, Dec 2018, pp. 1017–1025.

[27] Z. Abbas, P. Sottovia, M. A. Hajj Hassan, D. Foroni, and S. Bortoli, "Real-time traffic jam detection and congestion reduction using streaming graph analytics," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 3109–3118.

[28] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 2012, pp. 85–98.

[29] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. ACM, 2016, p. 5.

[30] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[31] A. Iyer, L. E. Li, and I. Stoica, "CellIQ: real-time cellular network analytics at scale," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 309–322.

[32] "Gcn-based partitioning of unbounded graph streams with bounded state," in *Under preparation for submission*.

[33] M. Fouladgar, M. Parchami, R. Elmasri, and A. Ghaderi, "Scalable deep traffic flow neural networks for urban traffic congestion prediction," in *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2251–2258.

[34] X. Ma *et al.*, "Learning traffic as images: a deep convolutional neural network for large-scale transportation network speed prediction," *Sensors*, vol. 17, no. 4, p. 818, 2017.

[35] X. Ma, H. Yu, Y. Wang, and Y. Wang, "Large-scale transportation network congestion evolution prediction using deep learning theory," *PLOS ONE*, vol. 10, no. 3, pp. 1–17, 03 2015.

[36] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, pp. 28–38, 2015.

[37] B. Anbaroglu, B. Heydecker, and T. Cheng, "Spatio-temporal clustering for non-recurrent traffic congestion detection on urban road networks," *Transportation Research Part C: Emerging Technologies*, vol. 48, pp. 47–65, Nov. 2014.

[38] B. Anbaroglu, T. Cheng, and B. Heydecker, "Non-recurrent traffic congestion detection on heterogeneous urban road networks," *Transportmetrica A: Transport Science*, vol. 11, pp. 1–33, 09 2015.

[39] F. Soylemezgiller, M. Kuscu, and D. Kilinc, "A traffic congestion avoidance algorithm with dynamic road pricing for smart cities," in *2013 IEEE 24th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, 2013, pp. 2571–2575.

[40] U. Elsner, "Graph partitioning-a survey," 1997.

[41] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," *ICASE LaRC Interdisciplinary Series in Science and Engineering*, vol. 4, pp. 323–368, 1997.

[42] J. Kim, I. Hwang, Y.-H. Kim, and B.-R. Moon, "Genetic approaches for graph partitioning: a survey," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2011, pp. 473–480.

[43] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[44] Y. Guo, S. Hong, H. Chafi, A. Iosup, and D. Epema, "Modeling, analysis, and experimental comparison of streaming graph-partitioning policies," *Journal of Parallel and Distributed Computing*, vol. 108, pp. 106–121, 2017.

[45] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, "An experimental comparison of partitioning strategies in distributed graph processing," *PVLDB*, vol. 10, no. 5, pp. 493–504, 2017.

[46] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[47] J. D. Bali, "Streaming graph analytics framework design," Master's thesis, KTH, School of Information and Communication Technology (ICT), 2015.

[48] M. Mitropolitsky, Z. Abbas, and A. H. Payberah, "Graph representation matters in device placement," in *Proceedings of the Workshop on Distributed Infrastructures for Deep Learning*, ser. DIDL'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–6. [Online]. Available: https://doi.org/10.1145/3429882.3430104

[49] S. Imtiaz, S.-F. Horchidan, Z. Abbas, M. Arsalan, H. N. Chaudhry, and V. Vlassov, "Privacy preserving time-series forecasting of user health data streams," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 3428–3437.

[50] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*. ACM, 2010, pp. 135–146.

[51] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: graph processing at Facebook-scale," *PVLDB*, vol. 8, no. 12, pp. 1804–1815, 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824077

[52] "Apache Storm project," http://storm.apache.org/.

[53] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 2, pp. 305–324, 2018.

[54] K. Andreev and H. Räcke, "Balanced graph partitioning," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2004, pp. 120–124.

[55] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2014, pp. 1456–1465.

[56] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[57] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent Advances in Graph Partitioning*. Cham: Springer International Publishing, 2016, pp. 117–158. [Online]. Available: https://doi.org/10.1007/978-3-319-49487-6_4

[58] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *ICPP (3)*, 1995, pp. 113–122.

[59] I. Stanton, "Streaming balanced graph partitioning algorithms for random graphs," in *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '14. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, pp. 1287–1301. [Online]. Available: http://dl.acm.org/citation.cfm?id=2634074.2634169

[60] C. Tsourakakis, "Streaming graph partitioning in the planted partition model," in *Proceedings of the 2015 ACM on Conference on Online Social Networks*, ser. COSN '15. New York, NY, USA: ACM, 2015, pp. 27–35. [Online]. Available: http://doi.acm.org/10.1145/2817946.2817950

[61] H. P. Sajjad, A. H. Payberah, F. Rahimian, V. Vlassov, and S. Haridi, "Boosting vertex-cut partitioning for streaming graphs," in *2016 IEEE International Congress on Big Data (BigData Congress)*, 2016, pp. 1–8.

[62] A. McGregor, "Graph stream algorithms: A survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[63] A. Nazi, W. Hang, A. Goldie, S. Ravi, and A. Mirhoseini, "Gap: Generalizable approximate graph partitioning framework," *arXiv preprint arXiv:1903.00614*, 2019.

[64] ——, "Generalized clustering by learning to optimize expected normalized cuts," *arXiv preprint arXiv:1910.07623*, 2019.

[65] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[66] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artifical Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159, 2020.

[67] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[68] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.

[69] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[70] Y. Zhang and K. Rohe, "Understanding regularized spectral clustering via graph conductance," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 10 631–10 640. [Online]. Available: http://papers.nips.cc/paper/8262-understanding-regularized-spectral-clustering-via-graph-conductance.pdf

[71] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big simple neural nets excel on handwritten digit recognition," *CoRR*, vol. abs/1003.0358, 2010.

[72] J. Dean *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[73] J. Konečnỳ *et al.*, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[74] M. J. Lighthill and G. B. Whitham, "On kinematic waves ii. a theory of traffic flow on long crowded roads," *Proc. R. Soc. Lond. A*, vol. 229, no. 1178, pp. 317–345, 1955.

[75] S. Sun, C. Zhang, and G. Yu, "A bayesian network approach to traffic flow forecasting," *IEEE Transactions on intelligent transportation systems*, vol. 7, no. 1, pp. 124–132, 2006.

[76] H. Sun, H. X. Liu, H. Xiao, R. R. He, and B. Ran, "Short term traffic forecasting using the local linear regression model," in *82nd Annual Meeting of the Transportation Research Board, Washington, DC*, 2003.

[77] M. S. Ahmed and A. R. Cook, "Analysis of freeway traffic time-series data by using box-jenkins techniques," *Transportation Research Record Journal of the Transportation Research Board*, no. 722, 1979.

[78] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang, "Traffic flow prediction with big data: a deep learning approach," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 865–873, 2015.

[79] N. G. Polson and V. O. Sokolov, "Deep learning for short-term traffic flow prediction," *Transportation Research Part C: Emerging Technologies*, vol. 79, pp. 1–17, 2017.

[80] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[81] Z. Zhao, W. Chen, X. Wu, P. C. Chen, and J. Liu, "Lstm network: a deep learning approach for short-term traffic forecast," *IET Intelligent Transport Systems*, vol. 11, no. 2, pp. 68–75, 2017.

[82] X. Ma, Z. Tao, Y. Wang, H. Yu, and Y. Wang, "Long short-term memory neural network for traffic speed prediction using remote microwave sensor data," *Transportation Research Part C: Emerging Technologies*, vol. 54, pp. 187–197, 2015.

[83] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *15th annual conference of the international speech communication association*, 2014.

[84] J. Inman, "Navigation and nautical astronomy, for the use of british seamen," 1849.

[85] Trafikverket, https://www.trafikverket.se/, 2010.

[86] F. Fouquet, T. Hartmann, S. Mosser, and M. Cordy, "Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1054–1061.

[87] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon, "Analyzing complex data in motion at scale with temporal graphs," 2017.

[88] M. Barth and K. Boriboonsomsin, "Real-world carbon dioxide impacts of traffic congestion," *Transportation Research Record*, vol. 2058, no. 1, pp. 163–171, 2008.

[89] P. Goodwin, "The economic costs of road traffic congestion," 2004.

[90] U. States., *Vehicle- and infrastructure-based technology for the prevention of rear-end collisions [electronic resource]*. National Transportation Safety Board Washington, D.C, 2001.

[91] D. Stokols, R. W. Novaco, J. Stokols, and J. Campbell, "Traffic congestion, Type A behavior, and stress." *Journal of Applied Psychology*, vol. 63, no. 4, pp. 467–480, 1978.

[92] J. Currie and R. Walker, "Traffic Congestion and Infant Health: Evidence from E-ZPass," *American Economic Journal: Applied Economics*, vol. 3, no. 1, pp. 65–90, Jan. 2011.

[93] B. S. Kerner, *The physics of traffic: empirical freeway pattern features, engineering applications, and theory*. Berlin: Springer, 2010.

[94] H. Rehborn and J. Palmer, "Asda/foto based on kerner's three-phase traffic theory in north rhine-westphalia and its integration into vehicles," in *2008 IEEE Intelligent Vehicles Symposium*, 2008, pp. 186–191.

[95] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing." NetDB, 2011.

[96] V. L. Knoop and W. Daamen, "Automatic fitting procedure for the fundamental diagram," *Transportmetrica B: Transport Dynamics*, vol. 5, no. 2, pp. 129–144, 2017.

[97] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "Graph distances in the data-stream model," *SIAM Journal on Computing*, vol. 38, no. 5, pp. 1709–1727, 2008.

[98] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. WieBner, "Microscopic traffic simulation using sumo," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2018, pp. 2575–2582.

[99] "The bright side of sitting in traffic: Crowdsourcing road congestion data." [Online]. Available: https://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html

[100] "General data protection regulation (eu)," Tech. Rep., 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=EN

[101] F. Porikli and Xiaokun Li, "Traffic congestion estimation using hmm models without vehicle tracking," in *IEEE Intelligent Vehicles Symposium, 2004*, 2004, pp. 188–193.

[102] G. Di Leo, A. Pietrosanto, and P. Sommella, "Metrological performance of traffic detection systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 58, no. 9, pp. 3199–3206, 2009.

[103] F. Mehboob, M. Abbas, and R. Jiang, "Traffic event detection from road surveillance vide os based on fuzzy logic," in *2016 SAI Computing Conference (SAI)*, 2016, pp. 188–194.

[104] Q. Wang, J. Wan, and Y. Yuan, "Locality constraint distance metric learning for traffic congestion detection," *Pattern Recogn.*, vol. 75, no. C, p. 272–281, Mar. 2018. [Online]. Available: https://doi.org/10.1016/j.patcog.2017.03.030

[105] S. A. Vaqar and O. Basir, "Traffic pattern detection in a partially deployed vehicular ad hoc network of vehicles," *IEEE Wireless Communications*, vol. 16, 2009.

[106] R. Bauza and J. Gozálvez, "Traffic congestion detection in large-scale scenarios using vehicle-to-vehicle communications," *Journal of Network and Computer Applications*, vol. 36, no. 5, pp. 1295–1307, 2013.

[107] L. Zhang, D. Gao, W. Zhao, and H.-C. Chao, "A multilevel information fusion approach for road congestion detection in vanets," *Mathematical and Computer Modelling*, vol. 58, no. 5-6, pp. 1206–1221, 2013.

[108] X. Yu, S. Xiong, Y. He, W. E. Wong, and Y. Zhao, "Research on campus traffic congestion detection using bp neural network and markov model," *Journal of information security and applications*, vol. 31, pp. 54–60, 2016.

[109] X. Cheng, W. Lin, E. Liu, and D. Gu, "Highway traffic incident detection based on bpnn," *Procedia Engineering*, vol. 7, pp. 482–489, 2010.

[110] B. P. L. Lo and S. Velastin, "Automatic congestion detection system for underground platforms," in *Proceedings of 2001 International Symposium on Intelligent Multimedia, Video and Speech Processing. ISIMP 2001 (IEEE Cat. No. 01EX489)*. IEEE, 2001, pp. 158–161.

[111] D. Bowers, R. Bretherton, and G. Bowen, "The astrid/ingrid incident detection system for urban areas," Tech. Rep., 1995.

[112] T. Cherrett, B. Waterson, and M. McDonald, "Remote automatic incident detection using inductive loops," in *Proceedings of the Institution of Civil Engineers-Transport*, vol. 158, no. 3. Thomas Telford Ltd, 2005, pp. 149–155.

[113] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[114] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[115] A. McGregor, "Graph mining on streams," *Encyclopedia of Database Systems*, pp. 1271–1275, 2009.

[116] J. Thaler, "Semi-streaming algorithms for annotated graph streams," *arXiv preprint arXiv:1407.3462*, 2014.

[117] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2008, pp. 16–24.

[118] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: consistent stateful distributed stream processing," *PVLDB*, vol. 10, no. 12, pp. 1718–1729, 2017.

[119] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[120] I. Stanton, "Streaming balanced graph partitioning algorithms for random graphs," in *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '14. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, pp. 1287–1301. [Online]. Available: http://dl.acm.org/citation.cfm?id=2634074.2634169

[121] C. Tsourakakis, "Streaming graph partitioning in the planted partition model," in *Proceedings of the 2015 ACM on Conference on Online Social Networks*, ser. COSN '15. New York, NY, USA: ACM, 2015, pp. 27–35. [Online]. Available: http://doi.acm.org/10.1145/2817946.2817950

[122] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan, "Managing large graphs on multi-cores with graph awareness." in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 41–52. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/prabhakaran

[123] N. Jain, G. Liao, and T. L. Willke, "Graphbuilder: scalable graph etl framework," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 4.

[124] J. Hopcroft and R. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[125] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: mining peta-scale graphs," *Knowledge and Information Systems*, vol. 27, no. 2, pp. 303–325, May 2011. [Online]. Available: https://doi.org/10.1007/s10115-010-0305-0

[126] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii, "Connected components in MapReduce and beyond," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 18:1–18:13. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670997

[127] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[128] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[129] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theoretical Computer Science*, vol. 348, no. 2, pp. 207–216, 2005.

[130] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[131] P. C. J. D. Bali, V. Kalavri, "Streaming graph analytics framework design," 2015, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-170425.

[132] M. Cha, A. Mislove, and K. P. Gummadi, "A measurement-driven analysis of information propagation in the Flickr social network," in *Proceedings of the 18th International Conference on World Wide Web*. ACM, 2009, pp. 721–730.

[133] "Online social networks research at the Max Planck Institute for Software Systems," http://socialnetworks.mpi-sws.org/data-imc2007.html.

[134] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298311

[135] "Grouplens," https://grouplens.org/datasets/movielens/.

[136] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th International Conference on World Wide Web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[137] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *2012 IEEE 12th International Conference on Data Mining*, Dec 2012, pp. 745–754.

[138] T.-Y. Cheung, "Graph traversal techniques and the maximum flow problem in distributed computation," *IEEE Transactions on Software Engineering*, no. 4, pp. 504–512, 1983.

[139] S. Guha and A. McGregor, "Stream order and order statistics: Quantile estimation in random-order streams," *SIAM Journal on Computing*, vol. 38, no. 5, pp. 2044–2059, 2009.

[140] J. J. Pfeiffer III, S. Moreno, T. La Fond, J. Neville, and B. Gallagher, "Attributed graph models: Modeling network structure with correlated attributes," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 831–842.

[141] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[142] S. Cao, W. Lu, and Q. Xu, "Grarep: Learning graph representations with global structural information," in *Proceedings of the 24th ACM international on conference on information and knowledge management*, 2015, pp. 891–900.

[143] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, and C. E. Leiserson, "Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics," *arXiv preprint arXiv:1908.02591*, 2019.

[144] "GCNSplit Project Repository and Appendix," https://github.com/, 2021.

[145] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[146] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[147] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai,

and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[148] B. Rozemberczki, C. Allen, and R. Sarkar, "Multi-scale attributed node embedding," 2019.

[149] B. Rozemberczki, R. Davies, R. Sarkar, and C. Sutton, "Gemsec: Graph embedding with self clustering," in *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2019*. ACM, 2019, pp. 65–72.

[150] K. Wang, Z. Shen, C. Huang, C.-H. Wu, Y. Dong, and A. Kanakia, "Microsoft academic graph: When experts are not enough," *Quantitative Science Studies*, vol. 1, no. 1, pp. 396–413, 2020.

[151] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[152] J. You, R. Ying, and J. Leskovec, "Position-aware graph neural networks," *arXiv preprint arXiv:1906.04817*, 2019.

[153] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 489–504.

[154] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "Lisa: A learned index structure for spatial data," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2119–2133.

[155] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "Alex: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.

[156] A. Pacaci and M. T. Özsu, "Experimental analysis of streaming algorithms for graph partitioning," ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1375–1392. [Online]. Available: https://doi.org/10.1145/3299869.3300076

[157] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[158] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1067–1077.

[159] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[160] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[161] B. Olivier, G. Pierre, H. Nicolas, O. Loïc, T. Olivier, and T. Philippe, *Multi sensor data fusion architectures for Air Traffic Control applications*. Citeseer, 2009.