Bulletin of the Technical Committee on

Data Engineering

September 2017 Vol. 40 No. 3

IEEE Computer Society

Letters

The TC on Data Engineering
Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is `http://tab.computer.org/tcde/index.html`.

The Data Engineering Bulletin
The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at
`http://tab.computer.org/tcde/bull_about.html`.

Letter from the Editor-in-Chief

Computer Society Election

The Computer Society has an election annually to choose officers and Board of Governors (BOG) members. Officers have one year terms. BOG members have three year terms, with one third being elected annually. The results for the just completed election are posted at `https://www.computer.org/web/election/election-results`. Hironori Kashara, elected president-elect last year, becomes president in January. The newly elected president-elect is Cecilia Metra. I want to congratulate them and wish them well as they begin their tenures in these leadership offices.

As for me, my current offices as treasurer and first vice president continue through the end of the year. As of January, I will again become a member of the BOG. I want to thank Computer Society members, and especially TCDE members, for electing me again to a three year term as a BOG member.

The Current Issue

Graph data management has grown in importance and interest over the past ten years, and especially as a result of the emergence of social media and social media companies such as Facebook and Twitter. Graphs, among other things, are used to represent people and their connections to other people. As we know from our own use of social media, from the importance of online advertising that connects people to shopping interests, and from how social media influenced the recent US presidential election, social data now has enormous sway on the lives, not just of technology folks, but the general population as well.

This now pervasive presence of social media and its wide and still growing influence have made catering to its needs, and especially its need for managing graphs, an important application for data management. Graphs share many of the problems of earlier data management applications, the need for high performance, low cost, high availability, security, etc. But managing graphs has some of these problems in the extreme. Scalability and geo-distribution are huge issues. These make graph partitioning important. Subgraph matching is an issue for social connectedness analysis, with important applications to ad placement. There are more.

The current issue shows one of the benefits of a publication like the Bulletin. It contains in a single place, contributions from both industry and academia, providing an up-to-date view of the area that is hard to match elsewhere. For readers who want to learn the latest about graph data processing, its application, and its importance, it is hard to beat the current issue. Haixun Wang, from Facebook, has served as the issue editor. He works in the area and knows it well, and the people who work in the area, both industrial and academic. My thanks to Haixun for succeeding in bringing this important collection of papers together and serving as issue editor.

David Lomet
Microsoft Corporation

Letter from the 2017 IEEE TCDE Impact Award Winner

This year I was honored to be given the Impact Award "For expanding the reach of data engineering within scientific disciplines." My interest in scientific applications started in the late 1980's when I met Dr. Chris Overton, who held a PhD in Developmental Biology and came to the University of Pennsylvania to complete a Master's in Computer and Information Science, because he believed that *the future of biology was computational* – quite a visionary for the time! After Chris was hired to head up the informatics component of the Center for Human Chromosome 22 in the 1990s, we frequently discussed the challenges he faced. This became a rich vein of research problems that the Database Group at Penn has worked on for over two decades. Most importantly, *addressing these challenges involved a close collaboration between end-users, systems builders and database experts.* Two of my favorite problems were data integration and provenance.

Data integration. Most data integration systems in the 1990's were built around relational databases, however genomic data was frequently stored in specialized file formats with programmatic interfaces. This led experts to state in a report of the 1993 Invitational DOE Workshop on Genome Informatics that "Until a fully relationalized sequence database is available, none of the queries in this appendix can be answered." However, the real problem was twofold: 1) integrating non-relational data sources; and 2) knowing what information was available and where. We answered the "unanswerable queries" within about a month using our data integration system, Kleisli, which used a complex-object model of data, language based on a comprehension syntax, and optimizations that went beyond relational systems. Our team also included experts who knew where the appropriate data sources were and how to use them. Since then, the database community has made excellent contributions in developing data integration systems that go well beyond the relational model; less progress has been made on knowing how to find the appropriate data sources and how to extract the right information.

Data provenance. Our team originally recognized the need for provenance when constructing an integrated dataset of genomic information: Not all data sources were equally trusted, but no-one wanted to express this opinion by failing to include a relevant data set. The solution was to make provenance available so that users could form their own conclusions. Since then, the importance of provenance has been widely recognized, especially as it relates to reproducibility and debugging, and the database community has made excellent progress in "coarse-grained" provenance for workflows, "fine-grained" database style provenance, and "event-log" style provenance. However, the usability of provenance remains a challenge: provenance is collected more often than it is used!

Bioinformatics is just a precursor of the "tsunami" that is now Data Science, and many even more interesting challenges lie ahead – see the CRA report on "Computing Research and the Emerging Field of Data Science" (available at http://cra.org). As before, these problems are best addressed by teams of people working together. I am encouraged to see our community rising to these challenges, and expanding the chain of end-users, systems builders and database experts to include statisticians and machine learning researchers, among many other types of experts required in developing solutions to real problems in Data Science.

Susan B. Davidson
University of Pennsylvania

Letter from the 2017 IEEE TCDE Service Award Winner

I am pleased and humbled to receive the 2017 IEEE TCDE Service Award. It is the recognition of my 36 years of work! Thanks to the award committee and those who nominated me and supported my nomination! Special thanks to my supervisor, Prof. Shixuan Sa, who was a professor of the Renmin University of China. Professor Sa introduced me to the field of database research.

From 1984 to 1986, I was a visiting scholar of the University of Maryland, where I learned the modern database technologies and participated in the system development of XDB (an extensible relational database system). I founded the first Institute of Data Engineering and Knowledge Engineering in China in 1987. The Institute specialized in the database system research and development. We have developed a series of database management system in the past 30 years, including RDBMS, parallel database system, parallel data warehouse, Chinese Natural Language Interface of RDBMS, Mobile Database System and memory database system, etc.

Many thanks to my colleagues and my students who researched and developed database systems with me. Together we started an enterprise on database, which is now providing DBMS products ,KingbaseES ,to the public sector in China.

As an educator I have put continuous efforts in promoting database education in China. I am the author of one of the classic database textbooks in China. The textbook was first published in 1983 and released its 5th edition in 2014. This textbook has been sold more than 2 million copies and affected generations of database researchers in China. The book has been translated in Tibetan and traditional Chinese.

In 1999, the China Computer Federation Technical Committee on Databases(CCF TCDB) was established. I was the first director of TCDB. One of the TCDB missions is to build and strengthen the relationship between China and the rest of the world. The CCF TCDB actively hold many international conferences, including VLDB2002, ER2004, SIGMOD/PODS2007, IEEE ICDE2009, DASFAA(many times), WAIM(many times), APWEB (many times), and so on. I have served as the general chairman or the honorary chairman for many of them.

We actively worked with international organizations. For example, the CCF TCDB and the Japanese Database Committee established good relationship for years. Two committees sent representatives to participate each other's national databases events.

I had served in China Computer Federation(CCF) and the TCDB for a long time, and served as vice president of the CCF. I was also a Steering Committee member of DASFAA, WAIM, APWeb, etc. I treated each work wholeheartedly and by doing so, I received my college's respect and trust. I was awarded the CCF Distinguished Contribution Award, DASFAA Outstanding Contribution Award, WAIM Outstanding Contribution Award, and APWeb Outstanding Contribution Award, and this 2017 IEEE TCDE Service Award.

Appreciate the recognition from all of the international experts and friends. I am truly honored.

Shan Wang
Renmin University

Letter from the 2017 IEEE TCDE Early Career Award Winner

Rethinking Data Analytics with Humans-in-the-loop

From large-scale physical simulations, to high-throughput genomic sequencing, and from conversational agent interactions, to sensor data from the Internet of Things, the need for data analytics—extracting insights from large datasets—has never been greater. At the same time, current data analytics tools are powerless in harnessing the hidden potential within these datasets. The bottleneck is not one of "scale"—we already know how to process large volumes of data quickly—but instead stems from the *humans-in-the-loop*. As dataset sizes have grown, the time for human analysis, the cognitive load taken on by humans, and the human skills to extract value from data, have either stayed constant, or haven't grown at a commensurate rate. Thus, at present, *there is a severe lack of powerful tools that incorporate humans as a "first-class citizen" in data analytics, helping them interactively manage, analyze, and make sense of their large datasets.*

My research has centered on the design of efficient and usable *Human-in-the-Loop Data Analytics* (HILDA) tools, spanning the spectrum from *manipulate* $\rightarrow$ *visualize* $\rightarrow$ *collaborate* $\rightarrow$ *understand*: (a) For users not currently able to even examine or *manipulate* their large datasets, I am developing DATASPREAD, a *spreadsheet-database hybrid* (dataspread.github.io). (b) Then, once users can examine their large datasets, the next step is to *visualize* it: I am developing ZENVISAGE, a *visualization search and recommendation system*, to allow users to rapidly search for and identify visual patterns of interest, without effort (zenvisage.github.io). (c) Then, to *collaborate* on and share the discovered insights with others, I am developing ORPHEUS, a collaborative analytics system that can *efficiently manage and maintain dataset versions* (orpheus-db.github.io). (d) Finally, to *understand* data at a finer granularity by using humans to annotate data for training machine learning algorithms, I am developing POPULACE, an *optimized crowdsourcing system* (populace-org.github.io).

Developing these HILDA tools requires techniques not just in database systems, but also in data mining and in Human-Computer Interaction (HCI)—we've had to evaluate our systems not just in terms of scalability and latency, but also accuracy and utility (from data mining), and interactivity and usability (from HCI). In developing these tools, we've also had to go outside of our comfort zone in talking to *real users*: biologists, battery scientists, ad analysts, neuroscientists, and astrophysicists, in identifying usage scenarios, pain-points, and challenges, thereby ensuring that our tools meet real user needs. Indeed, many of these individuals and teams have access to large datasets, and a pressing need to extract insights and value from them, but are not able to do so. This is due to the lack of powerful tools that can reduce the amount of human effort, labor, time, and tedium, and at the same time, minimize the need for sophisticated programming and analysis skills.

While our tools represent a promising start, we are barely scratching the surface of this nascent research field. Future research on HILDA will hopefully enable us to make steps towards meeting the grand challenge of empowering scientists, business users, consultants, finance analysts, and lay users with a new class of tools that equips them with what they need to manage, make sense of, and unlock value from data. We envision that data-driven discovery of insights in the future will no longer be bottlenecked on the "humans-in-the-loop", and will instead depend on fluid interactions facilitated by powerful, scalable, usable, and intelligent HILDA tools.

Aditya Parameswaran
University of Illinois UC

Letter from the Special Issue Editor

In the last decade, a tremendous amount of work has been devoted to managing and mining large graphs. We have witnessed during this golden time the dramatic rise of social networks, knowledge graphs, and data of increasingly rich relationships. Publications on graph related research also thrived. In this issue, we review some of the biggest challenges, survey a few brilliant solutions, and reflect on the current status and the future of this field.

Big graphs bring a lot of algorithmic challenges, and as a result, topics such as graph partitioning, graph reachability, keyword search, subgraph matching, etc. have attracted a lot of attention. In this issue, we chose not to focus on any specific algorithms, partly because there are just too many to cover. Instead, we take a system oriented approach, that is, we focus on work on managing and understanding graphs that lead to general purpose systems that support a variety of algorithms on big graphs. When the data we are dealing with contains billions of records (nodes) and trillions of relationships (edges), how to manage the data needs to take precedence over how to design ad-hoc algorithms that assume certain data organization tailored for the algorithms.

In "Graph Processing in RDBMSs," Zhao and Yu showed that, instead of devising specific algorithms with specific data organization, many graph processing needs can be supported in RDBMs with an advanced query language. They revisit and enhance SQL recursive queries and show that a set of fundamental graph operations are ensured to have a fixpoint. In "Trinity Graph Engine and its Applications," Shao, Li, Wang and Xia introduced the Trinity Graph Engine, whichh is an open-source distributed in-memory data processing engine, underpinned by a strongly-typed in-memory key-value store and a general distributed computation engine. It is used to serve real-time queries for many real-life big graphs such as Microsoft Knowledge Graph and Microsoft Academic Graph. In "GRAPE: Conducting Parallel Graph Computations without Developing Parallel Algorithms", Fan, Xu, Luo, Wu, Yu, and Xu develop a general purpose framework to parallelize existing sequential graph algorithms, without recasting the algorithms into a parallel model. It is clear that the two most important tasks are managing and mining graph data. In "Towards A Unified Graph Model for Supporting Data Management and Usable Machine Learning," Li, Zhang, Chen, and Ooi present a preliminary design of a graph model for supporting both data management and usable machine learning at the same time.

In many graph systems including the above, graphs are stored in their native forms (nodes and edges). Machine learning tasks on graphs may require a very different representation of graphs. In "Representation Learning on Graphs: Methods and Applications," Hamilton, Ying, and Leskovec tried to find a way to represent or encode graph structure so that it can be easily exploited by machine learning models. They develop a unified framework to describe recent approaches, and highlighted a number of important applications and directions for future work. In "On Summarizing Large-Scale Dynamic Graphs," Shah, Koutra, Jin, Zou, Gallagher, and Faloutsos focused on how to describe a large, dynamic graph over time using an information-theoretic approach. Specifically, it compresses graphs by summarizing important temporal structures and finds patterns that agree with intuition.

Finally, we also highlight the biggest challenges in the field. In "Billion-Node Graph Challenges", Xiao and Shao describe the challenges in the managing and mining billon-node graphs in a distributed environment, while in "Mining Social Streams: Models and Applications," Subbian and Aggarwal focus specifically on challenges in online social networks.

Haixun Wang
Facebook

Graph Processing in RDBMSs

Kangfei Zhao, Jeffrey Xu Yu
The Chinese University of Hong Kong
Hong Kong, China
{kfzhao,yu}@se.cuhk.edu.hk

Abstract

To support analytics on massive graphs such as online social networks, RDF, Semantic Web, etc. many new graph algorithms are designed to query graphs for a specific problem, and many distributed graph processing systems are developed to support graph querying by programming. A main issue to be addressed is how RDBMS can support graph processing. And the first thing is how RDBMS can support graph processing at the SQL level. Our work is motivated by the fact that there are many relations stored in RDBMS that are closely related to a graph in real applications and need to be used together to query the graph, and RDBMS is a system that can query and manage data while data may be updated over time. To support graph processing, we propose 4 new relational algebra operations, MM*-join,* MV*-join, anti-join, and union-by-update. Here,* MM*-join and* MV*-join are join operations between two matrices and between a matrix and a vector, respectively, followed by aggregation computing over groups, given a matrix/vector can be represented by a relation. Both deal with the semiring by which many graph algorithms can be supported. The anti-join removes nodes/edges in a graph when they are unnecessary for the following computing. The union-by-update addresses value updates to compute PageRank, for example. The 4 new relational algebra operations can be defined by the 6 basic relational algebra operations with group-by & aggregation. We revisit SQL recursive queries and show that the 4 operations with others are ensured to have a fixpoint, following the techniques studied in* DATALOG*, and we enhance the recursive* with *clause in SQL'99. RDBMSs are capable of dealing with graph processing in reasonable time.*

1 Introduction

Graph processing has been extensively studied to respond the needs of analyzing massive online social networks, *RDF*, Semantic Web, knowledge graphs, biological networks, and road networks. A large number of graph algorithms have been used/proposed/revisited. Such graph algorithms include *BFS* (Breadth-First Search) [20], *Connected-Component* [57], shortest distance [20], topological sorting [34], *PageRank* [42], *Random-Walk-with-Restart* [42], *SimRank* [31], *HITS* [42], *Label-Propagation* [55], *Maximal-Independent-Set* [46], and *Maximal-Node-Matching* [52], to name a few. In addition to the effort to design efficient graph algorithms to analyze large graphs, many distributed graph processing systems have been developed using the vertex-centric programming. A recent survey can be found in [44]. Such distributed graph processing systems

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

provide a framework on which users can implement graph algorithms to achieve high efficiency. Both new graph algorithms and distributed graph processing systems focus on efficiency. On the other hand, graph query languages have also been studied [69, 15, 24, 28, 38, 28]. In addition, DATALOG has been revisited to support graph analytics [63, 62, 61, 59, 60, 17].

In this work, we investigate how *RDBMS* can support graph processing at *SQL* level for the following reasons. First, *RDBMS* is to manage various application data in relations as well as to query data in relations efficiently using the sophisticated query optimizer. A graph may be a labeled graph with node/edge label, and it is probable that additional information is associated with the graph (e.g. attributed graphs). A key point is that we need to provide a flexible way for users to manage and query a graph together with many relations that are closely related to the graph. The current graph systems are developed for processing but not for data management. We need a system to fulfill both. Second, there is a requirement to query graphs. In the literature, many new graph algorithms are studied to query a specific graph problem. And the current graph processing systems developed do not have a well-accepted graph query language for querying graphs. In other words, it needs coding, when there is a need to compute a graph algorithm based on the outputs of other graph algorithms.

We revisit recursive *SQL* queries [22] and show that a large class of graph algorithms can be supported by *SQL* in *RDBMS*s in [74]. The issue of supporting graph algorithms in *RDBMS* at *SQL* level is the issue how recursive *SQL* can be used to support graph algorithms. There are two main concerns regarding recursive *SQL* queries. One is a set of operations that are needed to support a large pool of graph algorithms and can be used in recursive *SQL* queries. The other is the way to ensure the recursive *SQL* queries can obtain a unique answer. The two concerns are interrelated.

The paper is organized as follows. Section 2 reviews the related works. Section 3 discusses the recursion handling by *SQL* in *RDBMS*s. In Section 4, we present our approach to support graph processing by *SQL* followed by the discussion on how to ensure the fixpoint semantics in Section 5. We conclude our work in Section 6.

2 Related Works

Graph Processing and Management. Many graph systems have been extensively studied recent years. Distributed graph processing systems like *Pregel* [41], *Giraph* [1], *GraphLab* [2] and *GraphX* [3] adopt 'think like a vertex' programming model. Program logic is expressed by vertex-centric push/pull based message passing, which is scalable for very large graph data. Apart from using some common built-in algorithms, users need to implement algorithms using the system APIs provided. The optimization techniques are system-dependent and at individual algorithm level. Such systems compute graphs imported and do not support graph maintenance. Different from graph processing systems, graph management systems e.g. *Neo4j* [4], *AgensGraph* [5], *Titan* [6] are designed for the transactional graph-like query. *Neo4j* adopts the property graph model and supports a visual UI for querying by the declarative language *Cypher*. *AgensGraph* is a multi-model graph database based on *PostgreSQL RDBMS*. *Titan* uses the functional language *Gremlin* to query and update graph.

Graph Query Languages. Graph query languages have been studied. A survey on query languages for graph databases can be found in [69], which covers conjunctive query (CQ), regular path query (RPQ), and CRPQ combining CQ and RPQ. Also, it surveys a large number of languages including *Lorel*, *StruQL*, *UnQL*, **G**, $\mathbf{G}^+$, *GraphLog*, *G-Log*, *SoSQL*, etc. Barceló investigates the expressive power and complexity of graph query languages [15]. Libkin et al. in [38] study how to combine data and topology by extending regular expressions to specify paths with data. The declarative, *SQL*-inspired query language *Cypher* [4], and functional language *Gremlin* [6] are integrated into transactional graph databases to describe graph patterns and traversal queries. There are several new attempts to query graphs. The *PGQL* [68] is a property graph query language in the *Oracle* Parallel Graph AnalytiX (PGX) toolkit, and is an *SQL*-like query language for graph matching. *PGQL* supports path queries and has the graph intrinsic type for graph construction and query composition. Moffitt and Stoyanovich in [47] present a declarative query language *Portal* for querying evolving graphs, which is based on temporal relational algebra and is implemented on *GraphX*. Gao et al. in [24] propose a graph language *GLog* on Relational-Graph, which is a data model by mixing relational and graph data. A *GLog* query is converted into a sequence of MapReduce jobs to be processed on distributed systems. Jindal and Madden propose *graphiQL* in [32] by exploring a way to combine the features of *Pregel* (vertex-centric programming) and *SQL*. He and Singh in [28] propose the language *GraphQL* on graph algebra which deals with graphs with attributes as a basic unit, where the operations in the graph algebra include selection, Cartesian product, and composition. Salihoglu and Widom in [58] propose *HeLP*, a set of basic operations needed in many graph processing systems.

Recursive SQL Queries: *SQL*'99 supports recursive queries [45, 22]. As mentioned, in supporting graph algorithms, there are two main issues regarding recursive *SQL* queries: a set of operations that can be used in recursive *SQL* queries, and a way to ensure unique solution by recursive *SQL* queries. For the former, Cabrera and Ordonez in [18] and Kang et al. in [36] discuss an operation to multiply a matrix with a vector using joins and group-by & aggregation. Cabrera and Ordonez in [18] discuss semiring for graph algorithms, and give a unified algorithm which is not in *SQL*. For the latter, recursive query processing is well discussed in [12]. Ordonez et al. in [50] compare *SQL* recursive query processing in columnar, row and array databases. Ghazal et al. propose an adaptive query optimization scheme for the recursive query in *Teradata*, which employs multi-iteration pre-planning and dynamic feedback to take advantage of global query optimization and pipelining [26]. Aranda et al. in [13] study broadening recursion in *SQL*. The *SQL* level optimizations for computing transitive closures are discussed in [49], with its focus on monotonic aggregation for transitive closures. All the works in [50, 26, 13, 49] do not deal with negation and aggregation. It is important to note that aggregation and negation in general are needed for a large pool of graph algorithms, but aggregations and negations cannot be used within a recursive *SQL* query for ensuring that an *SQL* query can get a unique solution.

Graph Analytics by SQL. Graph analytics in *RDBMS*s using *SQL* have been studied. Srihari et al. in [64] introduce an approach for mining dense subgraphs in a *RDBMS*. Gao et al. in [23] leverage the window functions and the merge statement in *SQL* to implement shortest path discovery in *RDBMS*. Zhang et al. in [73] provide an *SQL*-based declarative query language *SciQL* to perform array computation in *RDBMS*s. Fan et al. in [21] propose *GRAIL*, a syntactic layer converting graph queries into *SQL* script. *GraphGene* [70] is a system for users to specify graph extraction layer over relational databases declaratively. *MADLib* is designed and implemented to support machine learning, data mining and statistics on database systems [19, 29]. Passing et al. in [51] propose a *PageRank* operator in main-memory *RDB*. This operator is implemented by extending the *SQL* recursive query by lambda expression. In [33], *Vertica* relational database is studied as the platform for vertex-centric graph analysis. In [65], a graph storage system *SQLGraph* is designed, which combines the relational storage for adjacency information with *JSON* for vertex and edge properties. It shows that it can outperform popular *NoSQL* graph stores. *GQ-Fast* [39] is an indexed and fragmented database which supports efficient *SQL* relationship queries for typed graph analytics. Ma et. al in [40] present *G-SQL*, an *SQL* dialect for graph exploration. Multi-way join operations are boosted by underlying graph processing engine. Aberger et al. in [11] develop a graph pattern engine, called *EmptyHead*, to process graph patterns as join processing in parallel.

Datalog-based Systems. DATALOG is a declarative query language used in early deductive databases. As its new applications derive from information extraction, data analytics and graph analytics, many DATALOG-based systems have been developed recently. A survey of early deductive database systems can be found in [56]. *LDL++* is a deductive database system in which negation and aggregation handling in recursive rules are addressed [71, 14]. Based on *LDL++*, a new deductive application language system *DeALS* is developed to support graph queries [63], and the optimization of monotonic aggregations is further studied [62]. *SociaLite* [59] allows users to write high-level graph queries based on DATALOG that can be executed in parallel and distributed environments [60]. These systems support graph analytics, especially iterative graph analytics e.g. transitive closure, weakly connect components, single source shortest distance since DATALOG has a great expressive power for recursion. DATALOG for machine learning is studied with *Pregel* and map-reduce-update style programming [17]. The efficient evaluation of DATALOG is investigated on data flow processing system *Spark* [61] and *BSP*-style graph processing engines [48]. In this work, we introduce the DATALOG techniques into *RDBMS*s to deal with recursive *SQL* queries, since DATALOG has greatly influenced the recursive *SQL* query handling.

3 The Recursion in RDBMS

Over decades, *RDBMS*s have provided functionality to support recursive queries, based on *SQL*'99 [45, 22]. The recursive queries are expressed using with clause in *SQL*. It defines a temporary recursive relation R in the initialization step, and queries by referring the recursive relation R iteratively in the recursive step until R cannot be changed. As an example, the edge transitive closure can be computed using with over the edge relation $E(F, T)$, where F and T are for "From" and "To". As shown below, initially, the recursive relation TC is defined to project the two attributes, F and T, from the relation E. Then, the query in every iteration is to union TC computed and a relation with two attributes $TC.F$ and $E.T$ by joining the two relations, TC and E.

with TC (F, T) **as** (**select** F, T **from** E **union all select** $TC.F$, $E.T$ **from** TC, E **where** $TC.T = E.F$)

SQL'99 restricts recursion to be a linear recursion and allows mutual recursion in a limited form [45]. Among the linear recursion, *SQL*'99 only supports monotonic queries, which is known as the monotonicity. In the context of recursion, a monotonic query means that the result of a recursive relation in any iteration does not lose any tuples added in the previous iterations. Such monotonicity ensures that the recursion ends at a fixpoint with a unique result. The definition of monotonicity can be found in [66]. As given in Theorem 3.3 in [66], union, select, projection, Cartesian product, natural joins, and θ-joins are monotone. On the other hand, negation is not monotone [25]. In *SQL*, the operations such as except, intersect, not exists, not in, <> some, <> all, distinct are the operations leading to negation. Also, aggregation can violate the monotonicity. *SQL*'99 does not prohibit negation in recursive queries completely since the monotonicity of recursive query is ensured if the negation is only applied to the relations that are completely known or computed prior to processing the result of the recursion. This is known as stratified negation [72].

SQL Recursion Handling in RDBMS: *SQL*'99 supports stratified negation. Below, we discuss *SQL* recursion handling in *RDBMS*s, following the similar discussions given in [53] in which Przymus et al. survey recursive queries handling in *RDBMS*s. We focus on *Microsoft SQL Server* (2016) [8], *Oracle* (11gR2) [9], *IBM DB2 10.5 Express-C* [7], and *PostgreSQL* (9.4) [10], where *Oracle* is not covered in [53]. We investigate the features related to recursive query processing in 5 categories. (A) linear/nonlinear/mutual recursion. (B) multiple queries used in the with clause, (C) the set operations other than union all that can be used to separate queries in the with clause, (D) the restrictions on group by, aggregate function, and general functions in the recursive step, and (E) the function to control the looping. Table 1 shows the summary, where "✓", "✗", and "–" denote the corresponding functionality is supported, prohibited, and not applicable, respectively, in the with clause.

Handing Recursion by PSM in RDBMS: There is another way to implement recursion, which is *SQL/PSM* (Persistent Stored Modules) included in *SQL* standard [66]. By *SQL/PSM* (or *PSM*), users can define functions/procedures in *RDBMS*s, and call such functions when querying. In a function/procedure definition, users can declare variables, create temporary tables, insert tuples, and use looping where conditions can be specified to exit (or leave) the loop. *PSM* provides users with a mechanism to issue queries using a general-purpose programming language.

Features		*PostgreSQL*	*DB2*	*Oracle*	*SQL Server*
	Linear Recursion	✓	✓	✓	✓
A	Nonlinear Recursion	✗	✗	✗	✗
	Mutual Recursion	✗	✗	✗	✗
B	Initial Step	✓	✓	✓	✓
	Recursive Step	✗	✓	✗	✓
	Between initial queries	✓	✓	✓	✓
C	Across initial & recursive queries	✓	✗	✗	✗
	Between recursive queries	–	✗	–	✗
	Negation	✗	✗	✗	✗
	Aggregate functions	✗	✗	✗	✗
	group by, having	✗	✗	✗	✗
	partition by	✓	✓	✓	✓
D	distinct	✓	✗	✗	✗
	General functions	✓	✗	✓	✓
	Analytical functions	✓	✗	✓	✓
	Subqueries without recursive ref	✓	✓	✓	✓
	Subqueries with recursive ref	✗	✗	✗	✗
	Infinite loop detection	✗	✗	✓	✓
	Cycle detection	✗	✗	✓	✗
E	cycle	✗	✗	✓	✗
	search	✗	✗	✓	✗

Table 1: The with Clause Supported by *RDBMS*s

4 The Power of Algebra

In this paper, we model a graph as a weighted directed graph $G = (V, E)$, where V is a set of nodes and E is a set of edges. A node is associated with a node-weight and an edge is associated with an edge-weight, denoted as $\omega(v_i)$ and $\omega(v_i, v_j)$, respectively. A graph can be represented in matrix form. The nodes with node-weights can be represented as a vector of n elements, denoted as V. The edges with edge-weights can be represented as a $n \times n$ matrix, denoted as M, where its M_{ij} value can be 1 to indicate that there is an edge from v_i to v_j, or the value of the edge weight. Such matrices and vectors have their relation representation. Let V and M be the relation representation of vector V and matrix M, such that $V(ID, vw)$ and $M(F, T, ew)$. Here, ID is the tuple identifier in V. F and T, standing for "From" and "To", form a primary key in M. vw and ew are the node-weight and edge-weight respectively.

4.1 The Four Operations

We discuss a set of 4 relational algebra operations, MM-join, MV-join, anti-join, and union-by-update. Here, MM-join and MV-join support the semiring by which many graph algorithms can be supported. The anti-join is used to remove nodes/edges in a graph when they are unnecessary in the following computing and serves as a selection. The union-by-update is used to deal with value updates in every iteration to compute a graph algorithm, e.g., *PageRank*. It is worth noting that there is no such an operation like union-by-update in relational algebra.

We show that all the 4 relational algebra operations can be defined using the 6 basic relational algebra operations (selection (σ), projection (Π), union ($\cup$), set difference ($-$), Cartesian product ($\times$), and rename (ρ)), together with group-by & aggregation. For simplicity, below, we use "$R_i \rightarrow R_j$" for the rename operation to rename a relation R_i to R_j, and use "$\leftarrow$" for the assignment operation to assign the result of a relational algebra to a temporal relation.

We explain why we need the 4 operations which can be supported by the relational algebra because they do not increase the expressive power of relational algebra. First, it is known that relational algebra can support graph algorithms. However, it is not well discussed how to support explicitly. The set of 4 operations is such an answer. Second, it is known that recursive query is inevitable. In other words, new operations cannot function if they cannot be used in recursive *SQL* queries in *RDBMS*. The 4 operations are the non-monotonic operations that cannot be used in recursive *SQL* queries allowed in *SQL*'99. With the explicit form of the 4 operations, in this work, we show that they can be used in recursive *SQL* queries which lead to a unique answer (fixpoint) by adopting the DATALOG techniques. Third, with the explicit form as a target, we can further study how to support them efficiently. We discuss the 4 operations below.

To support graph analytics, the algebraic structure, namely semiring, is shown to have sufficient expressive power to support many graph algorithms [37, 18]. The semiring is a set of $\mathcal{M}$ including two identity elements, $\mathbf{0}$ and $\mathbf{1}$, with two operations: addition ($+$) and multiplication ($\cdot$). In brief, (1) $(\mathcal{M}, +)$ is a commutative monoid with $\mathbf{0}$, (2) $(\mathcal{M}, \cdot)$ is a monoid with $\mathbf{1}$, (3) the multiplication ($\cdot$) is left/right distributes over the addition ($+$), and (4) the multiplication by $\mathbf{0}$ annihilates $\mathcal{M}$. Below, A and B are two 2×2 matrix, and C is a vector with 2 elements.

$$\mathsf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad \mathsf{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad \mathsf{C} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

The matrix-matrix (matrix-vector) multiplication ($\cdot$), and matrix entrywise sum ($+$) are shown below.

$$\begin{aligned} \mathsf{A} \cdot \mathsf{B} &= \begin{pmatrix} a_{11} \odot b_{11} \oplus a_{12} \odot b_{21} & a_{11} \odot b_{12} \oplus a_{12} \odot b_{22} \\ a_{21} \odot b_{11} \oplus a_{22} \odot b_{21} & a_{21} \odot b_{12} \oplus a_{22} \odot b_{22} \end{pmatrix} \\ \mathsf{A} + \mathsf{B} &= \begin{pmatrix} a_{11} \oplus b_{11} & a_{12} \oplus b_{12} \\ a_{21} \oplus b_{21} & a_{22} \oplus b_{22} \end{pmatrix} \\ \mathsf{A} \cdot \mathsf{C} &= \begin{pmatrix} a_{11} \odot c_1 \oplus a_{12} \odot c_2 \\ a_{21} \odot c_1 \oplus a_{22} \odot c_2 \end{pmatrix} \end{aligned}$$

We focus on the multiplication ($\cdot$), since it is trivial to support the addition ($+$) in relational algebra. Let A and B be two $n \times n$ matrices, and C be a vector with n elements. For the multiplication $\mathsf{AB} = \mathsf{A} \cdot \mathsf{B}$, and $\mathsf{AC} = \mathsf{A} \cdot \mathsf{C}$, we have the following.

$$\mathsf{AB}_{ij} = \bigoplus_{k=1}^{n} \mathsf{A}_{ik} \odot \mathsf{B}_{kj} \tag{1}$$

$$\mathsf{AC}_{i} = \bigoplus_{k=1}^{n} \mathsf{A}_{ik} \odot \mathsf{C}_{k} \tag{2}$$

Here, M_{ij} is the value at the i-th row and j-th column in the matrix M, and V_i is the element at the i-th row in the vector V. Let $A(F, T, ew)$ and $B(F, T, ew)$ be the relation representation for a $n \times n$ matrix and $C(ID, vw)$ be a relation representation for a vector with n elements. To support matrix-matrix multiplication (Eq. (1)) and matrix-vector multiplication (Eq. (2)), we define two aggregate-joins, namely, MM-join and MV-join. The first aggregate-join, called MM-join, is used to join two matrix relations A and B, to compute $\mathsf{A} \cdot \mathsf{B}$. The MM-join is denoted as $A \overset{\oplus(\odot)}{\underset{A.T=B.F}{\bowtie}} B$, and it is defined by the following relational algebra.

$$A \overset{\oplus(\odot)}{\underset{A.T=B.F}{\bowtie}} B = {}_{A.F,B.T}\,\mathcal{G}_{\oplus(\odot)}(A \underset{A.T=B.F}{\bowtie} B) \tag{3}$$

The second aggregate-join, called MV-join, is used to join a matrix relation and a vector relation, A and C, to compute $\mathsf{A} \cdot \mathsf{C}$. The MV-join is denoted as $A \overset{\oplus(\odot)}{\underset{T=ID}{\bowtie}} C$, and it is defined by the following relational algebra.

$$A \overset{\oplus(\odot)}{\underset{T=ID}{\bowtie}} C = {}_{F}\,\mathcal{G}_{\oplus(\odot)}(A \underset{T=ID}{\bowtie} C) \tag{4}$$

Here, ${}_{X}\mathcal{G}_{Y}$ is a group-by & aggregation operation to compute the aggregate function defined by Y over the groups by the attributes specified in X. Note that MV-join is discussed in [49, 36], and MM-join is similar to MV-join. There are two steps to compute MM-join. The first step is to join A and B by the join condition

$A.T = B.F$. This step is to join the k value in order to compute $\odot$ for $\mathsf{A}_{ik} \odot \mathsf{B}_{kj}$ as given in Eq. (1). The second step is to do group-by & aggregation, where the group-by attributes are the attributes that are in the primary key but do not appear in the join condition, namely, $A.F$ and $B.T$, and the aggregate function is to compute Eq. (1). In a similar fashion, there are two steps to compute MV-join. The first step is to join A and C by the join condition $A.T = C.ID$. This step is to join the k value in order to compute $\odot$ for $\mathsf{A}_{ik} \odot \mathsf{C}_k$ as given in Eq. (2). The second step is to do group-by & aggregation, where the group-by attribute is the attribute $A.F$, and the aggregate function is to compute Eq. (2).

We adopt the anti-join, $R \bar{\ltimes} S$, which is defined as the result of R that cannot be semi-joined by S, such that $R - (R \ltimes S)$. In addition, we propose a new union operation, called union-by-update, for the purpose of updating values in either a matrix or a vector, denoted as $\uplus$. Let $R(A, B)$ and $S(A, B)$ be two relations, where A and B are two sets of attributes. $R \uplus_A S$ is a relation, $RS(A, B)$. Let r be a tuple in R and s be a tuple in S. Different from the conventional union operation ($\cup$) where two tuples are identical if $r = s$, with $R \uplus_A S$, two tuples, r and s, are identical if $r.A = s.A$. The union-by-update is to update the B attributes values of r by the B attributes values of s if $r.A = s.A$. In other words, if $r.A = s.A$, then s is in RS but not r. There are 2 cases that r and s do not match. If s does not match any r, then s is in RS. If r does not match any s, then r is in RS. It is worth noting that there can be multiple r match multiple s on the attributes A. We allow multiple r to match a single tuple s, but we do not allow multiple s to match a single r, since the answer is not unique. When A attributes in both R and S are defined as the primary key, there is at most one pair of r and s matches.

The 4 operations are independent among themselves, since we can discover a property that is possessed by one operation but is not possessed by the composition of the other three only [66].

4.2 Relational Algebra plus While

To support graph processing, a control structure is needed in addition to the relational algebra operations discussed. We follow the "algebra + while" given in [12].

```
initialize $R$
**while** ($R$ changes) { $\cdots$; $R \leftarrow \cdots$ }
```

In brief, in the looping, R may change by the relational algebra in the body of the looping. The looping will terminate until R becomes stable. As discussed in [12], there are two semantics for "algebra + while", namely, noninflationary and inflationary. Consider the assignment, $R \leftarrow \mathcal{E}$, which is to assign relation R by evaluating the relational algebra expression $\mathcal{E}$. By the noninflationary, the assignment can be destructive in the sense that the new value will overwrite the old value. By the inflationary, the assignment needs to be cumulative. For the termination of the looping, as pointed out in [12], explicit terminating condition does not affect the expressive power. In this work, the conventional union ($\cup$) is for the inflationary semantics, whereas union-by-update ($\uplus$) is for the noninflationary semantics.

In this work, the expressive power, and the complexity remain unchanged as given in [12] under the scheme of "algebra + while", because the 4 operations added can be supported by the existing relational algebra operations. From the viewpoint of relational algebra, we can support all basic graph algorithms, including those that need aggregation (Table 2) but excluding those complicated algorithms for spectral analytics that need matrix inverse. The 4 operations make it clear how to support graph algorithms in relational algebra. In particular, all graph algorithms, that can be expressed by the semiring, can be supported under the framework of "algebra + while" and hence *SQL* recursion enhanced.

4.3 Supporting Graph Processing

We show how to support graph algorithms by the "algebra + while" approach, using MM-join and MV-join, anti-join, union-by-update, as well as other operations given in relational algebra. For simplicity, we represent a graph $G = (V, E)$ with n nodes by an $n \times n$ E and a vector V with n elements. We represent the vector V by a relation $V(ID, vw)$, where ID is the tuple identifier for the corresponding node with value vw associated. Moreover, we represent the matrix E by a relation $E(F, T, ew)$, where F and T, standing for "From" and "To", form a primary key in E, which is associated with an edge value ew. Below, to emphasize the operations in every iteration, we omit the while looping. Note that some graph algorithms can be computed by either union or union-by-update.

We use the examples of *PageRank*, *Floyd-Warshall* and *TopoSort* to illustrate how to support graph algorithms by the new algebra. The relational algebra for *PageRank* is given below.

$$V \leftarrow \rho_V(E \overset{f_1(\cdot)}{\underset{T=ID}{\bowtie}} V) \tag{5}$$

Here, $f_1(\cdot)$ is a function to calculate $c * sum(vw * ew) + (1 - c)/n$, where c is the damping factor and n is the total number of tuples in V. Note, $vw * ew$ is computed when joining the tuples from E and V, regarding $\odot$, and the aggregate function sum is computed over groups, given $vw * ew$ computed, along with other variables in $f_1(\cdot)$, regarding $\oplus$.

To implement *Floyd-Warshall*, the relational algebra is given as follows.

$$E \leftarrow \rho_E((E \rightarrow E_1) \overset{min(E_1.ew+E_2.ew)}{\underset{E_1.T=E_2.F}{\bowtie}} (E \rightarrow E_2)) \tag{6}$$

In Eq (6), we use one MM-join with $+$ and min serving as the $\odot$ and $\oplus$ respectively. For *PageRank* and *Floyd-Warshall*, one union-by-update operation is performed to update recursive relation V and E.

Below, we show how to support *TopoSort* (Topological Sorting) for *DAG* (Directed Acyclic Graph) using anti-join. To compute *TopoSort*, we assign a level L value to every node. For two nodes, u and v, if $u.L < v.L$, then $u < v$ in the *TopoSort*; if $u.L = v.L$, then either $u < v$ or $v < u$ is fine, since the *TopoSort* is not unique. Let $Topo(ID, L)$ be a relation that contains a set of nodes having no incoming edges with initial L value 0. The initial $Topo$ can be generated by $\Pi_{ID,0}(V \bar{\ltimes}_{ID=E.T} E)$. In the recursive part, it is done by several steps.

$$\begin{array}{rcl} L_n & \leftarrow & \rho_L(\mathcal{G}_{max(L)+1} Topo) \\ V_1 & \leftarrow & V \underset{V.ID=T.ID}{\bar{\ltimes}} Topo \\ E_1 & \leftarrow & \Pi_{E.F,E.T}(V_1 \underset{ID=E.F}{\bowtie} E) \\ T_n & \leftarrow & \Pi_{ID,L}(V_1 \underset{V_1.ID=E_1.T}{\bar{\ltimes}} E_1) \times L_n \\ Topo & \leftarrow & Topo \cup T_n \end{array} \tag{7}$$

Here, first, we compute the L value to be used for the current iteration, which is the max L value used in the previous iteration plus one. It is stored in L_n. Next, we remove those nodes that have already been sorted by anti-join and obtain V_1. With $V_1 \subseteq V$, we obtain the edges among nodes in V_1 as E_1. T_n is the set of nodes that are sorted in the current iteration. Finally, we get the new $Topo$ by union of the previous $Topo$ and the newly sorted T_n. It repeats until T_n is empty.

Table 2 shows some representative graph algorithms that can be supported by the 4 operations including MM-join, MV-join, anti-join and union-by-update. As a summary, MV-join together with union-by-update can be used to implement *PageRank*, weakly *Connected-Component*, *HITS*, *Label-Propagation*, *Keyword-Search* and *K-core*, whereas MM-join together with union-by-update can be used to support *Floyd-Warshall*, *SimRank* and *Markov-Clustering*. The anti-join serves as a selection to filter nodes/edges which are unnecessary in the following iterations. It is important to note that anti-join is not only for efficiency but also for the correctness. Equipped with anti-join, *TopoSort* is easy to be implemented. The combination of MV-join and anti-join support *Maximal-Independent-Set* and *Maximal-Node-Matching*.

Graph Algorithm	MV/MM-join	$\uplus/\cup$	anti-join	linear	nonlinear
TC [20]	–	$\cup$		✓	✓
BFS [20]	MV	$\uplus$		✓	
Connected-Component [57]	MV	$\uplus$		✓	
Bellman-Ford [20]	MV	$\uplus$		✓	
Floyd-Warshall [20]	MM	$\uplus$			✓
PageRank [42]	MV	$\uplus$		✓	
Random-Walk-with-Restart [42]	MV	$\uplus$		✓	
SimRank [31]	MM	$\uplus$		✓	
HITS [42]	MV	$\uplus$			✓
TopoSort [34]	–	$\cup$	✓		✓
Keyword-Search [16]	MV	$\uplus$		✓	
Label-Propagation [55]	MV	$\uplus$		✓	
Maximal-Independent-Set [46]	–	$\cup$	✓		✓
Maximal-Node-Matching [52]	MV	$\cup$	✓		✓
Diameter-Estimation [35]	–	$\cup$		✓	
Markov-Clustering [67]	MM	$\uplus$			✓
K*-core* [43]	MV	$\uplus$			✓
K*-truss* [54]	–	$\uplus$			✓
Graph-Bisimulation [30]	–	$\cup$			✓

Table 2: Graph Algorithms

5 XY-Stratified

As discussed in Section 3, *SQL*'99 supports stratified negation in recursion, which means it is impossible to support graph processing that needs the functions beyond stratified negation. Recall that the 4 operations are not monotone and are not stratified negation. To address this issue, we discuss relational algebra operations in the context of DATALOG using rules. The rules for selection, projection, Cartesian product, union, θ-join are given in [66]. In a similar way, we can express the 4 new operators using rules. As union, selection, projection, Cartesian product and θ-joins are monotone, recursive queries using such operations are stratified. But, MM-join, MV-join, anti-join, and union-by-update are not monotonic. The approach we take is based on XY-stratification [71, 72, 14]. An XY-stratified program is a special class of locally stratified programs [27]. As proposed by Zaniolo et al. in [71], it is a syntactically decidable subclass for non-monotonic recursive programs to handle negation and aggregation, and it captures the expressive power of inflationary fixpoint semantics [12]. An XY-program is a locally stratified DATALOG program that can be checked at compile-time by syntax.

Based on XY-stratification, we extend the with clause in *SQL*'99, "**with** R **as** $\langle$ R initialization $\rangle$ $\langle$ recursive querying involving R $\rangle$", to support a class of recursive query that can be used to support many graph analytical tasks. To minimize such extension, we restrict that the with clause only has one recursive relation R, and there is only one cycle in the corresponding dependency graph. The extension is to allow negation as well as aggregation in a certain form for a recursive query. We show that the recursive queries using the 4 operations discussed can have a fixpoint by which a unique answer can be obtained [74].

6 Conclusion

To support a large pool of graph algorithms, we propose 4 operations, namely, MM-join, MV-join, anti-join and union-by-update, that can be supported by the basic relational algebra operations, with group-by & aggregation. Among the 4 operations, union-by-update plays an important role in allowing value updates in iterations. The 4 non-monotonic operations are not allowed to be used in a recursive query as specified by *SQL*'99. We show that the 4 operations proposed together with others have a fixpoint semantics based on its limited form, based on DATALOG techniques. In other words, a fixpoint exists for the 4 operations that deal with negation and aggregation. We enhance the recursive with clause in *SQL* and translate the enhanced recursive with into *SQL/PSM* to be processed in *RDBMS*s. In [74], we conduct extensive performance studies to test 10 graph algorithms using 9 large real graphs on 3 major *RDBMS*s (*Oracle*, *DB2*, and *PostgreSQL*) at *SQL* level, and we show that *RDBMS*s are capable of dealing with graph processing in reasonable time. There is high potential to improve the efficiency by main-memory *RDBMS*s, efficient join processing in parallel, and new storage management.

ACKNOWLEDGES: This work was supported by the Research Grants Council of Hong Kong SAR, China, No. 14221716.

References

[1] http://giraph.apache.org.

[2] https://github.com/dato-code/PowerGraph.

[3] http://spark.apache.org/graphx/.

[4] http://neo4j.com.

[5] http://www.agensgraph.com/.

[6] http://thinkaurelius.github.io/titan/.

[7] IBM DB2 10.5 for linux, unix and windows documentation. http://www.ibm.com/support/knowledgecenter/#!/SSEPGG_10.5.0/com.ibm.db2.luw.kc.doc/welcome.html.

[8] Microsoft SQL documentation. https://docs.microsoft.com/en-us/sql/.

[9] Oracle database SQL language reference. http://docs.oracle.com/cd/E11882_01/server.112/e41084/toc.htm.

[10] Postgresql 9.4.7 documentation. http://www.postgresql.org/files/documentation/pdf/9.4/postgresql-9.4-A4.pdf.

[11] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *Proc. o SIGMOD'16*, 2016.

[12] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[13] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández. Formalizing a broader recursion coverage in SQL. In *Proc. of PADL'13*, 2013.

[14] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system LDL++. *TPLP*, 3(1), 2003.

[15] P. Barceló. Querying graph databases. In *Proc. of PODS'13*, 2013.

[16] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of ICDE'02*, 2002.

[17] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[18] W. Cabrera and C. Ordonez. Unified algorithm to solve several graph problems with relational queries. In *Proc of AMW'16*, 2016.

[19] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *PVLDB*, 2(2), 2009.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 3 edition, 2009.

[21] J. Fan, A. Gerald, S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *Proc. of CIDR'15*, 2015.

[22] S. J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. *ISO-IEC JTC1/SC21 WG3 DBL MCI*, (X3H2-96-075), 1996.

[23] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational approach for shortest path discovery over large graphs. *PVLDB*, 5(4), 2011.

[24] J. Gao, J. Zhou, C. Zhou, and J. X. Yu. Glog: A high level graph analysis system using mapreduce. In *Proc. of ICDE'14*, 2014.

[25] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems The Complete Book*. Prentice Hall, 2002.

[26] A. Ghazal, D. Seid, A. Crolotte, and M. Al-Kateb. Adaptive optimizations of recursive queries in teradata. In *Proc. of SIGMOD'12*, 2012.

[27] S. Greco and C. Molinaro. *Datalog and Logic Databases*. Morgan & Claypool Publishers, 2015.

[28] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. of SIGMOD'08*, 2008.

[29] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *PVLDB*, 5(12), 2012.

[30] M. R. Henzinger, T. Henzinger, P. W. Kopke, et al. Computing simulations on finite and infinite graphs. In *Proc. of FOCS'95*, 1995.

[31] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *Proc. of SIGKDD'02*, 2002.

[32] A. Jindal and S. Madden. Graphiql: A graph intuitive query language for relational databases. In *Proc. of BigData'14*, 2014.

[33] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using the vertica relational database. *arXiv preprint arXiv:1412.5263*, 2014.

[34] A. B. Kahn. Topological sorting of large networks. *CACM*, 5(11), 1962.

[35] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2), 2011.

[36] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: mining peta-scale graphs. *Knowledge and information systems*, (2), 2011.

[37] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[38] L. Libkin, W. Martens, and D. Vrgoc. Querying graphs with data. *J. ACM*, 63(2), 2016.

[39] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer. Fast in-memory SQL analytics on typed graphs. *PVLDB*, pages 265–276, 2016.

[40] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang. G-SQL: fast query processing via graph exploration. *PVLDB*, pages 900–911, 2016.

[41] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD'10*, 2010.

[42] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge University Press, 2008.

[43] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *JACM*, 30(3), 1983.

[44] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), 2015.

[45] J. Melton and A. R. Simon. *SQL: 1999: understanding relational language components*. Morgan Kaufmann, 2001.

[46] Y. Métivier, J. M. Robson, N. Saheb-Djahromi, and A. Zemmari. An optimal bit complexity randomized distributed MIS algorithm. *Distributed Computing*, 23(5-6), 2011.

[47] V. Z. Moffitt and J. Stoyanovich. Towards a distributed infrastructure for evolving graph analytics. In *Proc. of WWW'16 Companion Volume*, 2016.

[48] W. E. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch. Datalography: Scaling datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data*.

[49] C. Ordonez. Optimization of linear recursive queries in sql. *TKDE*, 22(2), 2010.

[50] C. Ordonez, W. Cabrera, and A. Gurram. Comparing columnar, row and array dbmss to process recursive queries on graphs. *Information Systems*, 63, 2017.

[51] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. SQL- and operator-centric data analytics in relational main-memory databases. In *Proc. of EDBT 2017.*, pages 84–95, 2017.

[52] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Proc. of STACS'99*, 1999.

[53] P. Przymus, A. Boniewicz, M. Burzańska, and K. Stencel. Recursive query facilities in relational databases: a survey. In *Proc. of DTA/BSBT'10*, 2010.

[54] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *Proc. of ASONAM'12*, 2012.

[55] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[56] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2), 1995.

[57] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Proc. of ICDE'13*, 2013.

[58] S. Salihoglu and J. Widom. Help: High-level primitives for large-scale graph processing. In *Proc. of Workshop on GRAph Data management Experiences and Systems*, 2014.

[59] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *Proc. of ICDE'13*, 2013.

[60] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14), 2013.

[61] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *Proc. of SIGMOD'16*, 2016.

[62] A. Shkapsky, M. Yang, and C. Zaniolo. Optimizing recursive queries with monotonic aggregates in DeALS. In *Proc. of ICDE'15*, 2015.

[63] A. Shkapsky, K. Zeng, and C. Zaniolo. Graph queries in a next-generation datalog system. *PVLDB*, 6(12), 2013.

[64] S. Srihari, S. Chandrashekar, and S. Parthasarathy. A framework for sql-based mining of large graphs on relational databases. In *Proc. of PAKDD'10*, 2010.

[65] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *Proc. of SIGMOD'15*, 2015.

[66] J. D. Ullman. *Principles of Database and Knowledge Base Systems (Vol I)*. Computer Science Press, 1988.

[67] S. M. van Dongen. Graph clustering by flow simulation. *PhD Thesis, University of Utrecht*, 2000.

[68] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *Proc. of GRADES'16*, 2016.

[69] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1), 2012.

[70] K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: exploring interesting graphs in relational data. *PVLDB*, 8(12), 2015.

[71] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *Proc. of DOOD*, 1993.

[72] C. Zaniolo, S. Stefano, Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced database systems*. Morgan Kaufmann, 1997.

[73] Y. Zhang, M. Kersten, and S. Manegold. Sciql: Array data processing inside an rdbms. In *Proc. of SIGMOD'13*, 2013.

[74] K. Zhao and J. X. Yu. All-in-one: Graph processing in rdbmss revisited. In *Proc. of SIGMOD'17*, 2017.

Trinity Graph Engine and its Applications

Bin Shao, Yatao Li, Haixun Wang[*], Huanhuan Xia
Micorosoft Research Asia, [*]Facebook
{binshao, yatli, lexi}@microsoft.com, [*]haixun@gmail.com

Abstract

Big data become increasingly connected along with the rapid growth in data volume. Connected data are naturally represented as graphs and they play an indispensable role in a wide range of application domains. Graph processing at scale, however, is facing challenges at all levels, ranging from system architectures to programming models. Trinity Graph Engine is an open-source distributed in-memory data processing engine, underpinned by a strongly-typed in-memory key-value store and a general distributed computation engine. Trinity is designed as a general-purpose graph processing engine with a special focus on real-time large-scale graph query processing. Trinity excels at handling a massive number of in-memory objects and complex data with large and complex schemas. We use Trinity to serve real-time queries for many real-life big graphs such as Microsoft Knowledge Graph and Microsoft Academic Graph. In this paper, we present the system design of Trinity Graph Engine and its real-life applications.

1 Introduction

In this big data era, data become increasingly connected along with the rapid growth in data volume. The increasingly linked big data underpins artificial intelligence, which is expanding its application territory at an unprecedented rate. Linked data are naturally represented and stored as graphs. As a result graph data have now become ubiquitous thanks to web graphs, social networks, and various knowledge graphs, to name but a few.

Graph processing at scale, however, is facing challenges at all levels, ranging from system architectures to programming models. On the one hand, graph data are not special and can be processed by many data management or processing systems such as relational databases [1] and MapReduce systems [2]. On the other hand, large graph processing has some unique characteristics [3], which make the systems that do not respect them in their design suffer from the "curse of connectedness" when processing big graphs. In this paper, we discuss the challenges faced by real-time parallel large graph processing and how to rise to them in the system design.

The complex nature of graph. Graph data is inherently complex. The contemporary computer architectures are good at processing linear and simple hierarchical data structures, such as *Lists*, *Stacks*, or *Trees*. Even when the data scale becomes large and is partitioned over many distributed machines, the *divide and conquer*

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]This work was done in Microsoft Research Asia.

computation paradigm still works well for these data structures. However, when we are handling graphs, especially big graphs, the situation is changed. Big graphs are difficult to process largely because they have a large number of interconnected relations encoded. The implication is twofold: 1) From the perspective of data access, the adjacent nodes of a graph node cannot be accessed without "jumping" in the data store no matter how we represent a graph. In other words, a massive amount of random data access is required during graph processing. Many modern program optimizations rely on data reuse. Unfortunately, the random data access nature of graph processing breaks this premise. Without a careful system design, this usually leads to poor performance since the CPU cache is not in effect for most of the time. 2) From the perspective of programming, parallelism is difficult to extract because of the unstructured nature of graphs. As widely acknowledged [3], a lot of graph problems are inherently irregular and hard to partition; this makes it hard to obtain efficient *divide and conquer* solutions for many large graph processing tasks.

Due to the random data access challenge, general-purpose graph computations usually do not have efficient, disk-based solutions. But under certain constraints, graph problems sometimes can have efficient disk-based solutions. A good example is GraphChi [4]. GraphChi can perform efficient disk-based graph computations under the assumption that the computations have asynchronous vertex-centric [5] solutions. An asynchronous solution is one where a vertex can perform its computation based only on the partially updated information from its incoming graph edges. This assumption eliminates the requirement of global synchronization, making performing computations block by block possible. On the other hand, it inherently cannot support traversal-based graph computations and synchronous graph computations because a graph node cannot efficiently access the graph nodes pointed by its outgoing edges.

The diversity of graph data and graph computations. There are many kinds of graphs. Graph algorithms' performance may vary a lot on different types of graphs. On the other hand, there are a large variety of graph computations such as path finding, subgraph matching, community detection, and graph partitioning. Each graph computation itself even deserves dedicated research; it is nearly impossible to design a system that can support all kinds of graph computations. Moreover, graphs with billions of nodes are common now, for example, the Facebook social network has more than 2 billion monthly active users[1]. The scale of the data size makes graph processing prohibitive for many graph computation tasks if we directly apply the classic graph algorithms from textbooks.

In this paper, we present Trinity Graph Engine – a system designed to meet the above challenges. Instead of being optimized for certain types of graph computations on certain types of graphs, Trinity tries to directly address the grand random data access challenge at the infrastructure level. Trinity implements a globally addressable distributed RAM store and provides a random access abstraction for a variety of graph computations. Trinity itself is not a system that comes with comprehensive built-in graph computation modules. However, with its flexible data and computation modeling capability, Trinity can easily morph into a customized graph processing system that is optimized for processing a certain type of graphs.

Many applications utilize large RAM to offer better performance. Large web applications, such as Facebook, Twitter, Youtube, and Wikipedia, heavily use memcached [6] to cache large volumes of long-lived small objects. As the middle tier between data storage and application, caching systems offload the server side work by taking over some data serving tasks. However, the cache systems cannot perform in-place computations to further reduce computation latencies by fully utilizing the in-memory data.

The design of Trinity is based on the belief that, as high-speed network access becomes more available and DRAM prices trend downward, all-in-memory solutions provide the lowest total cost of ownership for a large range of applications [7]. For instance, RAMCloud [8] envisioned that advances in hardware and operating system technology will eventually enable all-in-memory applications, and low latency can be achieved by deploying faster network interface controllers (NICs) and network switches and by tuning the operating systems, the NICs, and the communication protocols (e.g., network stack bypassing). Trinity realizes this vision

[1]`http://newsroom.fb.com/company-info/.`

for large graph applications, and Trinity does not rely on hardware/platform upgrades and/or special operating system tuning, although Trinity can leverage these techniques to achieve even better performance.

The rest of the paper is organized as follows. Section 2 outlines the design of the Trinity system. Section 3 introduces Trinity's distributed storage infrastructure – Memory Cloud. Section 4 introduces Trinity Specification Language. Section 5 discusses fault tolerance issues. Section 6 introduces Trinity applications. Section 7 concludes.

2 An Overview of Trinity

Trinity is a data processing engine on distributed in-memory infrastructure called Trinity Memory Cloud. Trinity organizes the main memory of multiple machines into a globally addressable memory address space. Through the memory cloud, Trinity enables fast random data access over a large distributed data set. At the same time, Trinity is a versatile computation engine powered by declarative message passing.

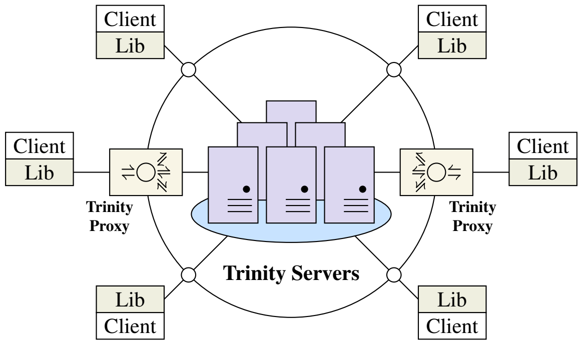


Figure 1: Trinity Cluster Structure

Fig. 1 shows the architecture of Trinity. A Trinity system consists of multiple components that communicate through a network. According to the roles they play, we classify them into three types: servers, proxies, and clients. A Trinity server plays two roles: storing data and performing computations on the data. Computations usually involve sending messages to and receiving messages from other Trinity components. Specifically, each server stores a portion of the data and processes messages received from other servers, proxies, or clients. A Trinity proxy only handles messages but does not own a data partition. It usually serves as a middle tier between servers and clients. For example, a proxy may serve as an information aggregator: it dispatches the requests coming from clients to servers and sends the results back to the clients after aggregating the partial results received from servers. Proxies are optional, that is, a Trinity system does not always need a proxy. A Trinity client is responsible for interacting with the Trinity cluster. Trinity clients are applications that are linked to the Trinity library. They communicate with Trinity servers and proxies through APIs provided by Trinity.

Fig. 2 shows the stack of Trinity system modules. The memory cloud is essentially a distributed key-value store underpinned by a strongly-typed RAM store and a general distributed computation engine. The RAM store manages memory and provides mechanisms for concurrency control. The computation engine provides an efficient, one-sided, machine-to-machine message passing infrastructure.

Due to the diversity of graphs and the diversity of graph applications, it is hard, if not entirely impossible, to support all kinds of graph computations using a fixed graph schema. Instead of using a fixed graph schema and fixed computation paradigms, Trinity allows users to define their own graph schemas, communication protocols through Trinity specification language (TSL) and realize their own computation paradigms. TSL bridges the needs of a specific graph application with the common storage and computation infrastructure of Trinity.

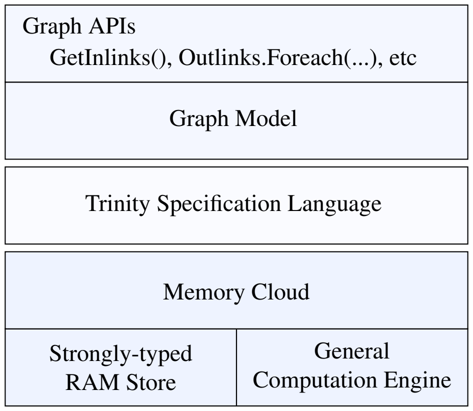

Graph APIs
GetInlinks(), Outlinks.Foreach(...), etc
Graph Model
Trinity Specification Language
Memory Cloud
Strongly-typed RAM Store
General Computation Engine

Figure 2: System Layers

3 Trinity Memory Cloud

We build a distributed RAM store – Trinity Memory Cloud – as Trinity's storage and computation infrastructure. The memory cloud consists of 2^p memory trunks, each of which is stored on one machine. Usually, we have $2^p > m$, where m is the number of machines. In other words, each machine hosts multiple memory trunks. We partition a machine's local memory space into multiple memory trunks so that trunk level parallelism can be achieved without any locking overhead. To support fault-tolerant data persistence, these memory trunks are backed up in a shared distributed file system called TFS (Trinity File System) [9], whose design is similar to that of HDFS [10].

We create a key-value store in the memory cloud. A key-value pair forms the most basic data structure of any system built on top of the memory cloud. Here, keys are 64-bit globally unique integer identifiers; values are blobs of arbitrary length. Because the memory cloud is distributed across multiple machines, we cannot address a key-value pair using its physical memory address. To address a key-value pair, Trinity uses a hashing mechanism. In order to locate the value of a given key, we first 1) identify the machine that stores the key-value pair, then 2) locate the key-value pair in one of the memory trunks on that machine. Through this hashing mechanism as illustrated by Figure 3, we provide a globally addressable memory space.

Specifically, given a 64-bit key, to locate its corresponding value in the memory cloud, we hash the key to a p-bit value i ($i \in [0, 2^p - 1]$), indicating that the key-value pair is stored in memory trunk i within the memory cloud. Trinity assigns a unique machine identifier *mid* to each machine in the Trinity cluster. To find out which machine contains memory trunk i, we maintain an "addressing table" with 2^p slots, where each slot stores a *mid* with which we can reach the corresponding Trinity server. Furthermore, in order for the global addressing to work, each machine keeps a replica of the addressing table. We will describe how we ensure the consistency of these addressing tables in Section 5.

We then locate the key-value pair in the memory trunk i. Each memory trunk is associated with a latch-free hash table on the machine whose *mid* is in the slot i of the addressing table. We hash the 64-bit key again to find the offset and size of the stored blob (the value part of the key-value pair) in the hash table. Given the memory offset and the size, we now can retrieve the key-value pair from the memory trunk.

The addressing table provides a mechanism that allows machines to dynamically join and leave the memory cloud. When a machine fails, we reload the memory trunks it owns from the TFS to other alive machines. All we need to do is to update the addressing table so that the corresponding slots point to the machines that host the data now. Similarly, when new machines join the memory cloud, we relocate some memory trunks to those new machines and update the addressing table accordingly.

Each key-value pair in the memory cloud may attach some metadata for a variety of purposes. Most notably, we associate each key-value pair with a spin lock. Spin locks are used for concurrency control and physical

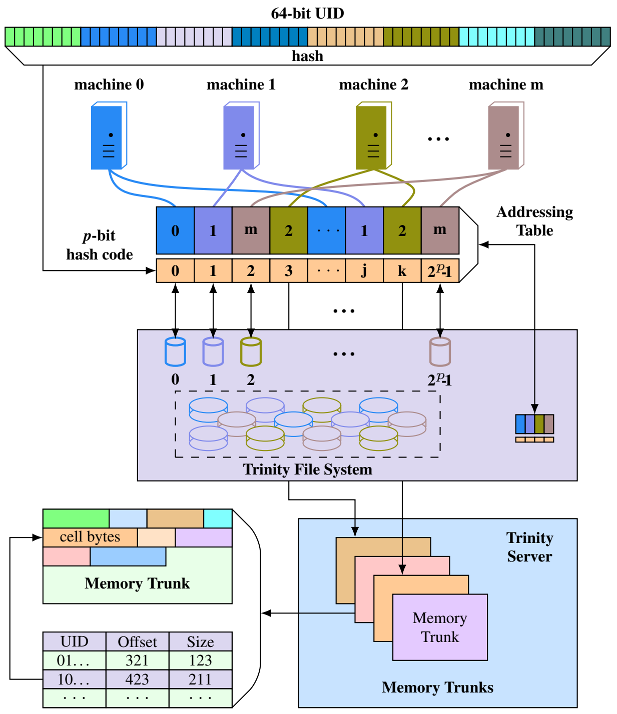


Figure 3: Data Partitioning and Addressing

memory pinning. Multiple threads may try to access the same key-value pair concurrently; we must ensure a key-value pair is locked and pinned to a fixed memory position before allowing any thread to manipulate it. In Trinity, all threads need to acquire the corresponding spin lock before it can access a key-value pair exclusively.

4 Trinity Specification Language

In this section, we introduce Trinity Specification Language (TSL). It is a declarative language designed to specify data schemas and message passing protocols using *cell* and *protocol* constructs. The TSL compiler is essentially a code generator: it generates optimized data access methods and message passing code according to the specificed TSL script.

4.1 Strongly-typed Data Modeling

Trinity supports property graphs[2] on top of its key-value store. "Keys" are globally unique identifiers introduced in Section 3 and their "values" are used for storing application data. The schema of the value part of a key-value pair can be specified by a *cell* structure in a TSL script. A *cell* structure in a TSL script specifies a user-defined data type. Defining a *cell* is pretty much like defining a struct in C/C++ as shown in Figure 4. The *value* part of such a key-value pair stored in Trinity memory cloud is called a *data cell* or simply *cell* when there is no ambiguity. Correspondingly, the *key* of the key-value pair is called its *cell Id*.

The TSL snippet shown in Figure 4 demonstrates how to model a graph node using a *cell* structure. A graph node represented by a *cell* and a *cell* can be referenced by its 64-bit *cell Id*, thus simple graph edges which reference a list of graph nodes can be represented by *List<int64>*. The data schema of graph edges that have associated data can be specified using a TSL *struct*. In the example shown in Figure 4, the schema of *MyEdges* is specified by the *MyEdge* struct.

[2]The nodes and edges of a property graph can have rich information associated.

```
struct MyEdge
{
    int64 Link;
    float Weight;
}
[GraphNode]
cell MyGraphNode
{
    string Name;
    [GraphEdge: Inlinks]
    List<int64> SimpleEdges;
    [GraphEdge: Outlinks]
    List<MyEdge> MyEdges;
}
```

Figure 4: Modeling a Graph Node

To distinguish the cell fields that specify graph edges from those that do not, we can annotate a cell and its data fields using TSL *attributes*. An attribute is a tag associated with a construct in TSL. Attributes provide the metadata about the construct and can be accessed during run time. An attribute can be a string or a pair of strings. An attribute is always regarded as a key-value pair. A single-string attribute is regarded as a key-value pair with an empty value. In the example shown in Figure 4, we use attribute *GraphNode* to indicate the cell *MyGraphNode* is a graph node and use attribute *GraphEdge* to indidate *SimpleEdges* and *MyEdges* are graph edges.

4.2 Modeling Computation Protocols

Trinity realizes a communication architecture called active messages [11] to support fine-grained one-sided communication. This communication architecture is desirable for data-driven computations and especially suitable for online graph query processing, which is sensitive to network latencies. TSL provides an intuitive way of writing such message passing programs.

```
struct MyMessage
{
    string Text;
}
protocol Echo
{
    Type: Syn;
    Request: MyMessage;
    Response: MyMessage;
}
```

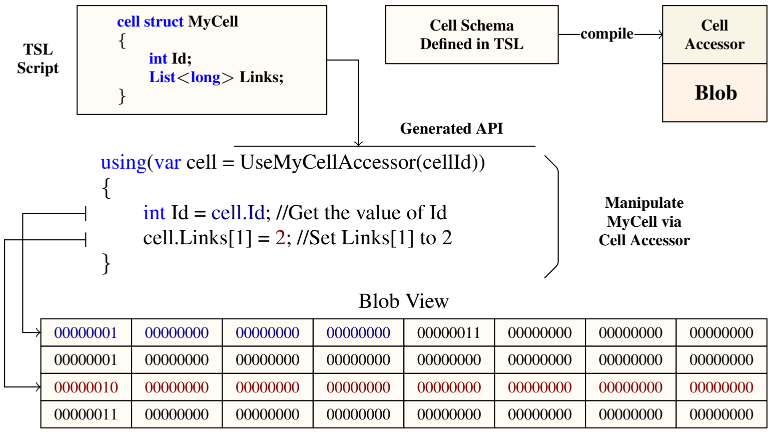
Figure 5: Modeling Message Passing

Fig. 5 shows an example. It specifies a simple "Echo" protocol: A client sends a message to a server, and the server simply sends the message back. The "Echo" protocol specifies its communication type is synchronous message passing, and the type of the messages to be sent and received is *MyMessage*. For this TSL script, the TSL compiler will generate an empty message handler *EchoHandler* and the user can implement the message handling logic for the handler. Calling a protocol defined in the TSL is like calling an ordinary local method. Trinity takes care of message dispatching, packing, etc., for the user.

4.3 Zero-copy Cell Manipulation

Trinity memory cloud provides a key-value pair store, where the values are binary blobs whose data schemas are specified via TSL. Alternatively, we can store graph nodes and edges as the runtime objects of an object-oriented programming language. Unfortunately, this is not a feasible approach for the following three reasons. First, we cannot reference these runtime objects across machine boundaries. Second, runtime objects incur significant storage overhead. For example, an empty runtime object (one that does not contain any data at all) in .Net Framwork requires 24 bytes of memory on a 64-bit system and 12 bytes of memory on a 32-bit system. For a billion-node graph, this is a big overhead. Third, although Trinity is an in-memory system, we do need to store memory trunks on the disk or over a network for persistence. For runtime objects, we need serialization and deserialization operations, which are costly.

Storing objects as blobs of bytes seems to be desirable since they are compact and economical with zero serialization and deserialization overhead. We can also make the objects globally addressable by giving them unique identifiers and using hash functions to map the objects to memory in a machine. However, blobs are not user-friendly. We no longer have object-oriented data manipulation interfaces; we need to know the exact memory layout before we can manipulate the data stored in the blob (using pointers, address offsets, and casting to access data elements in the blob). This makes programming difficult and error-prone[3].


TSL Script
Cell Schema Defined in TSL
compile
Cell Accessor
Blob
Generated API
Manipulate MyCell via Cell Accessor
Blob View

```
cell struct MyCell
{
    int Id;
    List<long> Links;
}
```

```
using(var cell = UseMyCellAccessor(cellId))
{
    int Id = cell.Id; //Get the value of Id
    cell.Links[1] = 2; //Set Links[1] to 2
}
```

00000001	00000000	00000000	00000000	00000011	00000000	00000000	00000000
00000001	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000010	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000011	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 6: Cell Accessor

To address this problem, Trinity introduces a mechanism called *cell accessor* to support object-oriented data manipulation on blob data. Users first declare the schema of a cell in TSL, then the TSL compiler automatically generates data access methods for manipulating cells stored as blobs in the memory cloud. One of the generated function is *UseMyCellAccessor*. Given a cellId, it returns an object of type *MyCellAccessor*. With the generated *MyCellAccessor*, users can manipulate its underlying blob data in an object-oriented manner as shown in Figure 6.

As a matter of fact, a cell accessor is not a data container, but a data mapper. It maps the data fields declared in TSL to the correct memory locations in the blob. Data access operations to a data field will be correctly mapped to the correct memory locations with zero memory copy overhead. In addition, using the spin lock associated with each key-value pair, Trinity guarantees the atomicity of the operations on a single data cell when the cell is manipulated via its *cell accessor*. However, Trinity does not provide built-in ACID transaction support. This means Trinity cannot guarantee serializability for concurrent threads. For applications that need transaction support, users can implement light-weight atomic operation primitives that span multiple cells, such as MultiOp primitives [12] or Mini-transaction primitives [13] on top of the atomic cell operations provided by

[3]Note that we cannot naively cast a blob to a structure defined in programming languages such as C or C++ because the data elements of a struct are not always flatly laid out in the memory. We cannot cast a flat memory region to a structured data pointer.

cell accessor.

5 Fault Tolerance

As a distributed in-memory system, Trinity needs to deal with subtle fault-tolerance issues. The fault-tolerance requirements are highly application dependent, we discuss the general fault tolerance approaches in this section and application developers should choose proper fault tolerance approaches according to their application needs.

5.1 Shared Addressing Table Maintenance

Trinity uses a shared addressing table to locate key-value pairs, as elaborated in Section 3. The addressing table is a shared data structure. A centralized implementation is unfeasible because of the performance bottleneck and the risk of single points of failure. A straightforward approach to these issues is to duplicate this table on each server. However, this leads to the potential problem of data inconsistency.

Trinity maintains a primary replica of the shared addressing table on a leader machine and uses the fault-tolerant Trinity File System to keep a persistent copy of the primary addressing table. An update to the primary table must be applied to the persistent replica before being committed.

Both ordinary messages and heartbeat messages are used to detect machine failures. For example, if machine A attempts to access a data item on machine B that is down, machine A can detect the failure of machine B. In this case, machine A informs the leader machine of the failure of machine B. Afterwards, machine A waits for the addressing table to be updated and attempts to access the item again once the addressing table is updated. In addition, machine-to-machine heartbeat messages are sent periodically to detect network or machine failures.

Upon confirmation of a machine failure, the leader machine starts a recovery process. During recovery, the leader reloads the data owned by the failed machine to other alive machines, updates the primary addressing table, and broadcasts it. Even if some machines cannot receive the broadcast message due to a temporary network problem, the protocol still works since a machine always syncs up with the primary addressing table replica when it fails to load a data item. If the leader machine fails, a new round of leader election will be triggered. The new leader sets a flag on the shared distributed file system to avoid multiple leaders in the case that the cluster machines are partitioned into disjointed sets due to network failures.

5.2 Fault Recovery

We provide different fault recovery mechanisms for different computation paradigms. Checkpoint and "periodical interruption" techniques are adopted for offline computations, while an asynchronous buffered logging mechanism is designed for online query processing.

5.2.1 Offline Computations

For BSP-based synchronous computations, we can make checkpoints every few super-steps [5]. These checkpoints are written to the Trinity File System for future recovery.

Because checkpoints cannot be easily created when the system is running in the asynchronous mode, the fault recovery is subtler than that of its synchronous counterpart. Instead of adopting a complex checkpoint technique such as the one described in [14], we use a simple "periodical interruption" mechanism to create snapshots. Specifically, we issue interruption signals periodically. Upon receiving an interruption signal, all servers pause after finishing their current job. After issuing the interruption signal, Safra's termination detection algorithm [15] is employed to check whether the system has ceased. A snapshot is then written to the Trinity File System once the system ceases.

5.2.2 Online Query Processing

We now discuss how to guarantee that any user query that has been successfully submitted to the system will be eventually processed, despite machine failures.

The fault recovery of online queries consists of three steps:

1. Log submitted queries.
2. Log all generated cell operations during the query processing,
3. Redo queries by replaying logged cell operations.

Query Logging We must log a user query for future recovery after it is submitted to the system. To reduce the logging latency, a buffered logging mechanism [16] is adopted. Specifically, when a user query is issued, the Trinity client submits it to at least two servers/proxies. The client is acknowledged only after the query replicas are logged in the remote memory buffers. Meanwhile, we use a background thread to asynchronously flush the in-memory logs to the Trinity File System. Once a query is submitted, we choose one of the Trinity servers/proxies that have logged the query to be the *agent* of this query. The query *agent* is responsible for issuing the query to the Trinity cluster on behalf of the client. The agent assigns a globally unique *query id* to each query once the query is successfully submitted.

For the applications that only have read-only queries, we just need to reload the data from the Trinity File System and redo the failed queries when a machine fails. For the applications that support online updates, a carefully designed logging mechanism is needed.

Operation Logging A query during its execution generates a number of messages that will be passed in the cluster. An update query transforms a set of data cells from their current states to a set of new states through the pre-defined message handlers. All message handlers are required to be deterministic[4] so that we can recover the system state from failures by replaying the logged operations. The resulting states of the data cells are determined by the initial cell states and the generated messages.

We designed an asynchronous buffered logging mechanism for handling queries with update operations. We keep track of all generated cell operations and asynchronously log operations to remote memory buffers. Each log entry is represented by a tuple $< qid, cid, m, sn >$, which consists of a query id qid , a cell id cid, a message m, and a sequential number sn. The query id, cell id, and the message that has trigged the current cell operation uniquely specifies a cell operation. The messages are logged to distinguish the cell operations generated by the same query. The sequential number indicates how many cell operations have been applied to the current cell. It represents the cell state in which the cell operation was generated.

we enforce a sequential execution order for each cell using the spin lock associated with the cell. We can safely determine a cell's current sequential number when it holds the spin lock. We send a log entry to at least two servers/proxies once it acquires the lock. We call asynchronous logging *weak logging* and synchronous logging *strong logging*. When a cell operation is acknowledged by all remote loggers, its state becomes *strongly logged*. We allow *weak logging* if all the cell operations on the current cell with sequential numbers less than *sn* $- \alpha$ ($\alpha \geq 1$) have been strongly logged, where *sn* is the sequential number of the current cell and α is a configurable parameter. Otherwise, we have to wait until the operations with sequential numbers less than *sn* $- \alpha$ have been strongly logged. There are two implications: 1) The strongly logged operations are guaranteed to be consecutive. This guarantees all strongly logged operations can be replayed in the future recovery. 2) Introducing a *weak logging* window of size α reduces the chance of blocking. Ideally, if the network round-trip latency is less than the execution time of α cell operations, then all cell operations can be executed without waiting for logging acknowledgements.

[4]Given a cell state, the resulting cell state of executing a message handler is deterministic.

System Recovery During system recovery, all servers enter a “frozen” state, in which message processing is suspended and all received messages are queued. In the “frozen” state, we reload the data from the Trinity File System and start to redo the logged queries. In this process, we can process these queries concurrently and replay the cell operations of different cells in parallel.

For any cell, the logged cell operations must be replayed in an order determined by their sequential numbers. For example, consider the following log snippet:

$$\cdots < q_1, c_1, m_{10}, 13 >, \cdots, < q_1, c_1, m_{11}, 15 > \cdots$$

For query q_1, after the log entry $< q_1, c_1, m_{10}, 13 >$ is replayed, the entry $< q_1, c_1, m_{11}, 15 >$ will be blocked until the sequential number c_1 is increased to 14 by another query.

Let us examine why we can restore each cell to the state before failures occur. Because all message handlers are deterministic, for a query, its output and the resulting cell states solely depend on its execution context, i.e., the cell states in which all its cell operations are executed. Thus, we can recover the system by replaying the logged cell operations on each cell in their execution order.

Since the system recovery needs to redo all the logged queries, we must keep the log size small to make the recovery process fast. We achieve this by incrementally applying the logged queries to the persistent data snapshot on the Trinity File System in the background.

6 Real-life Applications

Trinity is a Microsoft open source project on GitHub[5]. It has been used in many real-life applications, including knowledge bases [17], knowledge graphs [18], academic graphs[6], social networks [19], distributed subgraph matching [20], calculating shortest paths [21], and partitioning billion-node graphs [22]. More technical details and experimental evaluation results can be found in [23], [20], [18], [21], [22], [24], and [25].

In this section, we use a representative real-life application of Trinity as a case to study how to serve real-time queries for big graphs with complex schemas. The graph used in this case study is Microsoft Knowledge Graph[7] (MKG). MKG is a massive entity network; it consists of 2.4+ billion entities, 8.0+ billion entity properties, and 17.4+ billion relations between entities. Inside Microsoft, we have a cluster of Trinity servers serving graph queries such as path finding and subgraph matching in real time.

Designing a system to serve MKG faces a new challenge of large complex data schemas besides the general challenges of parallel large graph processing discussed in Section 1. Compared to typical social networks that tend to have a small number of entity types such as *person* and *post*, the real-world knowledge graph MKG has 1610 entity types and 5987 types of relationships between entities[8]. Figure 7 shows a small portion (about 1/120) of the MKG schema graph.

The large complex schemas of MKG makes it a challenging task to efficiently model and serve the knowledge graph. Thanks to the flexible design of Trinity Specification Language, Trinity has met the challenge in an ‘unusual’ but very effective way. With the powerful code generation capability of the TSL compiler, we can *beat the big schema with big code*! For MKG, the TSL compiler generated about 18.7 million lines of code for modeling the knowledge graph entities in an extremely fine-grained manner. With the generated fine-grained strongly-typed data access methods, Trinity provides very efficient random data access support for the graph query processor.

[5]https://github.com/Microsoft/GraphEngine

[6]https://azure.microsoft.com/en-us/services/cognitive-services/academic-knowledge/

[7]The knowledge graph is also known as Satori knowledge graph.

[8]The size of MKG is ever growing; this number is only for an MKG snapshot.

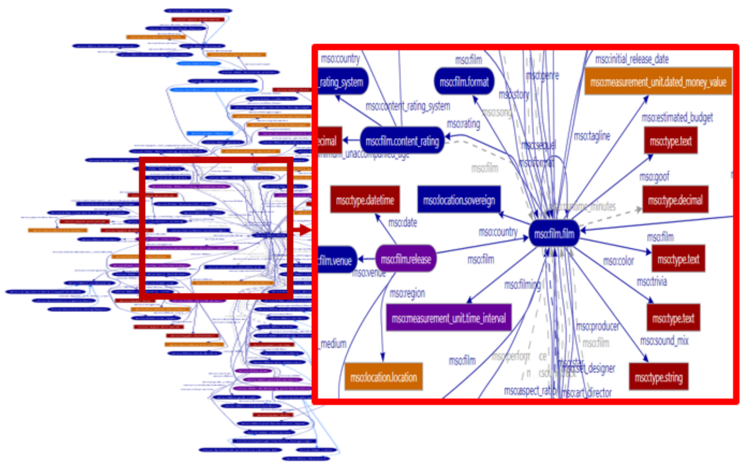


Figure 7: A small portion of the MKG schema graph

7 Conclusion

In this paper, we presented Trinity – a graph engine on a distributed in-memory infrastructure called Trinity Memory Cloud. Instead of being optimized for certain types of graph computations on certain types of graphs, Trinity is designed as a versatile "graph engine" to support a large variety of graph applications. Trinity now is a Microsoft open source project on GitHub and we have been using Trinity to support all kinds of graph applications including knowledge graphs, academic graphs, and social networks.

References

[1] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient rdf store over a relational database," in *SIGMOD*, 2013.

[2] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," ser. ICDM '09. IEEE Computer Society, 2009, pp. 229–238.

[3] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[4] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *OSDI*, 2012, pp. 31–46.

[5] 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," ser. SIGMOD '10. ACM.

[6] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, August 2004.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," ser. SOSP '09. ACM, pp. 1–14.

[8] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 92–105, 2010.

[9] J. Zhang and B. Shao, “Trinity file system specification,” Microsoft Research, Tech. Rep., 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/?id=201523

[10] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, 2007.

[11] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active messages: a mechanism for integrated communication and computation,” in *Proceedings of the 19th annual international symposium on Computer architecture*, ser. ISCA ’92. New York, NY, USA: ACM, 1992, pp. 256–266.

[12] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: an engineering perspective,” ser. PODC ’07, 2007, pp. 398–407.

[13] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, “Sinfonia: a new paradigm for building scalable distributed systems,” ser. SOSP ’07, 2007, pp. 159–174.

[14] H. Higaki, K. Shima, T. Tachikawa, and M. Takizawa, “Checkpoint and rollback in asynchronous distributed systems,” ser. INFOCOM ’97. IEEE Computer Society, 1997.

[15] E. W. Dijkstra, “Shmuel Safra’s version of termination detection,” Jan. 1987. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF

[16] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, “Fast crash recovery in ramcloud,” ser. SOSP ’11. ACM, 2011, pp. 29–41.

[17] W. Wu, H. Li, H. Wang, and K. Q. Zhu, “Probase: a probabilistic taxonomy for text understanding,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’12. New York, NY, USA: ACM, 2012, pp. 481–492.

[18] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, “A distributed graph engine for web scale rdf data,” in *Proceedings of the 39th international conference on Very Large Data Bases*, ser. PVLDB’13. VLDB Endowment, 2013, pp. 265–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=2488329.2488333

[19] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, “Online search of overlapping communities,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13. New York, NY, USA: ACM, 2013, pp. 277–288.

[20] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, “Efficient subgraph matching on billion node graphs,” *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, May 2012. [Online]. Available: http://dx.doi.org/10.14778/2311906.2311907

[21] Z. Qi, Y. Xiao, B. Shao, and H. Wang, “Toward a distance oracle for billion-node graphs,” *Proc. VLDB Endow.*, vol. 7, no. 1, pp. 61–72, Sep. 2013.

[22] L. Wang, Y. Xiao, B. Shao, and H. Wang, “How to partition a billion-node graph,” in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, Eds. IEEE, 2014, pp. 568–579.

[23] B. Shao, H. Wang, and Y. Li, “Trinity: A distributed graph engine on a memory cloud,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13. New York, NY, USA: ACM, 2013, pp. 505–516.

[24] L. He, B. Shao, Y. Li, and E. Chen, “Distributed real-time knowledge graph serving,” in *2015 International Conference on Big Data and Smart Computing, BIGCOMP 2015, Jeju, South Korea, February 9-11, 2015*. IEEE, 2015, pp. 262–265.

[25] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang, “G-sql: Fast query processing via graph exploration,” *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 900–911, Aug. 2016.

GRAPE: Conducting Parallel Graph Computations without Developing Parallel Algorithms

Wenfei Fan^{1,2}, Jingbo Xu^{1,2}, Xiaojian Luo^{2}, Yinghui Wu^{3}, Wenyuan Yu^{2}, Ruiqi Xu^{1}
^{1}*University of Edinburgh* ^{2}*Beihang University* ^{3}*Washington State University*
{wenfei@inf, jingbo.xu@}ed.ac.uk, luoxiaojian@buaa.edu.cn yinghui@eecs.wsu.edu
yuwenyuan@act.buaa.edu.cn, Ruiqi.Xu@ed.ac.uk

Abstract

Developing parallel graph algorithms with correctness guarantees is nontrivial even for experienced programmers. Is it possible to parallelize existing sequential graph algorithms, without recasting the algorithms into a parallel model? Better yet, can the parallelization guarantee to converge at correct answers as long as the sequential algorithms provided are correct? GRAPE *tackles these questions, to make parallel graph computations accessible to a large group of users. This paper presents (a) the parallel model of* GRAPE*, based on partial evaluation and incremental computation, and (b) a performance study, showing that* GRAPE *achieves performance comparable to the state-of-the-art systems.*

1 Introduction

The need for graph computations is evident in transportation network analysis, knowledge extraction, Web mining, social network analysis and social media marketing, among other things. Graph computations are, however, costly in real-life graphs. For instance, the social graph of Facebook has billions of nodes and trillions of edges [16]. In such a graph, it is already expensive to compute shortest distances from a single source, not to mention graph pattern matching by subgraph isomorphism, which is intractable in nature.

To support graph computations in large-scale graphs, several parallel systems have been developed, *e.g.,* Pregel [23], GraphLab [22], Giraph++ [29], GraphX [15], GRACE [33], GPS [27] and Blogel [34], based on MapReduce [8] and (variants of) BSP (Bulk Synchronous Parallel) models [30]. These systems, however, do not allow us to reuse existing sequential graph algorithms, which have been studied for decades and are well optimized. To use Pregel [23], for instance, one has to "think like a vertex" and recast existing algorithms into a vertex-centric model; similarly when programming with other systems, *e.g.,* [34], which adopts vertex-centric programming by treating blocks as vertices. The recasting is nontrivial for people who are not very familiar with the parallel models. This makes parallel graph computations a privilege of experienced users only.

Is it possible to make parallel graph computations accessible to users who only know conventional graph algorithms covered in undergraduate textbooks? Can we have a system such that given a graph computation problem, we can "plug in" its existing sequential algorithms for the problem as a whole, without recasting or "thinking in parallel", and the system automatically parallelizes the computation across multiple processors? Moreover, can the system guarantee that the parallelization terminates and converges at correct answers as long as the sequential algorithms plugged in are correct? Furthermore, can the system inherit optimization techniques

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

well developed for sequential graph algorithms, such as indexing and pruning? Better yet, despite the ease of programming, can the system achieve performance comparable to the state-of-the-art parallel graph systems?

These questions motivate us to develop GRAPE, a parallel GRAPh Engine [13]. GRAPE advocates a parallel model based on simultaneous fixpoint computation that starts with partial evaluation and takes incremental computation as the intermediate consequence operator. It answers all the questions above in the affirmative. As a proof of concept, we have developed a preliminary implementation of GRAPE [12].

This paper presents (a) a brief introduction to the parallel model of GRAPE, and (b) a performance study of the system, using real-life larger than those that [13] experimented with.

2 Parallelizing Sequential Graph Algorithms

We present the programming model and parallel computation model of GRAPE, illustrating how GRAPE parallelizes sequential graph algorithms. We encourage the interested reader to consult [13] for more details.

2.1 Programming Model

Consider a graph computation problem $\mathcal{Q}$. Using our familiar terms, we refer to an instance Q of $\mathcal{Q}$ as a *query* of $\mathcal{Q}$. To answer queries $Q \in \mathcal{Q}$ with GRAPE, a user only needs to specify three functions as follows.

PEval: an algorithm for $\mathcal{Q}$ that given a query $Q \in \mathcal{Q}$ and a graph G, computes the answer $Q(G)$ to Q in G.

IncEval: an incremental algorithm for $\mathcal{Q}$ that given Q, G, $Q(G)$ and updates ΔG to G, computes changes ΔO to the old output $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$, where $G \oplus \Delta G$ denotes graph G updated by ΔG.

Assemble: a function that collects partial answers computed locally at each worker by PEval and IncEval (see Section 2.2), and assembles them into complete answer $Q(G)$. It is typically straightforward.

The three functions PEval, IncEval and Assemble are referred to as *a* PIE *program for* $\mathcal{Q}$. Here PEval and IncEval are *existing sequential* (incremental) algorithms for $\mathcal{Q}$, with the following additions to PEval.

- *Update parameters.* PEval declares *status variables* $\vec{x}$ for a set of nodes and edges. As will be seen shortly, these variables are the candidates to be updated by incremental steps IncEval.
- *Aggregate function.* PEval specifies a function f_{aggr}, *e.g.,* min, max, to resolve conflicts when multiple workers attempt to assign different values to the same update parameter.

The update parameters and aggregate function are specified in PEval and are shared by IncEval.

From these we can see that programming with GRAPE is as simple as MapReduce, if not simpler. Indeed, GRAPE asks the users to provide three functions, where PEval and IncEval are existing sequential algorithms without recasting them into a new model, unlike MapReduce, and Assemble is typically a simple function.

Graph partition. GRAPE supports data-partitioned parallelism. It allows users to pick an (existing) graph partition strategy $\mathcal{P}$ registered in GRAPE, and partitions a (possibly big) graph G into smaller fragments.

More specifically, we consider graphs $G = (V, E, L)$, directed or undirected, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; (3) each node v in V (resp. edge $e \in E$) carries label $L(v)$ (resp. $L(e)$), indicating its content, as found in social networks and property graphs.

Given a number m, strategy $\mathcal{P}$ partitions G into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $E = \bigcup_{i\in[1,m]} E_i$, and $V = \bigcup_{i\in[1,m]} V_i$. Here F_i is a *subgraph of* G if $V_i \subseteq V$, $E_i \subseteq E$, and for each node $v \in V_i$ (resp. edge $e \in E_i$), $L_i(v) = L(v)$ (resp. $L_i(e) = L(e)$).

Here $\mathcal{P}$ can be any partition strategy, *e.g.,* vertex-cut [20] or edge cut [6]. When $\mathcal{P}$ is edge-cut, denote by

- $F_i.I$ the set of nodes $v \in V_i$ such that there is an edge (v', v) *incoming* from a node v' in F_j $(i \neq j)$;
- $F_i.O$ the set of nodes v' such that there exists an edge (v, v') in E, $v \in V_i$ and v' is in F_j $(i \neq j)$; and
- $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$, $\mathcal{F}.I = \bigcup_{i\in[1,m]} F_i.I$; $\mathcal{F}.O = \mathcal{F}.I$.

For vertex-cut, $\mathcal{F}.O$ and $\mathcal{F}.I$ correspond to exit vertices and entry vertices, respectively.

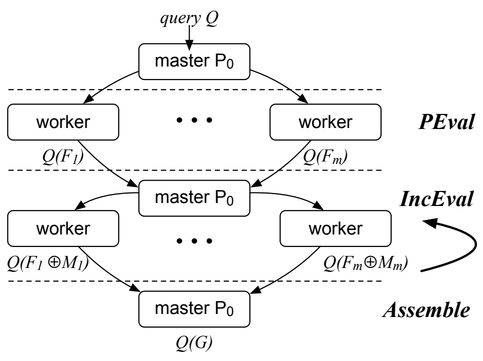


Figure 1: Workflow of GRAPE

2.2 Parallel Model

Given a partition strategy $\mathcal{P}$ and a PIE program ρ (PEval, IncEval, Assemble) for $\mathcal{Q}$, GRAPE parallelizes ρ as follows. It first partitions G into $(F_1, \ldots, F_m)$ with $\mathcal{P}$, and distributes fragments F_i's across m shared-nothing *virtual workers* $(P_1, \ldots, P_m)$. It maps m virtual workers to n physical workers. When $n < m$, multiple virtual workers mapped to the same worker share memory. Graph G is partitioned *once* for *all queries* $Q \in \mathcal{Q}$ on G.

Partial evaluation and incremental computation. We start with basic ideas behind GRAPE parallelization.

Given a function $f(s, d)$ and the s part of its input, *partial evaluation* is to specialize $f(s, d)$ *w.r.t.* the known input s [19]. That is, it performs the part of f's computation that depends only on s, and generates a partial answer, *i.e.,* a residual function f' that depends on the as yet unavailable input d. For each worker P_i in GRAPE, its local fragment F_i is its known input s, while the data residing at other workers accounts for the yet unavailable input d. As will be seen shortly, given a query $Q \in \mathcal{Q}$, GRAPE computes $Q(F_i)$ in parallel as partial evaluation.

Workers exchange *changed values* of their local update parameters with each other. Upon receiving message M_i that consists of changes to the update parameters at fragment F_i, worker P_i treats M_i as *updates* to F_i, and *incrementally* computes changes ΔO_i to $Q(F_i)$ such that $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$. This is often more efficient than recomputing $Q(F_i \oplus M_i)$ starting from scratch, since in practice M_i is often small, and so is O_i. Better still, the incremental computation may be *bounded*: its cost can be expressed as a function in $|M_i|+|\Delta O_i|$, *i.e.,* the size of changes in the input and output, instead of $|F_i|$, no matter how big F_i is [11,26].

Model. Following BSP [30], given a query $Q \in \mathcal{Q}$ at master P_0, GRAPE answers Q in the partitioned graph G. It posts the same query Q to all the workers, and computes $Q(G)$ in three phases as follows, as shown in Fig. 1.

(1) Partial evaluation (**PEval**). In the first superstep, upon receiving Q, each worker P_i applies function PEval to its local fragment F_i, to compute partial results $Q(F_i)$, in parallel ($i \in [1, m]$). After $Q(F_i)$ is computed, PEval generates a message at each worker P_i and sends it to master P_0. The message is simply *the set of update parameters at fragment* F_i, denoted by $C_i.\bar{x}$. More specifically, $C_i.\bar{x}$ consists of status variables associated with a set C_i of nodes and edges within d-hops of nodes in $F_i.O$. Here d is an integer determined by query Q only, specified in function PEval. In particular, when $d = 0$, C_i is $F_i.O$.

For each $i \in [1, m]$, master P_0 maintains update parameters $C_i.\bar{x}$. It deduces a message M_i to worker P_i based on the following *message grouping policy*. (a) For each status variable $x \in C_i.\bar{x}$, it collects the set S_x of values for x from all messages, and computes $x_{\mathsf{aggr}} = f_{\mathsf{aggr}}(S_x)$ by applying the aggregate function f_{aggr}. (b) Message M_i includes only those x_{aggr}'s such that $x_{\mathsf{aggr}} \neq x$, *i.e.,* only *changed* parameter values.

(2) Incremental computation (**IncEval**). GRAPE iterates the following supersteps until it terminates. Following BSP, each superstep starts after the master P_0 receives messages (possibly empty) from *all workers* P_i for $i \in [1, m]$. A superstep has two steps itself, one at P_0 and the other at the workers.

(a) Master P_0 routes (nonempty) messages from the last superstep to workers, if there exists any.
(b) Upon receiving message M_i, worker P_i *incrementally* computes $Q(F_i \oplus M_i)$ by applying IncEval, and by *treating M_i as updates*, in parallel for $i \in [1, m]$.

At the end of the process of IncEval, P_i sends a message to P_0 that encodes *updated values* of $C_i.\bar{x}$, if any. Upon receiving messages from all workers, master P_0 deduces message M_i to each worker P_i following the message grouping policy given above; it sends message M_i to worker P_i in the next superstep.

(3) Termination (**Assemble**). At each superstep, master P_0 checks whether for all $i \in [1, m]$, P_i is inactive, *i.e.,* P_i is done with its local computation, and there exists no more change to any update parameter of F_i. If so, GRAPE pulls partial results from all workers, and applies Assemble to group together the partial results and get the final result at P_0, denoted by $\rho(Q, G)$. It returns $\rho(Q, G)$ and terminates.

Fixpoint. The GRAPE parallelization of the PIE program can be modeled as a simultaneous fixpoint operator $\phi(R_1, \ldots, R_m)$ defined on m fragments. It starts with PEval for partial evaluation, and conducts incremental computation by taking IncEval as the intermediate consequence operator, as follows:

$$\begin{aligned} R_i^0 &= \mathsf{PEval}(Q, F_i^0[\bar{x}_i]), \\ R_i^{r+1} &= \mathsf{IncEval}(Q, R_i^r, F_i^r[\bar{x}_i], M_i), \end{aligned}$$

where $i \in [1, m]$, r indicates a superstep, R_i^r denotes partial results in step r at worker P_i, fragment $F_i^0 = F_i$, $F_i^r[\bar{x}_i]$ is fragment F_i at the end of superstep r carrying update parameters $\bar{x}_i$, and M_i is a message indicating changes to $\bar{x}_i$. More specifically, (1) in the first superstep, PEval computes partial answers R_i^0 ($i \in [1, m]$). (2) At step $r + 1$, the partial answers R_i^{r+1} are incrementally updated by IncEval, taking Q, R_i^r and message M_i as input. (3) The computation proceeds until $R_i^{r_0+1} = R_i^{r_0}$ at a fixpoint r_0 for all $i \in [1, m]$. Function Assemble is then invoked to combine all partial answers $R_i^{r_0}$ and get the final answer $\rho(Q, G)$.

Convergence. To characterize the correctness of the fixpoint computation, we use the following notations. (a) A sequential algorithm PEval for $\mathcal{Q}$ is *correct* if given all queries $Q \in \mathcal{Q}$ and graphs G, it terminates and returns $Q(G)$. (b) A sequential incremental algorithm IncEval for $\mathcal{Q}$ is *correct* if given all $Q \in \mathcal{Q}$, graphs G, old output $Q(G)$ and updates ΔG to G, it computes changes ΔO to $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$. (c) Assemble is *correct for $\mathcal{Q}$ w.r.t. $\mathcal{P}$* if when GRAPE with PEval, IncEval and $\mathcal{P}$ terminates at superstep r_0, $\mathsf{Assemble}(Q(F_1[\bar{x}_1^{r_0}]), \ldots, Q(F_m[\bar{x}_m^{r_0}])) = Q(G)$, where $\bar{x}_i^{r_0}$ denotes the values of parameters $C_i.\bar{x}_i$ at round r_0. (d) We say that GRAPE *correctly parallelizes* a PIE program ρ with a partition strategy $\mathcal{P}$ if for all queries $Q \in \mathcal{Q}$ and graphs G, GRAPE guarantees to reach a fixpoint such that $\rho(Q, G) = Q(G)$.

It is shown [13] that under BSP, GRAPE correctly parallelizes a PIE program ρ for a graph computation problem $\mathcal{Q}$ if (a) its PEval and IncEval are correct sequential algorithms for $\mathcal{Q}$, and (b) Assemble correctly combines partial results, and (c) PEval and IncEval satisfy a monotonic condition. The condition is as follows: for all variables $x \in C_i.\bar{x}$, $i \in [1, m]$, (a) the values of x are computed from the active domain of G and (b) there exists a partial order p_x on the values of x such that IncEval updates x in the order of p_x. That is, x draws values from a finite domain (condition (a)), and x is updated "monotonically" following p_x (condition (b)).

Example 1: We show how GRAPE parallelizes the computation of Single Source Shortest Path (SSSP), a common graph computation problem. Consider a directed graph $G = (V, E, L)$ in which for each edge e, $L(e)$ is a positive number. The length of a path $(v_0, \ldots, v_k)$ in G is the sum of $L(v_{i-1}, v_i)$ for $i \in [1, k]$. For a pair (s, v) of nodes, denote by $\mathsf{dist}(s, v)$ the *distance* from s to v, *i.e.,* the length of a shortest path from s to v. Given graph G and a node s in V, SSSP computes $\mathsf{dist}(s, v)$ for all $v \in V$.

Adopting edge-cut partition [6], GRAPE takes the set $F_i.O$ of "border nodes" as C_i at each worker P_i, *i.e.,* nodes with edges across different fragments. The PIE program of SSSP in GRAPE specializes three functions: (1) a standalone sequential algorithm for SSSP as PEval, *e.g.,* our familiar Dijkstra's algorithm [14], to compute $Q(F_i)$ as partial evaluation; (2) a bounded sequential incremental algorithm of [25] as IncEval that computes $Q(F_i \oplus M_i)$, where messages M_i include updated (smaller) $\mathsf{dist}(s, u)$ (due to new "shortcut" from s) for border nodes u; and (3) Assemble that takes a union of the partial answers $Q(F_i)$ as $Q(G)$.

(1) *PEval*. As shown in Fig. 2a, PEval (lines 1-14) is verbally identical to Dijsktra's algorithm [14]. One only

```
Input: $F_i(V_i, E_i, L_i)$, source vertex $s$
Output: $Q(F_i)$ consisting of current $\mathsf{dist}(s, v)$ for all $v \in V_i$

Declare: (designated) /*candidate set $C_i$ is $F_i.O$*/
for each node $v \in V_i$, an integer variable $\mathsf{dist}(s, v)$;
message $M_i := \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$;
function aggregateMsg = $\mathsf{min}(\mathsf{dist}(s, v))$;

/*sequential algorithm for SSSP (pseudo-code)*/
1.  initialize priority queue Que;
2.  $\mathsf{dist}(s, s)$ := 0;
3.  for each $v$ in $V_i$ do
4.     if $v! = s$ then
5.     $\mathsf{dist}(s, v) := \infty$;
6.  Que.addOrAdjust($s$, $\mathsf{dist}(s, s)$);
7.  while Que is not empty do
8.     $u$ := Que.pop() // pop vertex with minimal distance
9.     for each child $v$ of $u$ do // only v that is still in Q
10.       $alt$ := $\mathsf{dist}(s, u)$ + $L_i(u, v)$;
11.       if $alt < \mathsf{dist}(s, v)$ then
12.          $\mathsf{dist}(s, v) := alt$;
13.          Que.addOrAdjust($v$, $\mathsf{dist}(s, v)$);
14. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$
```

(a) PEval for SSSP

```
Input: $F_i(V_i, E_i, L_i)$, partial result $Q(F_i)$, message $M_i$
Output: $Q(F_i \oplus M_i)$

Declare: message $M_i = \{\mathsf{dist}(s, v) \mid v \in F_i.O,\ \mathsf{dist}(s, v)$ decreased$\}$;
1.  initialize priority queue Que;
2.  for each $\mathsf{dist}(s, v)$ in $M$ do
3.     Que.addOrAdjust($v$, $\mathsf{dist}(s, v)$);
4.  while Que is not empty do
5.     $u$ := Que.pop() /* pop vertex with minimum distance*/
6.     for each children $v$ of $u$ do
7.        $alt$ := $\mathsf{dist}(s, u)$ + $L_i(u, v)$;
8.        if $alt < \mathsf{dist}(s, v)$ then
9.           $\mathsf{dist}(s, v) := alt$;
10.          Que.addOrAdjust($v$, $\mathsf{dist}(s, v)$);
11. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$
```

(b) IncEval for SSSP

Figure 2: GRAPE for SSSP

needs to declare status variable as an integer variable $\mathsf{dist}(s, v)$ for each node v, initially ∞ (except $\mathsf{dist}(s, s)$ = 0) and (a) update parameters (in message M_i) as $C_i.\bar{x} = \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$, *i.e.,* the status variables associated with nodes in $F_i.O$ at fragment F_i; and (b) min as an aggregate function (**aggregateMsg**). If there are multiple values for the same $\mathsf{dist}(s, v)$, the smallest value is taken by the linear order on integers.

At the end of its process, PEval sends $C_i.\bar{x}$ to master P_0. At P_0, GRAPE maintains $\mathsf{dist}(s, v)$ for all $v \in \mathcal{F}.O = \mathcal{F}.I$. Upon receiving messages from all workers, it takes the smallest value for each $\mathsf{dist}(s, v)$. It finds those variables with smaller $\mathsf{dist}(s, v)$ for $v \in F_j.O$, groups them into message M_j, and sends M_j to P_j.

(2) *IncEval*. We give IncEval in Fig. 2b. It is the sequential incremental algorithm for SSSP in [26], in response to changed $\mathsf{dist}(s, v)$ for v in $F_i.I$ (deduced by leveraging $\mathcal{F}.I = \mathcal{F}.O$). Using a queue Que, it starts with changes in M_i, propagates the changes to affected area, and updates the distances (see [26]). The partial result now consists of the revised distances (line 11). At the end of the process, it sends to master P_0 the updated values of those status variables in $C_i.\bar{x}$, as in PEval. It applies the aggregate function min to resolve conflicts.

The only changes to the algorithm of [26] are underlined in Fig. 2b. Following [26], one can show that IncEval is *bounded*: its cost is determined by the sizes of "updates" $|M_i|$ and the changes to the output. This reduces the cost of iterative computation of SSSP (the While and For loops).

(3) *Assemble* simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of the shortest distance for each node in each F_i.

The GRAPE process converges at correct $Q(G)$. Updates to $C_i.\bar{x}$ are "monotonic": the value of $\mathsf{dist}(s, v)$ for each node v decreases or remains unchanged. There are finitely many such variables. Moreover, $\mathsf{dist}(s, v)$ is the shortest distance from s to v as warranted by the sequential algorithms [14, 26] (PEval and IncEval). □

Expressive power. The simple parallel model of GRAPE does not come with a price of degradation in the functionality. More specifically, following [31], we say that a parallel model $\mathcal{M}_1$ can *optimally simulate* model $\mathcal{M}_2$ if there exists a compilation algorithm that transforms any program with cost C on $\mathcal{M}_2$ to a program with cost $O(C)$ on $\mathcal{M}_1$. The cost includes computational cost and communication cost.

As shown in [13], GRAPE optimally simulates parallel models MapReduce [8], BSP [30] and PRAM (Parallel Random Access Machine) [31]. That is, all algorithms in MapRedue, BSP or PRAM with n workers can be simulated by GRAPE using n processors with the same number of supersteps and memory cost. Hence algorithms developed for graph systems based on MapReduce or BSP, *e.g.,* Pregel, GraphLab and Blogel, can be readily migrated to GRAPE without extra cost. We have established a stronger result, *i.e.,* the simulation result above holds in the message-passing model described above, referred to as the designated message model in [13].

As promised, GRAPE has the following unique features. (1) For a graph computation problem $\mathcal{Q}$, GRAPE allows one to plug in existing sequential algorithms PEval and IncEval, subject to minor changes, and it automatically parallelizes the computation. (2) Under a monotone condition, the parallelization guarantees to converge at the correct answer in any graph, as long as the sequential algorithms PEval and IncEval are correct. (3) MapReduce, BSP and PRAM [31] can be optimally simulated by GRAPE. (4) GRAPE easily capitalizes on existing optimization techniques developed for sequential graph algorithms, since it plugs in sequential algorithms as a whole, and executes these algorithms on entire graph fragments. (5) GRAPE reduces the costs of iterative graph computations by using IncEval, to minimize unnecessary recomputations. The speedup is particularly evident when IncEval is bounded [26], localizable or relatively bounded (see [9] for the latter two criteria).

2.3 Programming with GRAPE

As examples, we outline PIE programs for graph pattern matching (Sim and SubIso), connectivity (CC) and collaborative filtering (CF). We will conduct experimental study with this variety of computations. We adopt edge-cut [6] for the PIE programs below. PIE programs under vertex-cut [20] can be developed similarly.

Graph simulation (Sim). A *graph pattern* is a graph $Q = (V_Q, E_Q, L_Q)$, in which (a) V_Q is a set of *query nodes*, (b) E_Q is a set of *query edges*, and (c) each node u in V_Q carries a label $L_Q(u)$.

A graph G *matches* a pattern Q via *simulation* if there is a binary relation $R \subseteq V_Q \times V$ such that (a) for each query node $u \in V_Q$, there exists a node $v \in V$ such that $(u, v) \in R$, and (b) for each pair $(u, v) \in R$, $L_Q(u) = L(v)$, and for each query edge (u, u') in E_q, there exists an edge (v, v') in graph G such that $(u', v') \in R$.

It is known that if G matches Q, then there exists a *unique maximum* relation [18], referred to as $Q(G)$. If G does not match Q, $Q(G)$ is the empty set. Graph simulation is in $O((|V_Q| + |E_Q|)(|V| + |E|))$ time [10, 18].

Given a directed graph G and a pattern Q, graph simulation is to compute the maximum relation $Q(G)$.

We show how GRAPE parallelizes graph simulation. GRAPE takes the sequential simulation algorithm of [18] as PEval to compute $Q(F_i)$ in parallel. PEval declares a Boolean variable $x_{(u,v)}$ for each query node u in V_Q and each node v in F_i, indicating whether v matches u, initialized true. It takes the set $F_i.I$ of "border nodes" as candidate set C_i, and $C_i.\bar{x}_{(u,v)}$ as the set of update parameters at F_i. At master P_0, GRAPE maintains $x_{(u,v)}$ for all $v \in \mathcal{F}.I$. Upon receiving messages from all workers, it changes $x_{(u,v)}$ to false if it is false in *one of* the messages. That is, it uses min as the aggregate function, taking the order false $\prec$ true. GRAPE identifies those variables that become false, groups them into messages M_j, and sends M_j to P_j.

IncEval is the *semi-bounded* sequential incremental algorithm of [11]: its cost is decided by the sizes of "updates" $|M_i|$ and changes necessarily checked by all incremental algorithms for Sim, not by $|F_i|$.

The process guarantees to terminate since the update parameters $C_i.\bar{x}$'s are monotonically updated. At this point, Assemble simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of the simulation relation at each F_i.

Subgraph isomorphism. Here a *match* of pattern Q in graph G is a subgraph of G that is isomorphic to Q, and the problem is to compute the set $Q(G)$ of all matches of Q in G. The problem is intractable.

GRAPE parallelizes subgraph isomorphism with *two* supersteps, one for PEval and the other for IncEval. (a) PEval is a sequential Breadth First Search (BFS) algorithm that "fetches" a set $\bigcup_{v \in F_i.I} N_{d_Q}(v)$ of nodes and edges at F_i in parallel. Here $N_{d_Q}(v)$ is the set of nodes and edges that are within d_Q hop of v, and d_Q is the diameter of pattern Q, *i.e.,* the length of the *undirected* shortest path between any two nodes in Q. PEval declares such d_Q-neighbors as $C_i.\bar{x}$ at each F_i, and identifies the update parameters via BFS. (b) IncEval is simply VF2, the sequential algorithm of [7] for subgraph isomorphism. It computes $Q(F_i \oplus M_i)$ at each worker P_i in parallel, in fragment F_i extended with d_Q-neighbor of each node in $F_i.I$. (c) Assemble takes the union of all partial matches computed by IncEval from all workers. The correctness is assured by VF2 and the locality of subgraph isomorphism: a pair (v, v') of nodes in G is in a match of Q only if v is in the d_Q-neighbor of v'.

When message $\bigcup_{v \in F_i.I} N_{d_Q}(v)$ is large, GRAPE further paralellizes PEval by expanding $N_{d_Q}(v)$ step by step, one hop at each step, to reduce communication cost and stragglers. More specifically, IncEval is revised such that it identifies new "border nodes", and finds the 1-hop neighbor of each border node via incremental BFS to propagate the changes. This proceeds until $\bigcup_{v \in F_i.I} N_{d_Q}(v)$ is in place, and at this point VF2 is triggered.

Graph connectivity (CC). Given an undirected graph G, CC computes all connected components of G. It picks a sequential CC algorithm as PEval, *e.g.,* DFS. It declares an integer variable v.cid for each node v, recording the component in which v belongs. The update parameters at fragment F_i include v.cid for all $v \in F_i.I$. At each fragment F_i, PEval computes its local connected components and creates their ids. It exchanges v.cid for all border nodes $v \in F_i.I$ with neighboring fragments. Given message M_i consisting of the changed v.cid's for border nodes, IncEval incrementally updates local components in fragment F_i. It merges two components whenever possible, by using min as the aggregate function that takes the smallest component id. The process proceeds until no more changes can be made. At this point, Assemble merges all the nodes having the same cid into a single connected component, and returns all the connected components.

The process terminates since the cids of the nodes are monotonically decreasing. Moreover, IncEval is *bounded*: its cost is a function of the number of v.cid's whose values are changed in response to M_i.

Collaborative filtering (CF). CF takes as input a bipartite graph G that includes users U and products P, and a set of weighted edges $E \subseteq U \times P$ [21]. (1) Each user $u \in U$ (resp. product $p \in P$) carries latent factor vector $u.f$ (resp. $p.f$). (2) Each edge $e = (u, p)$ in E carries a weight $r(e)$, estimated as $u.f^T * p.f$ ($\emptyset$ for "unknown") that encodes a rating from user u to product p. The *training set* E_T refers to edge set $\{e \mid r(e) \neq \emptyset, e \in E\}$, *i.e.,* all the known ratings. Given these, CF computes the missing factor vectors $u.f$ and $p.f$ to minimize an error function $\epsilon(f, E_T) = \min \sum_{((u,p) \in E_T)} (r(u,p) - u.f^T p.f)^2 + \lambda(\|u.f\|^2 + \|p.f\|^2)$. This is typically carried out by the stochastic gradient descent (SGD) algorithm [21], which iteratively (1) predicts error $\epsilon(u, p) = r(u, p) - u.f^T * p.f$, for each $e = (u, p) \in E_T$, and (2) updates $u.f$ and $p.f$ accordingly to minimize $\epsilon(f, E_T)$.

GRAPE parallelizes CF by adopting SGD [21] as PEval, and the incremental algorithm ISGD of [32] as IncEval, using master P_0 to synchronize the shared factor vectors $u.f$ and $p.f$. More specifically, PEval declares $v.x = (v.f, t)$ for each node v (initially $\emptyset$), and t bookkeeps a timestamp at which $v.f$ is lastly updated. At fragment F_i, the update parameters include $v.x$ for all $v \in F_i.O$. At P_0, GRAPE maintains $v.x = (v.f, t)$ for all $v \in \mathcal{F}.I = \mathcal{F}.O$. Upon receiving updated $(v.f', t')$ with $t' > t$, it changes $v.f$ to $v.f'$, *i.e.,* takes max as the aggregate function on timestamps. Given M_i at worker P_i, IncEval operates on $F_i \oplus M_i$ by treating M_i as updates to factor vectors of nodes in $F_i.I$, and only modifies affected vectors. The process proceeds until $\epsilon(f, E_T)$ becomes smaller than a threshold, or after a predefined number of step. Assemble simply takes the union of all the vectors from the workers. As long as the sequential SGD algorithm terminates, the PIE program converges at the correct answer since the updates are monotonic with the latest changes as in the sequential SGD.

3 Performance Study

We next empirically evaluate GRAPE, for its (1) efficiency, and (2) communication cost, using real-life graphs larger than those that [13] experimented with. We evaluated the performance of GRAPE compared with Giraph (an open-source version of Pregel), GraphLab, and Blogel (the fastest block-centric system we are aware of).

Experimental setting. We used five real-life graphs of different types, including (1) movieLens [3], a dense recommendation network (bipartite graph) that has 20 million movie ratings (as weighted edges) between a set of 138000 users and 27000 movies; (2) UKWeb [5], a large Web graph with 133 million nodes and 5 billion edges, (3) DBpedia [1], a knowledge base with 5 million entities and 54 million edges, and in total 411 distinct labels, (4) Friendster [2], a social network with 65 million users and 1.8 billion relations; and (5) traffic [4], an (undirected) US road network with 23 million nodes (locations) and 58 million edges. To make use of unlabeled Friendster for Sim and SubIso, we assigned up to 100 random labels to nodes. We also randomly assigned weights to all the graphs for testing SSSP.

Queries. We randomly generated the following queries. (a) We sampled 10 source nodes in each graph, and constructed an SSSP query for each node. (b) We generated 20 pattern queries for Sim and SubIso, controlled by $|Q| = (|V_Q|, |E_Q|)$, the number of nodes and edges, respectively, using labels drawn from the graphs.

We remark that GRAPE can process query workload without reloading the graph, but GraphLab, Giraph and Blogel require the graph to be reloaded each time a single query is issued, which is costly over large graphs.

Algorithms. We implemented the core functions PEval, IncEval and Assemble for those query classes given in Sections 2.3, registered in the API library of GRAPE. We used XtraPuLP [28] as the default graph partition strategy. We adopted basic sequential algorithms for all the systems without optimization.

We also implemented algorithms for the queries for Giraph, GraphLab and Blogel. We used "default" code provided by the systems when available, and made our best efforts to develop "optimal" algorithms otherwise (see [13] for more details). We implemented synchronized algorithms for both GraphLab and Giraph for the ease of comparison. We expect the observed relative performance trends to hold on other similar graph systems.

We deployed the systems on a cluster of up to 12 machines, each with 16 processors (Intel Xeon 2.2GHz) and 128G memory (thus in total 192 workers). Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the efficiency of GRAPE by varying the number n of worker used, from 64 to 192. For SSSP and CC, we experimented with UKWeb, traffic and Friendster. For Sim and SubIso, we used over Friendster and DBpedia. We used movieLens for CF as its application in movie recommendation.

(1) SSSP. Figures 3a-3c report the performance of the systems for SSSP over Friendster, UKWeb and traffic, respectively. The results on other graphs are consistent (not shown). From the results we can see the following.

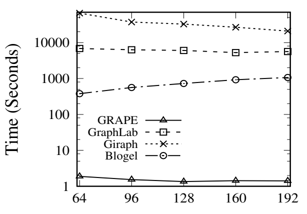
(a) GRAPE outperforms Giraph, GraphLab and Blogel by 14842, 3992 and 756 times, respectively, over traffic with 192 workers (Fig 3a). In the same setting, it is 556, 102 and 36 times faster over UKWeb (Fig. 3b), and 18, 1.7 and 4.6 times faster over Friendster (Fig. 3c). By simply parallelizing sequential algorithms without further optimization, GRAPE already outperforms the state-of-the-art systems in response time.

The improvement of GRAPE over all the systems on traffic is much larger than on Friendster and UKWeb. (i) For Giraph and GraphLab, this is because synchronous vertex-centric algorithms take more supersteps to converge on graphs with larger diameters, *e.g.,* traffic. With 192 workers, Giraph take 10749 supersteps over traffic and 161 over UKWeb; similarly for GraphLab. In contrast, GRAPE is not vertex-centric and it takes 31 supersteps on traffic and 24 on UKWeb. (ii) Blogel also takes more (1690) supersteps over traffic than over UKWeb (42) and Friendster (23). It generates more blocks over traffic (with larger diameter) than UKWeb and Friendster. As Blogel treats blocks as vertex, the benefit of parallelism is degraded with more blocks.

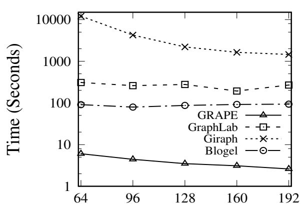
(b) In all cases, GRAPE take less time when n increases. On average, it is 1.4, 2.3 and 1.5 times faster for n from 64 to 192 over traffic, UKWeb and Friendster, respectively. (i) Compared with the results in [13] using less workers, this improvement degrades a bit. This is mainly because the larger number of fragments leads to more communication overhead. On the other hand, such impact is significantly mitigated by IncEval that only ships changed update parameters. (ii) In contrast, Blogel does not demonstrate such consistency in scalability. It takes more time on traffic when n is larger. When n varies from 160 to 192, it takes longer over Friendster. Its communication cost dominates the parallel cost as n grows, "canceling out" the benefit of parallelism. (iii) GRAPE has scalability comparable to GraphLab over Friendster and scales better over UKWeb and traffic. Giraph has better improvement with larger n, but with constantly higher cost (see (a)) than GRAPE.

(c) GRAPE significantly reduces supersteps. It takes on average 22 supersteps, while Giraph, GraphLab and Blogel take 3647, 3647 and 585 supersteps, respectively. This is because GRAPE runs sequential algorithms over fragmented graphs with cross-fragment communication only when necessary, and IncEval ships only *changes* to status variables. In contrast, Giraph, GraphLab and Blogel pass vertex-vertex (vertex-block) messages.

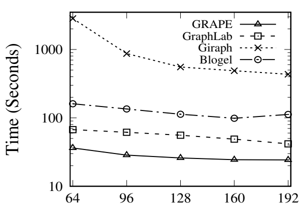
(2) CC. Figures 3d-3f report the performance for CC detection, and tell us the following. (a) Both GRAPE and Blogel substantially outperform Giraph and GraphLab. For instance, when $n = 192$, GRAPE is on average



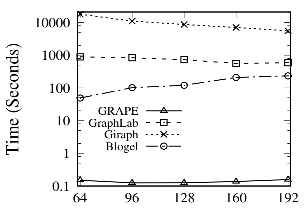
(a) Varying n: SSSP (traffic)



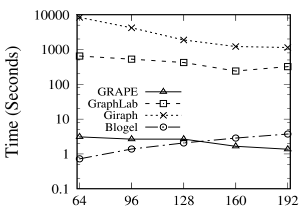
(b) Varying n: SSSP (UKWeb)



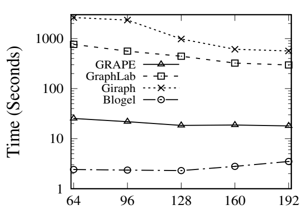
(c) Varying n: SSSP (Friendster)



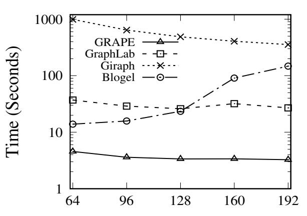
(d) Varying n: CC (traffic)



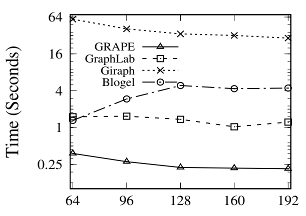
(e) Varying n: CC (UKWeb)



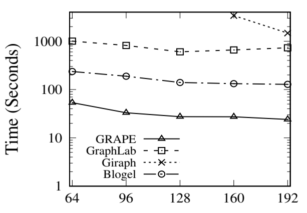
(f) Varying n: CC (Friendster)



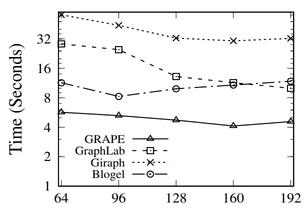
(g) Varying n: Sim (Friendster)



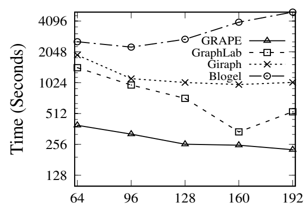
(h) Varying n: Sim (DBpedia)



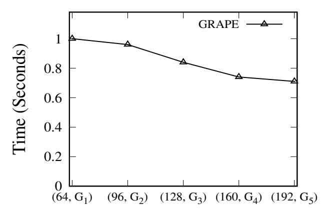
(i) Varying n: SubIso(Friendster)



(j) Varying n: SubIso (DBpedia)



(k) Varying n: CF(movieLens)



(l) Scale-up of GRAPE (Synthetic)

Figure 3: Efficiency of GRAPE

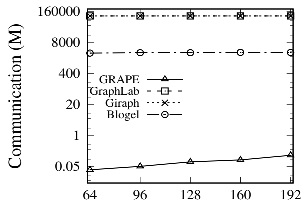
12094 and 1329 times faster than Giraph and GraphLab, respectively. (b) Blogel is faster than GRAPE in some cases, *e.g.,* 3.5s vs. 17.9s over Friendster when $n = 192$. This is because Blogel embeds the computation of CC in its graph partition phase as precomputation, while this graph partition cost (on average 357 seconds using its built-in Voronoi partition) is *not* included in its response time. In other words, without precomputation, the performance of GRAPE is already comparable to the near "optimal" case reported by Blogel.

(3) Sim. Fixing $|Q| = (6, 10)$, *i.e.,* patterns Q with 6 nodes and 10 edges, we evaluated graph simulation over DBpedia and Friendster. As shown in Figures 3g-3h, (a) GRAPE consistently outperforms Giraph, GraphLab and Blogel over all queries. It is 109, 8.3 and 45.2 times faster over Friendster, and 136.7, 5.8 and 20.8 times faster over DBpedia on average, respectively, when $n = 192$. (b) GRAPE scales better with the number n of workers than the others. (c) GRAPE takes at most 21 supersteps, while Giraph, GraphLab and Blogel take 38, 38 and 40 supersteps, respectively. This empirically validates the convergence guarantee of GRAPE under monotonic status variable updates and its positive effect on reducing parallel and communication cost.

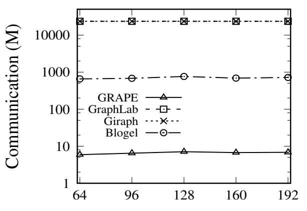
(4) SubIso. Fixing $|Q| = (3, 5)$, we evaluated subgraph isomorphism. As shown in Figures 3i-3j over Friendster and DBpedia, respectively, GRAPE is on average 34, 16 and 4 times faster than Giraph, GraphLab and Blogel when $n = 192$. It is 2.2 and 1.3 times faster when n varies from 64 to 192 over Friendster and DBpedia, respectively. This is comparable with GraphLab that is 1.4 and 2.8 times faster, respectively.

*(5) Collaborative filtering (*CF*).* We used movieLens [3] with a training set $|E_T| = 90\%|E|$. We compared GRAPE with the built-in CF in GraphLab, and with CF implemented for Giraph and Blogel. Note that CF favors "vertex-centric" programming since each node only needs to exchange data with their neighbors, as indicated by that GraphLab and Giraph outperform Blogel. Nonetheless, Figure 3k shows that GRAPE is on average 4.1, 2.6 and 12.4 times faster than Giraph, GraphLab and Blogel, respectively. Moreover, it scales well with n.

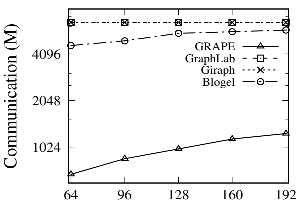
(6) Scale-up of GRAPE. The speed-up of a system may degrade over more workers [24]. We thus evaluate the scale-up of GRAPE, which measures the ability to keep the same performance when both the size of graph G (denoted as $(|V|, |E|)$) and the number n of workers increase proportionally. We varied n from 64 to 192, and



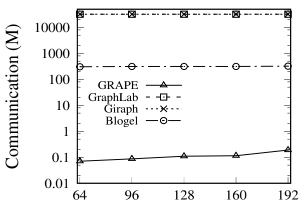
(a) Varying n: SSSP (traffic)



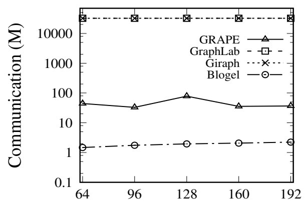
(b) Varying n:SSSP (UKWeb)



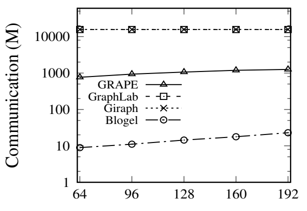
(c) Varying n: SSSP (Friendster)



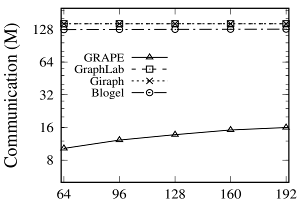
(d) Varying n: CC (traffic)



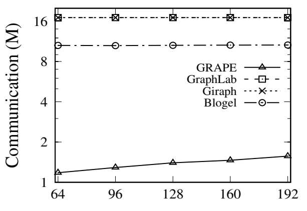
(e) Varying n: CC (UKWeb)



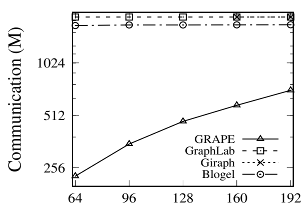
(f) Varying n: CC (Friendster)



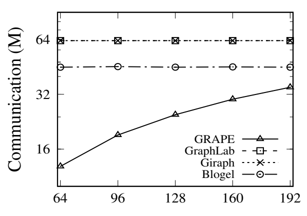
(g) Varying n: Sim (Friendster)



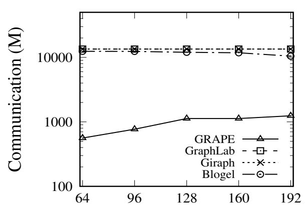
(h) Varying n: Sim (DBpedia)



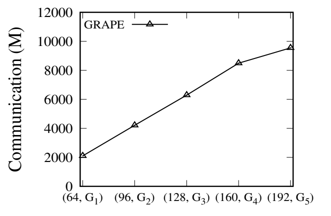
(i) Varying n:SubIso(Friendster)



(j) Varying n: SubIso (DBpedia)



(k) Varying n: CF(movieLens)



(l) SSSP (Synthetic)

Figure 4: Communication costs

for each n, deployed GRAPE over a synthetic graph. The graph size varies from $(50M, 500M)$ to $(250M, 2.5B)$ (denoted as G_5), with fixed ratio between edge number and node number and proportional to n . The scale up at *e.g.,* $(128, G_3)$ is the ratio of the time using 64 workers over G_1 to its counterpart using 128 workers over G_3. As shown in Fig. 3l, GRAPE preserves a reasonable scale-up (close to linear scale-up, the optimal scale-up).

Exp-2: Communication cost. The communication cost (in bytes) reported by Giraph, GraphLab and Blogel depends on their own implementation of message blocks and protocols [17], where Giraph ships less data (in bytes) than GraphLab over specific datasets and queries, and vice versa for others. For a fair and consistent comparison, we report the total number of exchanged messages.

In the same setting as Exp-1, Figure 4 reports the communication costs of the systems. We observe that Giraph and GraphLab ship roughly the same amount of messages. GRAPE incurs much less cost than Giraph and GraphLab. For SSSP with 192 workers, it ships on average 10%, 0.0017%, 45%, and 9.3% of the data shipped for Sim, CC, SubIso and CF by Giraph (GraphLab), respectively, and reduces their cost by 6 orders of magnitude! While it ships more data than Blogel for CC due to the precomputation of Blogel, it only ships 13.5%, 54%, 0.014% and 16% of the data shipped by Blogel for Sim, SubIso, SSSP and CF, respectively.

(1) SSSP. Figures 4a-4c show that both GRAPE and Blogel incurs communication costs that are orders of magnitudes less than those of GraphLab and Giraph. This is because vertex-centric programming incurs a large number of inter-vertex messages. Both block-centric programs (Blogel) and PIE programs (GRAPE) effectively reduce unnecessary messages, and trigger inter-block messages only when necessary. We also observe that GRAPE ships 0.94% and 17.9% of the data shipped by Blogel over UKWeb and Friendster, respectively. Indeed, GRAPE ships only updated values. This significantly reduces the amount of messages that need to be shipped.

(2) CC. Figures 4d-4f show similar improvement of GRAPE over GraphLab and Giraph for CC. It ships on average 0.0017% of the data shipped by Giraph and GraphLab. As Blogel precomputes CC (see Exp-1(2)), it ships little data. Nonetheless, GRAPE is not far worse than the near "optimal" case of Blogel, sometimes better.

(3) Sim. Figures 4g and 4h report the communication cost for graph simulation over Friendster and DBpedia, respectively. One can see that GRAPE ships substantially less data, *e.g.,* on average 8.5%, 8.5% and 11.4% of the data shipped by Giraph, GraphLab and Blogel, respectively. Observe that the communication cost of Blogel is much higher than that of GRAPE, even though it adopts inter-block communication. This shows that the extension of vertex-centric to block-centric by Blogel has limited improvement for more complex queries. GRAPE works better than these systems by employing incremental IncEval to reduce excessive messages.

(4) SubIso. Figures 4i and 4j report the results for SubIso over Friendster and DBpedia, respectively. On average, GRAPE ships 26%, 26% and 31% of the data shipped by Giraph, GraphLab and Blogel, respectively.

(5) CF. Figure 4k reports the result for CF over movieLens. On average, GRAPE ships 6.5%, 6.5% and 7.3% of the data shipped by Giraph, GraphLab and Blogel, respectively.

(6) Communication cost (synthetic). In the same setting as Figure 3l, Figure 4l reports the communication cost for SSSP. It takes more cost over larger graphs and more workers due to increased "border nodes", as expected.

Summary. Our experimental study verifies the findings in [13] using larger real-life graphs, with some new observations. (1) Over large-scale graphs, GRAPE remains comparable to state-of-the-art systems (Giraph, GraphLab, Blogel) by automatically parallelizing sequential algorithms. (2) GRAPE preserves reasonable scalability and demonstrates good scale-up when using more workers because its incremental computation mitigates the impact of more border nodes and fragments. In contrast, Blogel may take longer time with larger number of workers since its communication cost cancels out the improvement from block-centric parallelism. (3) GRAPE outperforms Giraph, GraphLab and Blogel in communication costs, by orders of magnitude on average.

4 Conclusion

We contend that GRAPE is promising for making parallel graph computations accessible to a large group of users, without degradation in performance or functionality. We have developed stronger fundamental results in connection with the convergence and expressive power of GRAPE, which will be reported in a later publication. We are currently extending GRAPE to support a variant of asynchronous parallel model (ASP).

Acknowledgments. Fan and Xu are supported in part by ERC 652976 and EPSRC EP/M025268/1. Fan, Luo and Yu are supported by Shenzhen Peacock Program 1105100030834361, NSFC 61421003, 973 Program 2014CB340302, Beijing Advanced Innovation Center for Big Data and Brain Computing, and NSFC Foundation for Innovative Research Groups. Wu is supported by NSF IIS 1633629 and Google Faculty Research Award.

References

[1] DBpedia. *http://wiki.dbpedia.org/Datasets.*

[2] Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*

[3] Movielens. *http://grouplens.org/datasets/movielens/.*

[4] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*

[5] UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/*, 2006.

[6] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *SIGKDD*, pages 1456–1465, 2014.

[7] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[9] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.

[10] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.

[11] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.

[12] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang. GRAPE: Parallelizing sequential graph computations. In *VLDB (demo)*, 2017. *http://grapedb.io/*.

[13] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, B. Zhang, Z. Zheng, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, 2017.

[14] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.

[15] 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 *OSDI*, pages 599–613, 2014.

[16] I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. Collecting and analyzing data from E-Government Facebook pages. In *ICT Innovations*, 2014.

[17] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *VLDB*, 7(12), 2014.

[18] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[19] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), 1996.

[20] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.

[21] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[22] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.

[23] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[24] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what cost? In *HotOS*, 2015.

[25] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[26] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

[27] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, 2013.

[28] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.

[29] Y. Tian, A. Balmin, S. A. Corsten, and J. M. Shirish Tatikonda. From "think like a vertex" to "think like a graph". *PVLDB*, 7(7):193–204, 2013.

[30] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[31] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol A*. 1990.

[32] J. Vinagre, A. M. Jorge, and J. Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In *UMAP*, 2014.

[33] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[34] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

Towards A Unified Graph Model for Supporting Data Management and Usable Machine Learning

Guoliang Li[*], Meihui Zhang[+], Gang Chen[†], Beng Chin Ooi[‡]
[*]Department of Computer Science, Tsinghua University, Beijing, China
[+]Department of Computer Science, Beijing Institute of Technology, Beijing, China
[†]Department of Computer Science, Zhejiang University, Hangzhou, China
[‡]School of Computing, National University of Singapore, Singapore

Abstract

Data management and machine learning are two important tasks in data science. However, they have been independently studied so far. We argue that they should be complementary to each other. On the one hand, machine learning requires data management techniques to extract, integrate, clean the data, to support scalable and usable machine learning, making it user-friendly and easily deployable. On the other hand, data management relies on machine learning techniques to curate data and improve its quality. This requires database systems to treat machine learning algorithms as their basic operators, or at the very least, optimizable stored procedures. It poses new challenges as machine learning tasks tend be iterative and recursive in nature, and some models have to be tweaked and retrained. This calls for a reexamination of database design to make it machine learning friendly.

In this position paper, we present a preliminary design of a graph model for supporting both data management and usable machine learning. To make machine learning usable, we provide a declarative query language, that extends SQL to support data management and machine learning operators, and provide visualization tools. To optimize data management procedures, we devise graph optimization techniques to support a finer-grained optimization than traditional tree-based optimization model. We also present a workflow to support machine learning (ML) as a service to facilitate model reuse and implementation, making it more usable and discuss emerging research challenges in unifying data management and machine learning.

1 Introduction

A data science workflow includes data extraction, data integration, data analysis, machine learning and interpretation. For example, consider healthcare analytics. Heterogeneity, timeliness, complexity, noise and incompleteness with big data impede the progress of creating value from electronic healthcare data [10]. We need to extract high quality data from multiple sources, clean the data to remove the inconsistency and integrate the heterogeneous data. Next we can use data analytics techniques to discover useful information and analyze the data to find interesting results (e.g., cohort analysis [7]). We also need to utilize machine learning techniques to

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

learn important features and make decision (e.g., Readmission risk [25]). Finally, we need to present the results in the medical context to enable users to better appreciate the findings.

The data science workflow (as shown in Figure 1) contains both data management components (data extraction, data integration, data analysis) and machine learning component. However, data management and machine learning have been independently studied, be it in industry or academia, even though they are in fact complementary to each other. First, machine learning requires high quality data to guarantee learning results. Thus it needs data management techniques to extract, integrate and clean the data. Second, machine learning relies on data management techniques to enable scalable machine learning. It is widely recognized that machine learning is hard for users without machine learning background to use while database has been featured prominently and used as the underlying system in many applications. Thus machine learning should borrow ideas from the database to make machine-learning algorithms usable and easily deployable. Third, many data management components rely on machine learning techniques to improve the quality of data extraction and integration. For example, we can use machine learning to learn the features that play important roles in data integration. This calls for a new framework to support data management and machine learning simultaneously, and their interaction and dependency in big data analytics.

To address these problems, we propose a unified graph model for modeling both data management and machine learning. We use a graph to model relational data, where nodes are database tuples and edges are foreign key relationships between tuples. A graph can be used to model the unstructured data, semi-structured data and structured data together by linking the tuples in relational database, documents or objects in unstructured data and semi-structured data. More importantly, most machine learning algorithms work on graph. We can therefore use a unified graph model to support both data management and machine learning. Hence, we propose a graph-based framework that integrates data management and machine learning together, which (1) supports data extraction, data integration, data analysis, SQL queries, and machine learning simultaneously and (2) provide machine learning (ML) as a service such that users without machine learning background can easily use machine learning algorithms.

In summary, we cover the following in this position paper.

(1) We sketch a unified graph model to model unstructured data, semi-structured data, and structured data and can support both data management and machine learning.

(2) We propose a user-friendly interface for the users to use data management and machine learning. We provide a declarative query language that extends SQL to support data management and machine learning operators. We also provide visualization tools to assist users in using the machine learning algorithms.

(3) We devise graph optimization techniques to improve the performance, which provide a finer-grained optimization than traditional tree-based optimization model.

(4) We discuss design principles and the important components in providing machine learning as a service workflow.

(5) We discuss research challenges in unifying data management and machine learning.

2 A Unified Graph Model

Graph Model. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ includes a vertex set $\mathcal{V}$ and an edge set $\mathcal{E}$, where a vertex can be a database tuple, a document, or a user, and an edge is a relationship between two vertices, e.g., foreign key between tuples for structured data, hyperlink between documents for unstructured data, or friendship among users in semi-structured data (e.g., social networks).

Graph Model for Relational Data. Given a database $\mathcal{D}$ with multiple relation tables $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n$, we can model the relational data as a graph, where each tuple is a vertex and there is an edge between two vertexes if their corresponding tuples have foreign key relationships [12, 14]. Given a SQL query q, we find compact subtrees corresponding to the query from the graph as answers [14] and the details will be discussed in Section 4.

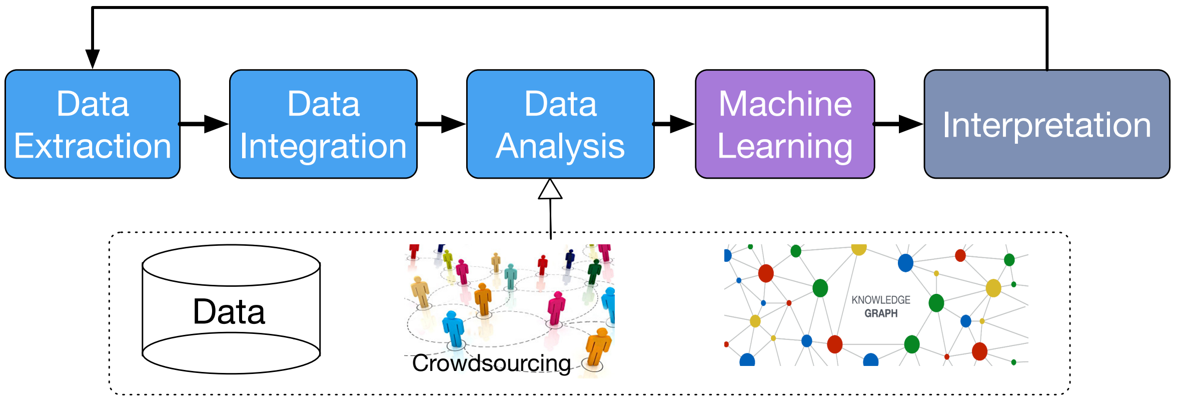


Figure 1: Workflow of Data Science.

Graph Model for Heterogeneous Data. Given heterogeneous data consisting of structured data, semi-structured data and unstructured data, we can also use a unified graph model to represent them. For unstructured data, e.g., html documents, the vertexes are documents and the edges are hyperlinks between documents. For semi-structured data, e.g., Facebook, the vertexes are users and the edges are friendships between users. We can also link the heterogeneous data together using the data extraction and integration techniques [4, 14].

Data Extraction. Data extraction aims to extract entities from the data, including documents, lists, tables, etc. There are some variants of entity extraction. Firstly, given a dictionary of entities and a document, it aims to extract entities from the document that exactly or approximately match the predefined entities in the dictionary [13]. Secondly, given a document and entities in knowledge bases, it aims to link the entities in knowledge base to the entities in the document to address the entity ambiguity problem [19]. Thirdly, it utilizes rules to identify the entities, e.g., a person born in a city, a person with PhD from a university [20]. We can use data extraction techniques to extract entities, based on which we can integrate the heterogeneous data.

Data Integration. We can also link the structured data and unstructured data together and use a unified graph model to represent the data. There are several ways to link the data. First, we can use a similarity-based method [8]. If two objects from different sources have high similarity, we can link them together. Second, we can use crowd-based method to link the data [3]. Given two data from different sources, we ask the crowd to label whether they can be linked. Since crowdsourcing is not free, it is expensive to ask the crowd to check every pair of data and we can use some reasoning techniques to reduce the crowd cost, e.g., transitivity. Third, we can use a knowledge-based method to link the data. We map the data from different sources to entities in knowledge bases and the data mapped to the same entity can be linked [11].

Graph Model for Machine Learning. Most machine learning algorithms adopt a graph model, e.g., PageRank, probabilistic graphical model, neural network, etc.

To summarize, we can utilize a unified graph to model heterogeneous data which can support both data management (including SQL queries, data extraction, data integration) and machine learning (any graph based learning algorithms).

3 A Versatile Graph Framework

We propose a versatile graph framework to support both data management and machine learning simultaneously with a user-friendly interface.

Query Interace. A natural challenge is to utilize the same query interface to support both data management and

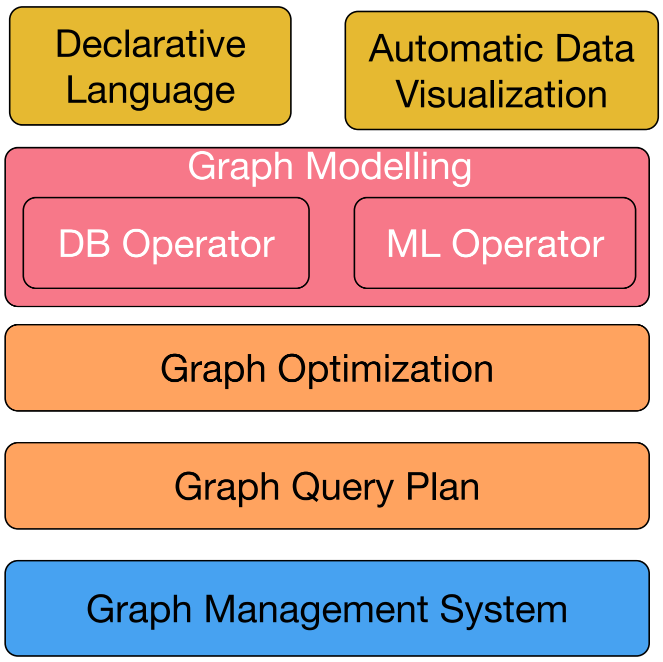


Figure 2: A Unified Graph Model and System.

machine learning. We aim to support two types of query interfaces: declarative query language and visualization tools. The declarative query language extends traditional SQL to support data analysis and machine learning. For example, we can use UDF to support machine learning. We can also add native keywords into declarative query language to enable in-core optimization. Thus users only need to specify what they want to obtain but do not need to know how to get the result. Our framework supports both of the two cases. To support the second case, we need to abstract the machine learning operators, e.g., regression, clustering, classification, correlation. Then our framework supports all the database operators, e.g., selection, join, group, sort, top-k, and machine learning operators, e.g., clustering, classification, correlation, regression, etc. The visualization tool can help users to better understand the data and can also use visualization charts as examples (e.g., line charts, bar charts, pie charts) to provide users with instant feedback, and the framework can automatically compute similar trends or results. More importantly, users can utilize visualization tools, e.g., adding or deleting an attribute like tableau, to visualize the data. Our framework can also automatically analyze the data and suggest the visualization charts. Since the data and features are very complicated, we need to utilize the machine learning techniques to learn the features, decide which charts are interesting and which charts are better than others.

Graph Query Plan. Based on the graph query, we generate a graph query plan. We use a workflow to represent the query plan, which includes multiple query operators (including data management and machine learning operators) and the query operators have some relationships (e.g., precedence order, dependency, decision relation). A straightforward method directly generates the physical plan, executes each query operators based on the relationships, and executes the physical plan without optimization. Note that it is important to select a good query plan to improve the performance and scalability.

Graph Optimization. Traditional database system employs a tree-based optimization model. It employs a table-level coarse-grained optimization, which selects an optimized table-level join order to execute the query. The motivation is to reduce the random access in disk-based setting. However a table-level join order may not be optimal, because different tuples may have different optimal join orders. In disk-based setting, it is hard to get the optimal order for different tuples. However, in memory setting, we have an opportunity to attempt more effective

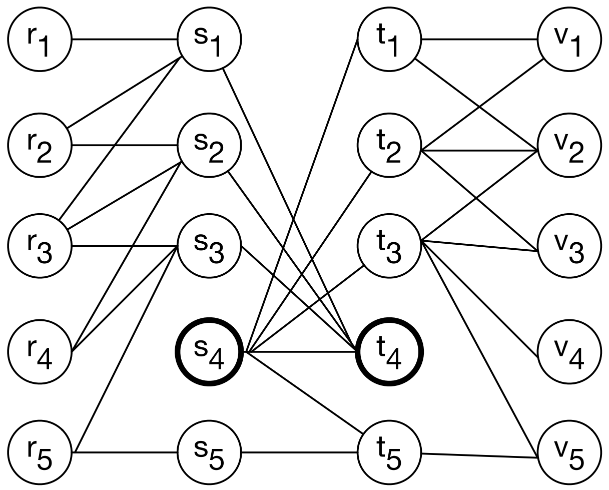


Figure 3: A Graph-Based Optimization Model.

optimizations. To address this problem, we propose a fine-grained tuple-level optimization model, which can find the best order for different tuples (see Section 4). We can also provide graph optimization techniques for other operations.

Graph Management System. There are many graph systems to support graph-based query processing, including disk-based graph systems, in-memory graph systems, and distributed graph systems [6, 5, 9, 18, 6, 24]. Disk-based systems optimize the model to reduce random access. In-memory graph systems utilize the transaction techniques to guarantee high parallelism. Distributed graph systems include synchronized model and asynchronized model, where the former has straggler and imbalance problem while the latter is hard to implement. Our framework should support any graph management system and we focus on automatic suggestion of the best graph execution paths to support different applications.

4 Fine-Grained Graph Processing Model

Given a SQL query, we utilize the graph to directly find the answers of the SQL query. We first consider a simple case that the tables are joined by a chain structure, i.e., each table is joined with at most two other tables and there is no cycle. There are some other join structures, e.g., star join structure and graph join with cycles, while will be discussed later [12].) Suppose there are x join predicates in the SQL. The answers of such SQL query are chains of the graph with x edges.

For example, consider four tables, R, S, T, V in Figure 3 and each table has five tuples. The tuples are connected based on foreign keys. Consider a SQL query with 3 join predicates to join the four tables, e.g., `Select * from R,S,T,V where` $R.A = S.A$, $S.B = T.B$ `and` $T.C = V.C$. The answers of the SQL query are chains in the graph that contain 3 edges and 4 tuples such that the tuples are from different tables. Traditional database systems employ a tree-based optimization model, possibly due to the disk-based optimization requirement. The tree-based optimization model employs a coarse-grained optimization and selects the best table-level join order. For example, the tree model first checks the join predicates between the first two tables and then joins with the third and the fourth tables. The cost of the tree based model is 28. Obviously we can select different best orders for different tuples. For example, we only need to check s_4, t_4 and r_5, s_5, t_5, v_4. The cost is only 6. Thus the graph-based optimization order has much lower cost and can provide finer-grained optimizations.

Graph-Based Optimization Techniques. We can first partition the graph into multiple disjoint connected components. The vertexes in different components cannot form a result. Then we can compute answers from each connected component. We assign each vertex in the graph with a pruning power, i.e., if the vertex is checked, how many tuples will be pruned. For example, the pruning power of s_4 is 8 and the pruning power of t_4 is also 8. Thus we can first check s_4 and t_4. Then we can prune r_1, r_2, r_3, r_4, t_1, t_2, t_3, t_4, s_1, s_2, s_3, s_4, v_1, v_2, v_3, v_4. Then we can get only one result (r_4, s_5, t_5, v_5).

Next we consider other join structures of queries.

Tree Join Structure. The tables are joined by a tree structure and there is no cycle. We can transform it into a chain structure as follows. We first find the longest chain in the tree. Suppose the chain is $T_1, T_2, \cdots, T_x$. Then for each vertex T_i on the chain, which (indirectly) connects other vertices $T'_1, T'_2, \cdots, T'_y$ that are not on the chain, we insert these vertices into the chain using a recursive algorithm. If these vertices are on a chain, i.e., $T_i, T'_1, T'_2, \cdots, T'_y$, then we insert them into the chain by replacing T_i with $T_i, T'_1, T'_2, \cdots, T'_{y-1}, T'_y$, T'_{y-1}, $\cdots, T_i$. If these vertices are not on a chain, we find the longest chain and insert other vertices not on the chain into this chain using the above method. In this way, we can transform a tree join structure to a chain structure. Note that the resulting chain has some duplicated tables. Hence, joining those tables may result in invalid join tuples (e.g., a join tuple that uses one tuple in the first copy of T_i, and a different tuple in the second copy of T_i). We need to remove those invalid join tuples.

Graph Join Structure. The tables are joined by a graph structure, i.e., there exist cycles in the join structure. We can transform it into a tree structure. For example, given a cycle $(T_1, T_2, \cdots, T_x, T_1)$, we can break the cycle by inserting a new vertex T'_1 and replacing it with T_1. Thus we can transform a cycle to a tree structure, by first finding a spanning tree of the graph using breadth first search, and breaking all non-tree edges.

In summary, given a graph query, a SQL query, a machine learning query, we can use a unified graph model and system to answer the query efficiently. However there are still many open problems in this unified graph based query optimization.

(1) How to build efficient indexes to support various SQL and machine learning queries?

(2) How to support both transactions in data management components and data analytics in machine learning component simultaneously?

(3) How to support concurrency control?

(4) How to support iterative processing?

5 ML As A Service

In 1970s, database was proposed to manage a collection of data. Database has been widely accepted and deployed in many applications because it is easy to use due to its user-friendly declarative query language. In 1990s, search engine was proposed to help Internet users to explore web data, which is also widely used by lay users. Although machine learning is very hot in recent years (every vertical domains claim that they want to use machine learning to improve the performance) and several machine learning systems have been deployed [17, 22, 21, 1, 16, 23], it is still not widely used by non-machine-learning users, because (1) machine learning requires users to understand the underlying model; (2) machine learning requires experts to tune the model to learn the parameters; (3) there is no user-friendly query interface for users to use machine learning; (4) some machine learning algorithms are not easy to explain and users may not appreciate the learning results that are hard to interpret. Stanford also launches a project DAWN on providing infrastructures for machine learning [2]. Different from DAWN, we focus on providing machine learning as a service, which enables ordinary users adopt the machine learning techniques and easily deploy machine learning applications.

Next we shall outline the design requirements.

(1) Scalability. The framework should scale up/out well and scale on volume and dimensionality.

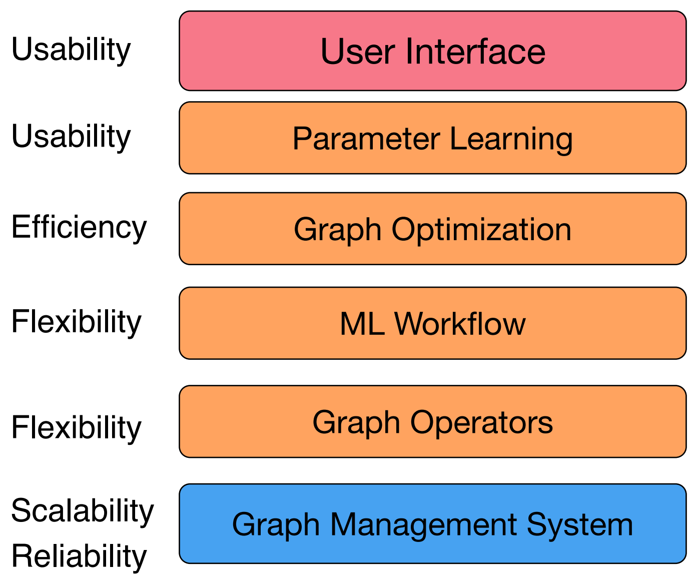


Figure 4: ML as a service

(2) Efficiency. The framework should have low latency and high throughput and we also need to balance the workload.

(3) Reliability. The framework can easily recover for data/machine/network failures.

(4) Flexibility. The framework can support various learning applications.

(5) Usability. The framework should be easy to use for any users. We need to have good abstraction and programming model.

Based on these five factors, we design a new graph-based framework to support ML as a service (as shown in Figure 4).

Encapsulated Operators. We need to encapsulate the machine learning operators so that ordinary users can easily use the operators to accomplish machine learning tasks. For each operator, we also encapsulate multiple algorithms to support the operator. Note that the algorithms are transparent to the end users. For example, we encapsulate clustering, classification, and regression operators. For clustering, we encapsulate multiple clustering algorithms, e.g., K-means, density-based clustering, hierarchical clustering, etc. The users can utilize a single operator or multiple operators to support various applications.

ML Workflow. Each ML operator can support some simple applications (e.g., clustering), and there are some complex applications that require multiple operators (e.g., healthcare). Thus we need to design a model to support applications with multiple operators. We can use a machine learning workflow to describe the model, where each node is an operator and each directed edge is a relationship between two operators. Then we can optimize a machine learning query based on the workflow. (1) We can select the appropriate algorithm for each operator for different applications. (2) We can change the order of two operators to optimize the workflow. (3) We can optimize the structure of the workflow. We can also devise cost-based model and rule-based model to optimize the ML workflow.

Parameter Learning. Machine learning algorithms contain many parameters that significantly affect the performance. Thus it is important to get good parameter values in different applications. However it is tedious and complicated to tune the parameters and therefore, experts are required to be involved to tune the parameters. This is an important reason why users without machine learning background are hard to use the ML algorithms.

Thus it is important to automatically learn the parameter values. In addition, it is important to automate the discovery of data transformation and automatically discover the data drift.

Human-in-the-loop Machine Learning. Machine learning requires to use high-quality labelled data to learn the results. However it is hard to use purely machine learning algorithms. Thus we propose human-in-the-loop machine learning, which utilizes both crowdsourcing and experts to improve machine learning [15]. We use crowd workers to provide high quality labelled data. We can ask the experts to guide the feature selection and parameter learning. Most importantly, we can use active learning techniques to decide when/where to use crowd and when/where to use experts.

6 Research Challenges

Explainable Machine Learning. Most existing machine-learning algorithms function like blackbox and the learning results are not easy to explain, e.g., SVM. As many applications and users require to understand the results to make decision, they require the algorithms to be explainable. For example, in healthcare analytics, the doctors should be convinced why machine-learning results are meaningful and applicable. Thus we require to design explainable techniques to help users better understand machine learning algorithms. We can utilize visualization techniques to visualize the learning process and results. We can also deign new explainable machine learning models.

End-to-end Learning and Deployment. It is expensive to learn the machine learning model and tune the parameters. It is rather hard to deploy the machine learning system for users without machine learning background. Thus it requires to build end-to-end systems that automatically learn the model, tune the parameters, and deploy the system for various applications. The system can also adapt to various domains and involves manual intervention as little as possible.

Incremental Machine Learning. In many applications, machine learning employs a standalone model, which loads the data outside the repository and runs the model out-of-core. However if the data is updated, the method needs to reload the data and run the model from scratch. Obviously this method is not efficient. Thus it calls for incremental learning model that utilizes the original model and the updated data to learn the new model without needing to relearn the model.

Machine Learning on New Hardware. With the development of new hardware, e.g., NVM, GPU, FPGA, RDMA, these new hardware pose new challenges in data management and machine learning. Thus we require to design new techniques and utilize the features of new hardware inherently to improve the performance and scalability of machine learning algorithms. We need to extend the graph model, graph algorithms and graph system to support the new hardware, and devise device-aware optimization techniques.

Graph Processing. Distributed graph processing has straggler and imbalance problems, and we have to design more efficient graph management systems and optimization techniques. Moreover, existing systems focus on iteration graph processing applications, they are expensive for some non-iterative graph applications, e.g., computing shortest path. Thus we need to design new efficient graph systems and framework to support general graph queries.

Machine Learning for Data Integration. We can use machine learning techniques to facilitate data integration. For example, we can use deep learning and embedding techniques to find candidate matching entity/column pairs in data integration. We can also use active learning techniques to reduce the monetary cost. There are two big challenges in using machine learning techniques. The first is to find a large training data to feed the machine learning algorithms. The second is to design new machine learning models for data integration.

7 Conclusion

In this position paper, we sketched a unified graph model for supporting both data management and machine learning. We proposed to provide a user-friendly interface for users to easily use data management and machine learning. We outlined graph-based optimization techniques to improve the performance, which provides a finer-grained optimization than traditional tree-based optimization model. We discussed machine learning as a service and presented a ML workflow with the aim of achieving high flexibility. We discussed emerging research challenges in unifying data management and machine learning.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Infrastructure for usable machine learning: The stanford DAWN project. *CoRR*, abs/1705.07538, 2017.

[3] C. Chai, G. Li, J. Li, D. Deng, and J. Feng. Cost-effective crowdsourced entity resolution: A partial-order approach. In *SIGMOD*, pages 969–984, 2016.

[4] D. Deng, G. Li, J. Feng, Y. Duan, and Z. Gong. A unified framework for approximate dictionary-based entity extraction. *VLDB J.*, 24(1):143–167, 2015.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[6] 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 *OSDI*, pages 599–613, 2014.

[7] D. Jiang, Q. Cai, G. Chen, H. V. Jagadish, B. C. Ooi, K. Tan, and A. K. H. Tung. Cohort query processing. *PVLDB*, 10(1):1–12, 2016.

[8] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[9] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[10] C. Lee, Z. Luo, K. Y. Ngiam, M. Zhang, K. Zheng, G. Chen, B. C. Ooi, and W. L. J. Yip. Big healthcare data analytics: Challenges and applications. In *Handbook of Large-Scale Distributed Computing in Smart Healthcare*, pages 11–41. Springer, 2017.

[11] G. Li. Human-in-the-loop data integration. *PVLDB*, 10(12):2006–2017, 2017.

[12] G. Li, C. Chai, J. Fan, X. Weng, J. Li, Y. Zheng, Y. Li, X. Yu, X. Zhang, and H. Yuan. CDB: optimizing queries with crowd-based selections and joins. In *SIGMOD*, pages 1463–1478, 2017.

[13] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD*, pages 529–540, 2011.

[14] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[15] B. C. Ooi, K. Tan, Q. T. Tran, J. W. L. Yip, G. Chen, Z. J. Ling, T. Nguyen, A. K. H. Tung, and M. Zhang. Contextual crowd intelligence. *SIGKDD Explorations*, 16(1):39–46, 2014.

[16] B. C. Ooi, K. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. H. Tung, Y. Wang, Z. Xie, M. Zhang, and K. Zheng. SINGA: A distributed deep learning platform. In *SIGMM*, pages 685–688, 2015.

[17] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2503–2511, 2015.

[18] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng. Graph analytics through fine-grained parallelism. In *SIGMOD*, pages 463–478, 2016.

[19] Z. Shang, Y. Liu, G. Li, and J. Feng. K-join: Knowledge-aware similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(12):3293–3308, 2016.

[20] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[21] W. Wang, G. Chen, H. Chen, T. T. A. Dinh, J. Gao, B. C. Ooi, K. Tan, S. Wang, and M. Zhang. Deep learning at scale and at ease. *TOMCCAP*, 12(4s):69:1–69:25, 2016.

[22] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K. Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Record*, 45(2):17–22, 2016.

[23] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *SIGKDD*, pages 1335–1344, 2015.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[25] K. Zheng, J. Gao, K. Y. Ngiam, B. C. Ooi, and J. W. L. Yip. Resolving the bias in electronic medical records. In *KDD*, pages 2171–2180, 2017.

Representation Learning on Graphs: Methods and Applications

William L. Hamilton
wleif@stanford.edu

Rex Ying
rexying@stanford.edu

Jure Leskovec
jure@cs.stanford.edu

Department of Computer Science
Stanford University
Stanford, CA, 94305

Abstract

Machine learning on graphs is an important and ubiquitous task with applications ranging from drug design to friendship recommendation in social networks. The primary challenge in this domain is finding a way to represent, or encode, graph structure so that it can be easily exploited by machine learning models. Traditionally, machine learning approaches relied on user-defined heuristics to extract features encoding structural information about a graph (e.g., degree statistics or kernel functions). However, recent years have seen a surge in approaches that automatically learn to encode graph structure into low-dimensional embeddings, using techniques based on deep learning and nonlinear dimensionality reduction. Here we provide a conceptual review of key advancements in this area of representation learning on graphs, including matrix factorization-based methods, random-walk based algorithms, and graph convolutional networks. We review methods to embed individual nodes as well as approaches to embed entire (sub)graphs. In doing so, we develop a unified framework to describe these recent approaches, and we highlight a number of important applications and directions for future work.

1 Introduction

Graphs are a ubiquitous data structure, employed extensively within computer science and related fields. Social networks, molecular graph structures, biological protein-protein networks, recommender systems—all of these domains and many more can be readily modeled as graphs, which capture interactions (*i.e.*, edges) between individual units (*i.e.*, nodes). As a consequence of their ubiquity, graphs are the backbone of countless systems, allowing relational knowledge about interacting entities to be efficiently stored and accessed [2].

However, graphs are not only useful as structured knowledge repositories: they also play a key role in modern machine learning. Machine learning applications seek to make predictions, or discover new patterns, using graph-structured data as feature information. For example, one might wish to classify the role of a protein in a biological interaction graph [28], predict the role of a person in a collaboration network, recommend new friends to a user in a social network [3], or predict new therapeutic applications of existing drug molecules, whose structure can be represented as a graph [21].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

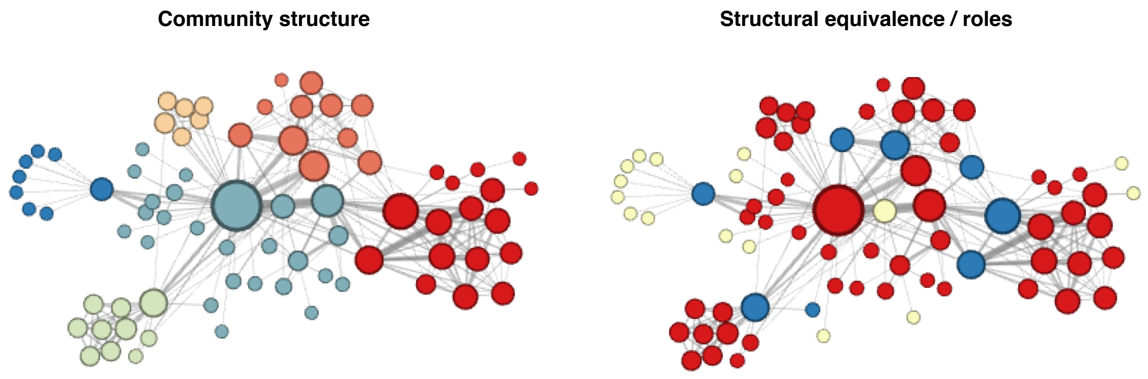


Figure 1: Two different views of a character-character interaction graph derived from the Les Misérables novel, where two nodes are connected if the corresponding characters interact. The coloring in the left figure emphasizes differences in the nodes' global positions in the graph: nodes have the same color if they belong to the same community, at a global level. In contrast, the coloring in the right figure denotes structural equivalence between nodes, or the fact that two nodes play similar roles in their local neighborhoods (*e.g.*, "bridging nodes" are colored blue). The colorings for both figures were generated using different settings of the node2vec node embedding method [27], described in Section 2. Reprinted from [27] with permission.[1]

The central problem in machine learning on graphs is finding a way to incorporate information about the structure of the graph into the machine learning model. For example, in the case of link prediction in a social network, one might want to encode pairwise properties between nodes, such as relationship strength or the number of common friends. Or in the case of node classification, one might want to include information about the global position of a node in the graph or the structure of the node's local graph neighborhood (Figure 1), and there is no straightforward way to encode this information into a feature vector.

To extract structural information from graphs, traditional machine approaches often rely on summary graph statistics (*e.g.*, degrees or clustering coefficients) [6], kernel functions [57], or carefully engineered features to measure local neighborhood structures [39]. However, these approaches are limited because these hand-engineered features are inflexible—*i.e.*, they cannot adapt during the learning process—and designing these features can be a time-consuming and expensive process.

More recently, there has been a surge of approaches that seek to *learn* representations that encode structural information about the graph. The idea behind these *representation learning* approaches is to learn a mapping that embeds nodes, or entire (sub)graphs, as points in a low-dimensional vector space, $\mathbb{R}^d$. The goal is to optimize this mapping so that geometric relationships in this learned space reflect the structure of the original graph. After optimizing the embedding space, the learned embeddings can be used as feature inputs for downstream machine learning tasks. The key distinction between representation learning approaches and previous work is how they treat the problem of capturing structural information about the graph. Previous work treated this problem as a pre-processing step, using hand-engineered statistics to extract structural information. In contrast, representation learning approaches treat this problem as machine learning task itself, using a data-driven approach to learn embeddings that encode graph structure.

Here we provide an overview of recent advancements in representation learning on graphs, reviewing techniques for representing both nodes and entire subgraphs. Our survey attempts to merge together multiple, disparate lines of research that have drawn significant attention across different subfields and venues in recent

[1]For this and all subsequent reprinted figures, the original authors retain their copyrights, and permission was obtained from the corresponding author.

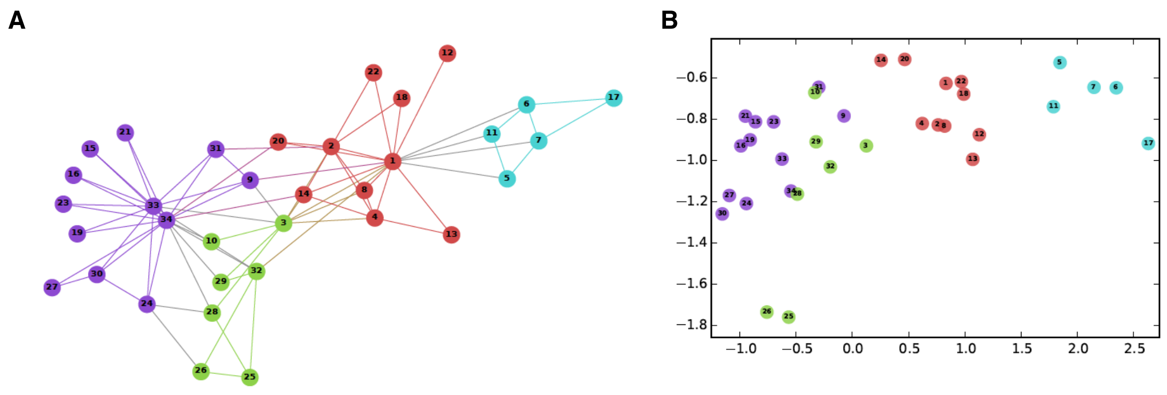


Figure 2: **A**, Graph structure of the Zachary Karate Club social network, where nodes are connected if the corresponding individuals are friends. The nodes are colored according to the different communities that exist in the network. **B**, Two-dimensional visualization of node embeddings generated from this graph using the DeepWalk method (Section 2.2.2) [46]. The distances between nodes in the embedding space reflect proximity in the original graph, and the node embeddings are spatially clustered according to the different color-coded communities. Reprinted with permission from [46, 48].

years—*e.g.*, node embedding methods, which are a popular object of study in the data mining community, and graph convolutional networks, which have drawn considerable attention in major machine learning venues. In doing so, we develop a unified conceptual framework for describing the various approaches and emphasize major conceptual distinctions.

We focus our review on recent approaches that have garnered significant attention in the machine learning and data mining communities, especially methods that are scalable to massive graphs (*e.g.*, millions of nodes) and inspired by advancements in deep learning. Of course, there are other lines of closely related and relevant work, which we do not review in detail here—including latent space models of social networks [32], embedding methods for statistical relational learning [42], manifold learning algorithms [37], and geometric deep learning [7]—all of which involve representation learning with graph-structured data. We refer the reader to [32], [42], [37], and [7] for comprehensive overviews of these areas.

1.1 Notation and essential assumptions

We will assume that the primary input to our representation learning algorithm is an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with associated binary adjacency matrix, $\mathbf{A}$.[2] We also assume that the methods can make use of a real-valued matrix of node attributes $\mathbf{X} \in \mathbb{R}^{m \times |\mathcal{V}|}$ (*e.g.*, representing text or metadata associated with nodes). The goal is to use the information contained in $\mathbf{A}$ and $\mathbf{X}$ to map each node, or a subgraph, to a vector $\mathbf{z} \in \mathbb{R}^d$, where $d << |\mathcal{V}|$.

Most of the methods we review will optimize this mapping in an *unsupervised* manner, making use of only information in $\mathbf{A}$ and $\mathbf{X}$, without knowledge of the downstream machine learning task. However, we will also discuss some approaches for *supervised* representation learning, where the models make use of classification or regression labels in order to optimize the embeddings. These classification labels may be associated with individual nodes or entire subgraphs and are the prediction targets for downstream machine learning tasks (*e.g.*, they might label protein roles, or the therapeutic properties of a molecule, based on its graph representation).

[2]Most of the methods we review are easily generalized to work with weighted or directed graphs, and we will explicitly describe how to generalize certain methods to the multi-modal setting (*i.e.*, differing node and edge types).

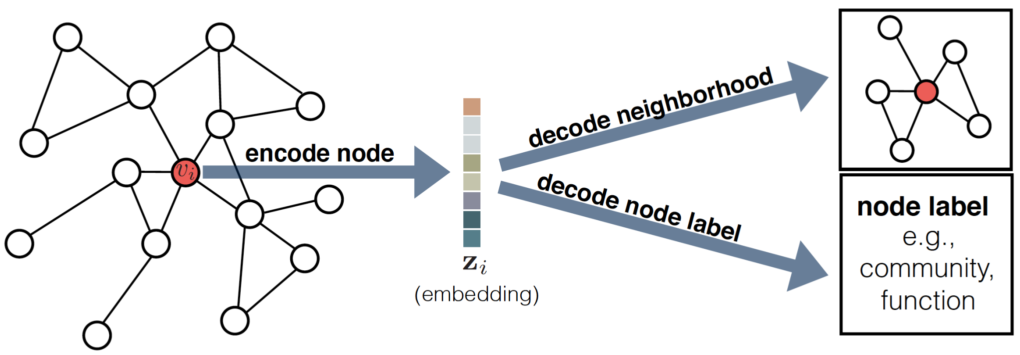


Figure 3: Overview of the encoder-decoder approach. First the encoder maps the node, v_i, to a low-dimensional vector embedding, $\mathbf{z}_i$, based on the node's position in the graph, its local neighborhood structure, and/or its attributes. Next, the decoder extracts user-specified information from the low-dimensional embedding; this might be information about v_i's local graph neighborhood (*e.g.*, the identity of its neighbors) or a classification label associated with v_i (*e.g.*, a community label). By jointly optimizing the encoder and decoder, the system learns to compress information about graph structure into the low-dimensional embedding space.

2 Embedding nodes

We begin with a discussion of methods for *node embedding*, where the goal is to encode nodes as low-dimensional vectors that summarize their graph position and the structure of their local graph neighborhood. These low-dimensional embeddings can be viewed as encoding, or projecting, nodes into a latent space, where geometric relations in this latent space correspond to interactions (*e.g.*, edges) in the original graph [32]. Figure 2 visualizes an example embedding of the famous Zachary Karate Club social network [46], where two dimensional node embeddings capture the community structure implicit in the social network.

2.1 Overview of approaches: An encoder-decoder perspective

Recent years have seen a surge of research on node embeddings, leading to a complicated diversity of notations, motivations, and conceptual models. Thus, before discussing the various techniques, we first develop a unified *encoder-decoder* framework, which explicitly structures this methodological diversity and puts the various methods on equal notational and conceptual footing.

In this framework, we organize the various methods around two key mapping functions: an *encoder*, which maps each node to a low-dimensional vector, or embedding, and a *decoder*, which decodes structural information about the graph from the learned embeddings (Figure 3). The intuition behind the encoder-decoder idea is the following: if we can learn to decode high-dimensional graph information—such as the global positions of nodes in the graph and the structure of local graph neighborhoods—from encoded low-dimensional embeddings, then, in principle, these embeddings should contain all information necessary for downstream machine learning tasks.

Formally, the *encoder* is a function,

$$\text{ENC} : \mathcal{V} \rightarrow \mathbb{R}^d, \tag{1}$$

that maps nodes to vector embeddings, $\mathbf{z}_i \in \mathbb{R}^d$ (where $\mathbf{z}_i$ corresponds to the embedding for node $v_i \in \mathcal{V}$). The *decoder* is a function that accepts a set of node embeddings and decodes user-specified graph statistics from these embeddings. For example, the decoder might predict the existence of edges between nodes, given their embeddings [1, 35], or it might predict the community that a node belongs to in the graph [28, 34] (Figure 3). In principle, many decoders are possible; however, the vast majority of works use a basic *pairwise decoder*,

$$\text{DEC} : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^+, \tag{2}$$

that maps pairs of node embeddings to a real-valued graph proximity measure, which quantifies the proximity of the two nodes in the original graph.

When we apply the pairwise decoder to a pair of embeddings ($\mathbf{z}_i$,$\mathbf{z}_j$) we get a *reconstruction* of the proximity between v_i and v_j in the original graph, and the goal is optimize the encoder and decoder mappings to minimize the error, or loss, in this reconstruction so that:

$$\text{DEC}(\text{ENC}(v_i), \text{ENC}(v_j)) = \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \approx s_{\mathcal{G}}(v_i, v_j), \tag{3}$$

where $s_{\mathcal{G}}$ is a user-defined, graph-based proximity measure between nodes, defined over the graph, $\mathcal{G}$. For example, one might set $s_{\mathcal{G}}(v_i, v_j) \triangleq \mathbf{A}_{i,j}$ and define nodes to have a proximity of 1 if they are adjacent and 0 otherwise [1], or one might define s_G according to the probability of v_i and v_j co-occurring on a fixed-length random walk over the graph $\mathcal{G}$ [27, 46]. In practice, most approaches realize the reconstruction objective (Equation 3) by minimizing an empirical loss, $\mathcal{L}$, over a set of training node pairs, $\mathcal{D}$:

$$\mathcal{L} = \sum_{(v_i,v_j)\in\mathcal{D}} \ell\left(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j), s_{\mathcal{G}}(v_i, v_j)\right), \tag{4}$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a user-specified loss function, which measures the discrepancy between the decoded (*i.e.*, estimated) proximity values, $\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)$, and the true values, $s_{\mathcal{G}}(v_i, v_j)$.

Once we have optimized the encoder-decoder system, we can use the trained encoder to generate embeddings for nodes, which can then be used as a feature inputs for downstream machine learning tasks. For example, one could feed the learned embeddings to a logistic regression classifier to predict the community that a node belongs to [46], or one could use distances between the embeddings to recommend friendship links in a social network [3, 27] (Section 2.7 discusses further applications).

Adopting this encoder-decoder view, we organize our discussion of the various node embedding methods along the following four methodological components:

1. A **pairwise proximity function** $s_{\mathcal{G}} : \mathcal{V} \times \mathcal{V} \to \mathbb{R}^+$, defined over the graph, $\mathcal{G}$. This function measures how closely connected two nodes are in $\mathcal{G}$.

2. An **encoder function**, ENC, that generates the node embeddings. This function contains a number of trainable parameters that are optimized during the training phase.

3. A **decoder function**, DEC, which reconstructs pairwise proximity values from the generated embeddings. This function usually contains no trainable parameters.

4. A **loss function**, ℓ, which determines how the quality of the pairwise reconstructions is evaluated in order to train the model, *i.e.*, how $\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)$ is compared to the true $s_{\mathcal{G}}(v_i, v_j)$ values.

As we will show, the primary methodological distinctions between the various node embedding approaches are in how they define these four components.

2.1.1 Notes on optimization and implementation details

All of the methods we review involve optimizing the parameters of the encoder algorithm, Θ_{ENC}, by minimizing a loss analogous to Equation (4).[3] In most cases, stochastic gradient descent is used for optimization, though some algorithms do permit closed-form solutions via matrix decomposition (*e.g.*, [9]). However, note that we will not focus on optimization algorithms here and instead will emphasize high-level differences that exist across different embedding methods, independent of the specifics of the optimization approach.

[3]Occasionally, different methods will add additional auxiliary objectives or regularizers beyond the standard encoder-decoder objective, but we will often omit these details for brevity. A few methods also optimize parameters in the decoder, Θ_{DEC}.

Table 1: A summary of some well-known direct encoding embedding algorithms. Note that the decoders and proximity functions for the random-walk based methods are asymmetric, with the proximity function, $p_{\mathcal{G}}(v_j|v_i)$, corresponding to the probability of visiting v_j on a fixed-length random walk starting from v_i.

Type	Method	Decoder	Proximity measure	Loss function (ℓ)
Matrix factorization	Laplacian Eigenmaps [4]	$\|\mathbf{z}_i - \mathbf{z}_j\|_2^2$	general	$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \cdot s_{\mathcal{G}}(v_i, v_j)$
	Graph Factorization [1]	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}$	$\|\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j)\|_2^2$
	GraRep [9]	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}, \mathbf{A}_{i,j}^2, ..., \mathbf{A}_{i,j}^k$	$\|\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j)\|_2^2$
	HOPE [44]	$\mathbf{z}_i^\top \mathbf{z}_j$	general	$\|\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j)\|_2^2$
Random walk	DeepWalk [46]	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v_j\|v_i)$	$-s_{\mathcal{G}}(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$
	node2vec [27]	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v_j\|v_i)$ (biased)	$-s_{\mathcal{G}}(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$

2.2 Direct encoding approaches

The majority of node embedding algorithms rely on what we call *direct encoding*. For these direct encoding approaches, the encoder function—which maps nodes to vector embeddings—is simply an "embedding lookup":

$$\text{ENC}(v_i) = \mathbf{Z}\mathbf{v}_i, \tag{5}$$

where $\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ is a matrix containing the embedding vectors for all nodes and $\mathbf{v}_i \in \mathbb{I}_{\mathcal{V}}$ is a one-hot indicator vector indicating the column of $\mathbf{Z}$ corresponding to node v_i. The set of trainable parameters for direct encoding methods is simply $\Theta_{\text{ENC}} = \{\mathbf{Z}\}$, *i.e.* the embedding matrix $\mathbf{Z}$ is optimized directly.

These approaches are largely inspired by classic matrix factorization techniques for dimensionality reduction [4] and multi-dimensional scaling [36]. Indeed, many of these approaches were originally motivated as factorization algorithms, and we reinterpret them within the encoder-decoder framework here. Table 1 summarizes some well-known direct-encoding methods within the encoder-decoder framework. Table 1 highlights how these methods can be succinctly described according to (i) their decoder function, (ii) their graph-based proximity measure, and (iii) their loss function. The following two sections describe these methods in more detail, distinguishing between matrix factorization-based approaches (Section 2.2.1) and more recent approaches based on random walks (Section 2.2.2).

2.2.1 Factorization-based approaches

Early methods for learning representations for nodes largely focused on matrix-factorization approaches, which are directly inspired by classic techniques for dimensionality reduction [4, 36].
Laplacian eigenmaps. One of the earliest, and most well-known instances, is the Laplacian eigenmaps (LE) technique [4], which we can view within the encoder-decoder framework as a direct encoding approach in which the decoder is defined as

$$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) = \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$$

and where the loss function weights pairs of nodes according to their proximity in the graph:

$$\mathcal{L} = \sum_{(v_i, v_j) \in \mathcal{D}} \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \cdot s_{\mathcal{G}}(v_i, v_j). \tag{6}$$

Inner-product methods. Following on the Laplacian eigenmaps technique, there are a large number of recent embedding methodologies based on a pairwise, inner-product decoder:

$$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) = \mathbf{z}_i^\top \mathbf{z}_j, \tag{7}$$

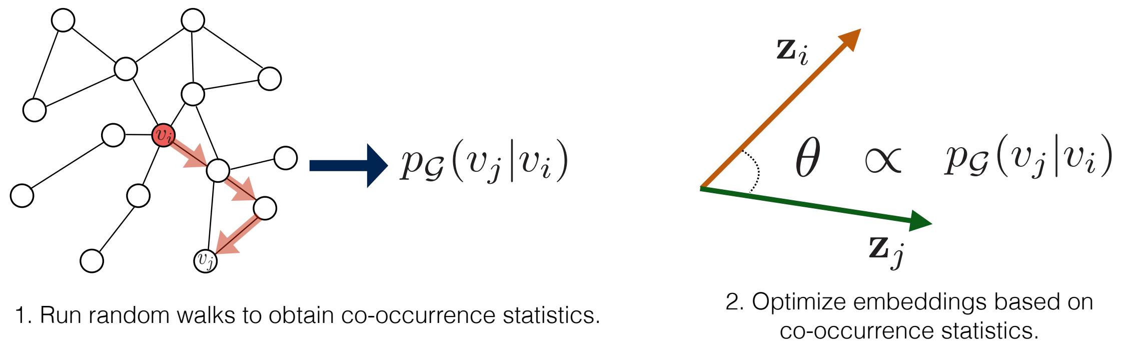

v_i
v_j
$p_{\mathcal{G}}(v_j|v_i)$
1. Run random walks to obtain co-occurrence statistics.
$\mathbf{z}_i$
$\theta \propto p_{\mathcal{G}}(v_j|v_i)$
$\mathbf{z}_j$
2. Optimize embeddings based on co-occurrence statistics.

Figure 4: The random-walk based methods sample a large number of fixed-length random walks starting from each node, v_i. The embedding vectors are then optimized so that the dot-product, or angle, between two embeddings, $\mathbf{z}_i$ and $\mathbf{z}_j$, is (roughly) proportional to the probability of visiting v_j on a fixed-length random walk starting from v_i.

where the strength of the relationship between two nodes is proportional to the dot product of their embeddings. The Graph Factorization (GF) algorithm[4] [1], GraRep [9], and HOPE [44] all fall firmly within this class. In particular, all three of these methods use an inner-product decoder, a mean-squared-error (MSE) loss,

$$\mathcal{L} = \sum_{(v_i,v_j)\in\mathcal{D}} \|\text{DEC}(\mathbf{z}_i,\mathbf{z}_j) - s_{\mathcal{G}}(v_i,v_j)\|_2^2, \tag{8}$$

and they differ primarily in the graph proximity measure used, *i.e.* how they define $s_{\mathcal{G}}(v_i, v_j)$. The Graph Factorization algorithm defines proximity directly based on the adjacency matrix (*i.e.*, $s_{\mathcal{G}}(v_i, v_j) \triangleq \mathbf{A}_{i,j}$) [1]; GraRep considers various powers of the adjacency matrix (*e.g.*, $s_{\mathcal{G}}(v_i, v_j) \triangleq \mathbf{A}^2_{i,j}$) in order to capture higher-order graph proximity [9]; and the HOPE algorithm supports general proximity measures (*e.g.*, based on Jaccard neighborhood overlaps) [44]. These various different proximity functions trade-off between modeling "first-order proximity", where $s_{\mathcal{G}}$ directly measures connections between nodes (*i.e.*, $s_{\mathcal{G}}(v_i, v_j) \triangleq \mathbf{A}_{i,j}$ [1]) and modeling "higher-order proximity", where $s_{\mathcal{G}}$ corresponds to more general notions of neighborhood overlap (*e.g.*, $s_{\mathcal{G}}(v_i, v_j) = \mathbf{A}^2_{i,j}$ [9]).

We refer to these methods in this section as matrix-factorization approaches because, averaging over all nodes, they optimize loss functions (roughly) of the form:

$$\mathcal{L} \approx \|\mathbf{Z}^\top\mathbf{Z} - \mathbf{S}\|_2^2, \tag{9}$$

where $\mathbf{S}$ is a matrix containing pairwise proximity measures (*i.e.*, $\mathbf{S}_{i,j} \triangleq s_{\mathcal{G}}(v_i, v_j)$) and $\mathbf{Z}$ is the matrix of node embeddings. Intuitively, the goal of these methods is simply to learn embeddings for each node such that the inner product between the learned embedding vectors approximates some deterministic measure of graph proximity.

2.2.2 Random walk approaches

Many recent successful methods that also belong to the class of *direct encoding* approaches learn the node embeddings based on random walk statistics. Their key innovation is optimizing the node embeddings so that nodes have similar embeddings if they tend to co-occur on short random walks over the graph (Figure 4). Thus, instead of using a deterministic measure of graph proximity, like the methods of Section 2.2.1, these random walk methods employ a flexible, stochastic measure of graph proximity, which has led to superior performance in a number of settings [26].

[4]Of course, Ahmed et al. [1] were not the first researchers to propose factorizing an adjacency matrix, but they were the first to present a scalable $O(|\mathcal{E}|)$ algorithm for the purpose of generating node embeddings.

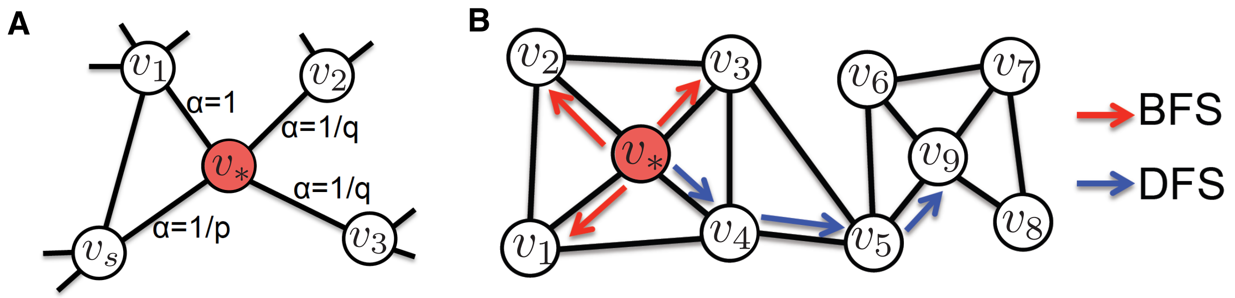


Figure 5: **A**, Illustration of how node2vec biases the random walk using the p and q parameters. Assuming that the walk just transitioned from v_s to v_*, the edge labels, α, are proportional to the probability of the walk taking that edge at next time-step. **B**, Difference between random-walks that are based on breadth-first search (BFS) and depth-first search (DFS). BFS-like random walks are mainly limited to exploring a node's immediate (*i.e.*, one-hop) neighborhood and are generally more effective for capturing structural roles. DFS-like walks explore further away from the node and are more effective for capturing community structures. Adapted from [27].

DeepWalk and node2vec. Like the matrix factorization approaches described above, DeepWalk and node2vec rely on direct encoding and use a decoder based on the inner product. However, instead of trying to decode a fixed deterministic distance measure, these approaches optimize embeddings to encode the statistics of random walks. The basic idea behind these approaches is to learn embeddings so that (roughly):

$$\begin{aligned}\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) &\triangleq \frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{v_k \in \mathcal{V}} e^{\mathbf{z}_i^\top \mathbf{z}_k}} \\ &\approx p_{\mathcal{G},T}(v_j|v_i),\end{aligned} \tag{10}$$

where $p_{\mathcal{G},T}(v_j|v_i)$ is the probability of visiting v_j on a length-T random walk starting at v_i, with T usually defined to be in the range $T \in \{2, ..., 10\}$. Note that unlike the proximity measures in Section 2.2.1, $p_{\mathcal{G},T}(v_j|v_i)$ is both stochastic and asymmetric.

More formally, these approaches attempt to minimize the following cross-entropy loss:

$$\mathcal{L} = \sum_{(v_i,v_j) \in \mathcal{D}} -\log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)), \tag{11}$$

where in this case the training set, $\mathcal{D}$, is generated by sampling random walks starting from each node (*i.e.*, where N pairs for each node, v_i, are sampled from the distribution $(v_i, v_j) \sim p_{\mathcal{G},T}(v_j|v_j)$). However, naively evaluating this loss is prohibitively expensive—in particular, $O(|\mathcal{D}||\mathcal{V}|)$—since evaluating the denominator of Equation (10) has time complexity $O(|\mathcal{V}|)$. Thus, DeepWalk and node2vec use different optimizations and approximations to compute the loss in Equation (11). DeepWalk employs a "hierarchical softmax" technique to compute the normalizing factor, using a binary-tree structure to accelerate the computation [46]. In contrast, node2vec approximates Equation (11) using "negative sampling": instead of normalizing over the full vertex set, node2vec approximates the normalizing factor using a set of random "negative samples" [27].

Beyond these algorithmic differences, the key distinction between node2vec and DeepWalk is that node2vec allows for a flexible definition of random walks, whereas DeepWalk uses simple unbiased random walks over the graph. In particular, node2vec introduces two random walk hyperparameters, p and q, that bias the random walk (Figure 5.A). The hyperparameter p controls the likelihood of the walk immediately revisiting a node, while q controls the likelihood of the walk revisiting a node's one-hop neighborhood. By introducing these hyperparameters, node2vec is able to smoothly interpolate between walks that are more akin to breadth-first or depth-first search (Figure 5.B). Grover et al. found that tuning these parameters allowed the model to trade off between learning embeddings that emphasize community structures or embeddings that emphasize local structural roles [27] (see also Figure 1).

Large-scale information network embeddings (LINE). Another highly successful direct encoding approach, which is not based random walks but is contemporaneous and often compared with DeepWalk and node2vec, is the LINE method [53]. LINE combines two encoder-decoder objectives that optimize "first-order" and "second-order" graph proximity, respectively. The first-order objective uses a decoder based on the sigmoid function,

$$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) = \frac{1}{1 + e^{-\mathbf{z}_i^\top \mathbf{z}_j}}, \tag{12}$$

and an adjacency-based proximity measure (*i.e.*, $s_{\mathcal{G}}(v_i, v_j) = \mathbf{A}_{i,j}$). The second-order encoder-decoder objective is similar but considers two-hop adjacency neighborhoods and uses an encoder identical to Equation (10). Both the first-order and second-order objectives are optimized using loss functions derived from the KL-divergence metric [53]. Thus, LINE is conceptually related to node2vec and DeepWalk in that it uses a probabilistic decoder and loss, but it explicitly factorizes first- and second-order proximities, instead of combining them in fixed-length random walks.

HARP: Extending random-walk embeddings via graph pre-processing. Recently, Chen et al. [13] introduced a "meta-strategy", called HARP, for improving various random-walk approaches via a graph pre-processing step. In this approach, a graph coarsening procedure is used to collapse related nodes in $\mathcal{G}$ together into "supernodes", and then DeepWalk, node2vec, or LINE is run on this coarsened graph. After embedding the coarsened version of $\mathcal{G}$, the learned embedding of each supernode is used as an initial value for the random walk embeddings of the supernode's constituent nodes (in another round of non-convex optimization on a "finer-grained" version of the graph). This general process can be repeated in a hierarchical manner at varying levels of coarseness, and has been shown to consistently improve performance of DeepWalk, node2vec, and LINE [13].

Additional variants of the random-walk idea. There have also been a number of further extensions of the random walk idea. For example, Perozzi et al. [47] extend the DeepWalk algorithm to learn embeddings using random walks that "skip" or "hop" over multiple nodes at each step, resulting in a proximity measure similar to GraRep [9], while Chamberlan et al. [11] modify the inner-product decoder of node2vec to use a hyperbolic, rather than Euclidean, distance measure.

2.3 Generalized encoder-decoder architectures

So far all of the node embedding methods we have reviewed have been direct encoding methods, where the encoder is a simply an embedding lookup (Equation 5). However, these direct encoding approaches train unique embedding vectors for each node independently, which leads to a number of drawbacks:

1. No parameters are shared between nodes in the encoder (*i.e.*, the encoder is simply an embedding lookup based on arbitrary node ids). This can be statistically inefficient, since parameter sharing can act as a powerful form of regularization, and it is also computationally inefficient, since it means that the number of parameters in direct encoding methods necessarily grows as $O(|\mathcal{V}|)$.

2. Direct encoding also fails to leverage node attributes during encoding. In many large graphs nodes have attribute information (*e.g.*, user profiles on a social network) that is often highly informative with respect to the node's position and role in the graph.

3. Direct encoding methods are inherently *transductive* [28], *i.e.*, they can only generate embeddings for nodes that were present during the training phase, and they cannot generate embeddings for previously unseen nodes unless additional rounds of optimization are performed to optimize the embeddings for these nodes. This is highly problematic for evolving graphs, massive graphs that cannot be fully stored in memory, or domains that require generalizing to new graphs after training.

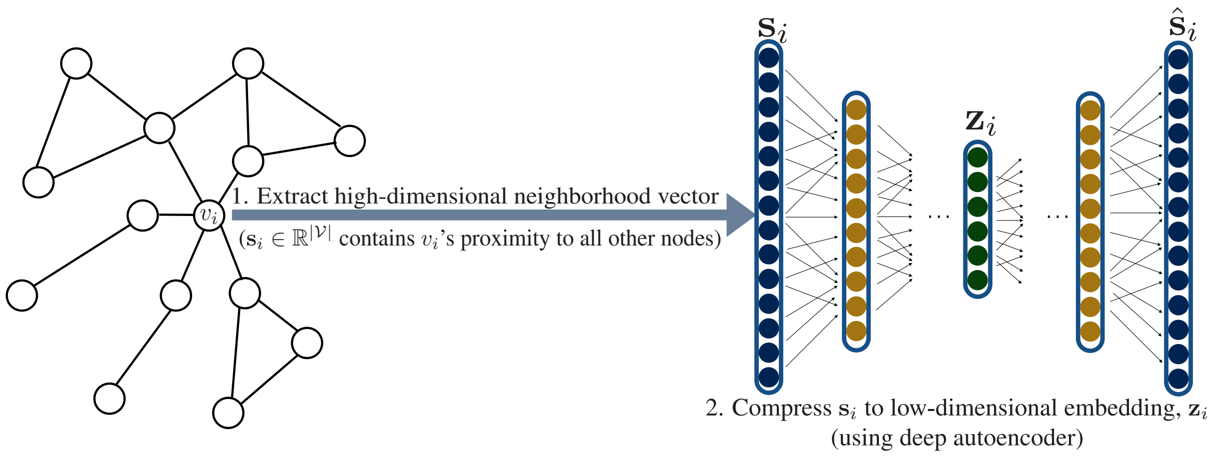

v_i
1. Extract high-dimensional neighborhood vector
($\mathbf{s}_i \in \mathbb{R}^{|\mathcal{V}|}$ contains v_i's proximity to all other nodes)
$\mathbf{s}_i$
$\mathbf{z}_i$
$\hat{\mathbf{s}}_i$
2. Compress $\mathbf{s}_i$ to low-dimensional embedding, $\mathbf{z}_i$
(using deep autoencoder)

Figure 6: To generate an embedding for a node, v_i, the neighborhood autoencoder approaches first extract a high-dimensional neighborhood vector $\mathbf{s}_i \in \mathbb{R}^{|\mathcal{V}|}$, which summarizes v_i's proximity to all other nodes in the graph. The $\mathbf{s}_i$ vector is then fed through a deep autoencoder to reduce its dimensionality, producing the low-dimensional $\mathbf{z}_i$ embedding.

Recently, a number of approaches have been proposed to address some, or all, of these issues. These approaches still fall firmly within the encoder-decoder framework outlined in Section 2.1, but they differ from the direct encoding methods of Section 2.2 in that they use a more complex encoders, which depend more generally on the structure and attributes of the graph.

2.3.1 Neighborhood autoencoder methods

Deep Neural Graph Representations (DNGR) [10] and Structural Deep Network Embeddings (SDNE) [58] address the first problem outlined above: unlike the direct encoding methods, they directly incorporate graph structure into the encoder algorithm. The basic idea behind these approaches is that they use autoencoders—a well known approach for deep learning [30]—in order to compress information about a node's local neighborhood (Figure 6). DNGR and SDNE also differ from the previously reviewed approaches in that they use a *unary decoder* instead of a pairwise one.

In these approaches, each node, v_i, is associated with a neighborhood vector, $\mathbf{s}_i \in \mathbb{R}^{|\mathcal{V}|}$, which corresponds to v_i's row in the matrix $\mathbf{S}$ (recall that $\mathbf{S}$ contains pairwise node proximities, *i.e.*, $\mathbf{S}_{i,j} = s_{\mathcal{G}}(v_i, v_j)$). The $\mathbf{s}_i$ vector contains v_i's pairwise graph proximity with all other nodes and functions as a high-dimensional vector representation of v_i's neighborhood. The autoencoder objective for DNGR and SDNE is to embed nodes using the $\mathbf{s}_i$ vectors such that the $\mathbf{s}_i$ vectors can then be reconstructed from these embeddings:

$$\text{DEC}(\text{ENC}(\mathbf{s}_i)) = \text{DEC}(\mathbf{z}_i) \approx \mathbf{s}_i. \tag{13}$$

In other words, the loss for these methods takes the following form:

$$\mathcal{L} = \sum_{v_i \in \mathcal{V}} \|\text{DEC}(\mathbf{z}_i) - \mathbf{s}_i\|_2^2. \tag{14}$$

As with the pairwise decoder, we have that the dimension of the $\mathbf{z}_i$ embeddings is much smaller than $|\mathcal{V}|$ (the dimension of the $\mathbf{s}_i$ vectors), so the goal is to compress the node's neighborhood information into a low-dimensional vector. For both SDNE and DNGR, the encoder and decoder functions consist of multiple stacked neural network layers: each layer of the encoder reduces the dimensionality of its input, and each layer of the decoder increases the dimensionality of its input (Figure 6; see [30] for an overview of deep autoencoders).

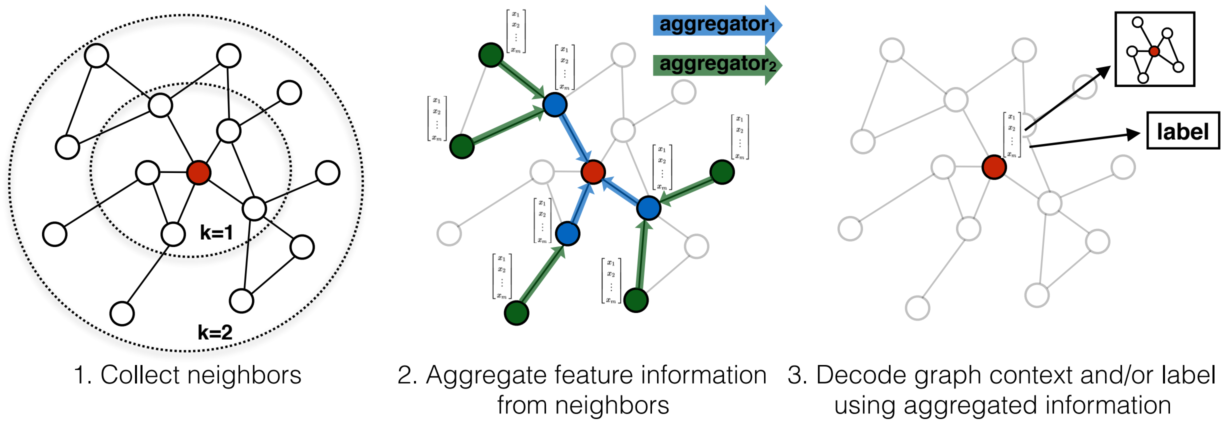

k=1
k=2
aggregator₁
aggregator₂
label
1. Collect neighbors
2. Aggregate feature information from neighbors
3. Decode graph context and/or label using aggregated information

Figure 7: Overview of the neighborhood aggregation methods. To generate an embedding for a node, these methods first collect the node's k-hop neighborhood (occasionally sub-sampling the full neighborhood for efficiency). In the next step, these methods aggregate the attributes of node's neighbors, using neural network aggregators. This aggregated neighborhood information is used to generate an embedding, which is then fed to the decoder. Adapted from [28].

SDNE and DNGR differ in the similarity functions they use to construct the neighborhood vectors $\mathbf{s}_i$ and also in the exact details of how the autoencoder is optimized. DNGR defines $\mathbf{s}_i$ according to the pointwise mutual information of two nodes co-occurring on random walks, similar to DeepWalk and node2vec. SDNE simply sets $\mathbf{s}_i \triangleq \mathbf{A}_i$, *i.e.*, equal to v_i's adjacency vector. SDNE also combines the autoencoder objective (Equation 13) with the Laplacian eigenmaps objective (Equation 6) [58].

Note that the encoder in Equation (13) depends on the input $\mathbf{s}_i$ vector, which contains information about v_i's local graph neighborhood. This dependency allows SDNE and DNGR to incorporate structural information about a node's local neighborhood directly into the encoder as a form of regularization, which is not possible for the direct encoding approaches (since their encoder depends only on the node id). However, despite this improvement, the autoencoder approaches still suffer from some serious limitations. Most prominently, the input dimension to the autoencoder is fixed at $|\mathcal{V}|$, which can be extremely costly and even intractable for graphs with millions of nodes. In addition, the structure and size of the autoencoder is fixed, so SDNE and DNGR are strictly transductive and cannot cope with evolving graphs, nor can they generalize across graphs.

2.3.2 Neighborhood aggregation and convolutional encoders

A number of recent node embedding approaches aim to solve the main limitations of the direct encoding and autoencoder methods by designing encoders that rely on a node's local neighborhood, but not necessarily the entire graph. The intuition behind these approaches is that they generate embeddings for a node by aggregating information from its local neighborhood (Figure 7).

Unlike the previously discussed methods, these *neighborhood aggregation* algorithms rely on node features or attributes (denoted $\mathbf{x}_i \in \mathbb{R}^m$) to generate embeddings. For example, a social network might have text data (*e.g.*, profile information), or a protein-protein interaction network might have molecular markers associated with each node. The neighborhood aggregation methods leverage this attribute information to inform their embeddings. In cases where attribute data is not given, these methods can use simple graph statistics as attributes (*e.g.*, node degrees) [28], or assign each node a one-hot indicator vector as an attribute [35, 52]. These methods are often called *convolutional* because they represent a node as a function of its surrounding neighborhood, in a manner similar to the receptive field of a center-surround convolutional kernel in computer vision [34].[5]

[5]These methods also have theoretical connections to approximate spectral kernels on graphs [18]; see [34] for a further discussion.

Algorithm 1: Neighborhood-aggregation encoder algorithm. Adapted from [28].

```
Input  : Graph $\mathcal{G}(\mathcal{V},\mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth $K$; weight matrices $\{\mathbf{W}^k, \forall k \in [1,K]\}$;
         non-linearity $\sigma$; differentiable aggregator functions $\{\text{AGGREGATE}_k, \forall k \in [1,K]\}$;
         neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$
Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$
1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;
2 for $k = 1...K$ do
3     for $v \in \mathcal{V}$ do
4         $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
5         $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{COMBINE}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6     end
7     $\mathbf{h}_v^k \leftarrow \text{NORMALIZE}(\mathbf{h}_v^k), \forall v \in \mathcal{V}$
8 end
9 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$
```

In the encoding phase, the neighborhood aggregation methods build up the representation for a node in an iterative, or recursive, fashion (see Algorithm 1 for pseudocode). First, the node embeddings are initialized to be equal to the input node attributes. Then at each iteration of the encoder algorithm, nodes aggregate the embeddings of their neighbors, using an aggregation function that operates over sets of vectors. After this aggregation, every node is assigned a new embedding, equal to its aggregated neighborhood vector combined with its previous embedding from the last iteration. Finally, this combined embedding is fed through a dense neural network layer and the process repeats. As the process iterates, the node embeddings contain information aggregated from further and further reaches of the graph. However, the dimensionality of the embeddings remains constrained as the process iterates, so the encoder is forced to compress all the neighborhood information into a low dimensional vector. After K iterations the process terminates and the final embedding vectors are output as the node representations.

There are a number of recent approaches that follow the basic procedure outlined in Algorithm 1, including graph convolutional networks (GCN) [34, 35, 52, 55], column networks [49], and the GraphSAGE algorithm [28]. The trainable parameters in Algorithm 1—a set of aggregation functions and a set weight matrices $\{\mathbf{W}^k, \forall k \in [1,K]\}$—specify how to aggregate information from a node's local neighborhood and, unlike the direct encoding approaches (Section 2.2), these parameters are shared across nodes. The same aggregation function and weight matrices are used to generate embeddings for all nodes, and only the input node attributes and neighborhood structure change depending on which node is being embedded. This parameter sharing increases efficiency (*i.e.*, the parameter dimensions are independent of the size of the graph), provides regularization, and allows this approach to be used to generate embeddings for nodes that were not observed during training [28].

GraphSAGE, column networks, and the various GCN approaches all follow Algorithm 1 but differ primarily in how the aggregation (line 4) and vector combination (line 5) are performed. GraphSAGE uses concatenation in line 5 and permits general aggregation functions; the authors experiment with using the element-wise mean, a max-pooling neural network and LSTMs [31] as aggregators, and they found the the more complex aggregators, especially the max-pooling neural network, gave significant gains. GCNs and column networks use a weighted sum in line 5 and a (weighted) element-wise mean in line 4.

Column networks also add an additional "interpolation" term before line 7, setting

$$\mathbf{h}_v^{k'} = \alpha \mathbf{h}_v^k + (1-\alpha)\mathbf{h}_v^{k-1}, \tag{15}$$

where α is an interpolation weight computed as a non-linear function of $\mathbf{h}_v^{k-1}$ and $\mathbf{h}_{\mathcal{N}(v)}^{k-1}$. This interpolation term allows the model to retain local information as the process iterates (*i.e.*, as k increases and the model integrates information from further reaches of the graph).

In principle, the GraphSAGE, column network, and GCN encoders can be combined with any of the previously discussed decoders and loss functions, and the entire system can be optimized using SGD. For example, Hamilton et al. [28] use an identical decoder and loss as node2vec, while Kipf et al. [35] use a decoder and loss function similar to the Graph Factorization approach.

Neighborhood aggregation encoders following Algorithm 1 have been found to provide consistent gains compared to their direct encoding counterparts, on both node classification [28, 34] and link prediction [55, 35, 52] benchmarks. At a high level, these approaches solve the four main limitations of direct encoding, noted at the beginning of Section 2.3: they incorporate graph structure into the encoder; they leverage node attributes; their parameter dimension can be made sub-linear in $|\mathcal{V}|$; and they can generate embeddings for nodes that were not present during training.

2.4 Incorporating task-specific supervision

The basic encoder-decoder framework described thus far is by default unsupervised, *i.e.*, the model is optimized, or trained, over set of node pairs to reconstruct pairwise proximity values, $s_{\mathcal{G}}(v_i, v_j)$, which depend only on the graph, $\mathcal{G}$. However, many node embedding algorithms—especially the neighborhood aggregation approaches presented in Section 2.3.2—can also incorporate task-specific supervision [28, 34, 52, 59]. In particular, it is common for methods incorporate supervision from node classification tasks in order to learn the embeddings.[6] For simplicity, we discuss the case where nodes have an associated binary classification label, but the approach we describe is easily extended to more complex classification settings.

Assume that we have a binary classification label, $y_i \in \mathbb{Z}$, associated with each node. To learn to map nodes to their labels, we can feed our embedding vectors, $\mathbf{z}_i$, through a logistic, or sigmoid, function $\hat{y}_i = \sigma(\mathbf{z}_i^\top \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is a trainable parameter vector. We can then compute the cross-entropy loss between these predicted class probabilities and the true labels:

$$\mathcal{L} = \sum_{v_i \in \mathcal{V}} y_i \log(\sigma(\text{ENC}(v_i)^\top \boldsymbol{\theta})) + (1 - y_i) \log(1 - \sigma(\text{ENC}(v_i)^\top \boldsymbol{\theta})). \tag{16}$$

The gradient computed according to Equation (16) can then be backpropagated through the encoder to optimize its parameters. This task-specific supervision can completely replace the reconstruction loss computed using the decoder (*i.e.*, Equation 3) [28, 34], or it can be included along with the decoder loss [59].

2.5 Extensions to multi-modal graphs

While we have focused on simple, undirected graphs, many real-world graphs have complex multi-modal, or multi-layer, structures (*e.g.*, heterogeneous node and edge types), and a number of works have introduced strategies to cope with this heterogeneity.

2.5.1 Dealing with different node and edge types

Many graphs contain different types of nodes and edges. For example, recommender system graphs consist of two distinct layers—users and content—while many biological networks have a variety of layers, with distinct interactions between them (*e.g.*, diseases, genes, and drugs).

A general strategy for dealing with this issue is to (i) use different encoders for nodes of different types [12] and (ii) extend pairwise decoders with type-specific parameters [42, 52]. For example, in graphs with varying

[6] The unsupervised pairwise decoder is already naturally aligned with the link prediction task.

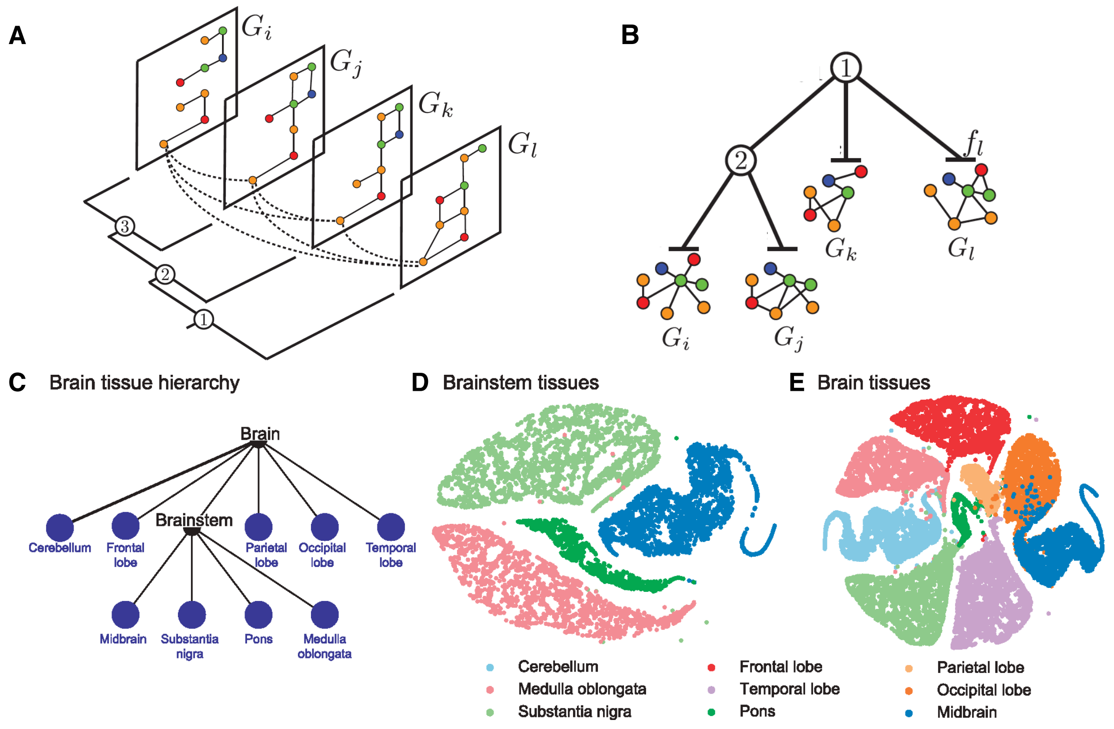


Figure 8: **A**, Example of a 4-layer graph, where the same nodes occur in multiple different layers. This multi-layer structure can be exploited to regularize learning at the different layers by requiring that the embeddings for the same node in different layers are similar to each other. **B**, Multi-layer graphs can exhibit hierarchical structure, where non-root layers in the hierarchy contain the union of the edges present in their child layers—*e.g.*, a biological interaction graph derived from the entire human brain contains the union of the interactions in the frontal and temporal lobes. This structure can be exploited by learning embeddings at various levels of the hierarchy, and only applying the regularization between layers that are in a parent-child relationship. **C-E**, Example application of multi-layer graph embedding to protein-protein interaction graphs derived from different brain tissues; **C** shows the hierarchy between the different tissue regions, while **D** and **E** visualize the protein embeddings generated at the brainstem and whole-brain layers. The embeddings were generated using the multi-layer OhmNet method and projected to two dimensions using t-SNE. Adapted from [60].

edge types, the standard inner-product edge decoder (*i.e.*, $\mathbf{z}_i^\top \mathbf{z}_j \approx \mathbf{A}_{i,j}$) can be replaced with a bilinear form [12, 42, 52]:

$$\text{DEC}_\tau(\mathbf{z}_i, \mathbf{z}_j) = \mathbf{z}^\top \mathbf{A}_\tau \mathbf{z}, \tag{17}$$

where τ indexes a particular edge type and $\mathbf{A}_\tau$ is a learned parameter specific to edges of type τ. The matrix, $\mathbf{A}_\tau$, in Equation (17) can be regularized in various ways (*e.g.*, constrained to be diagonal) [52], which can be especially useful when there are a large number of edge types, as in the case for embedding knowledge graphs. Indeed, the literature on knowledge-graph completion—where the goal is predict missing relations in knowledge graphs—contains many related techniques for decoding a large number of edge types (*i.e.*, relations) [42].[7]

Recently, Dong et al. [19] also proposed a strategy for sampling random walks from heterogeneous graphs, where the random walks are restricted to only transition between particular types of nodes. This approach allows many of the methods in Section 2.2.2 to be applied on heterogeneous graphs and is complementary to the idea of including type-specific encoders and decoders.

2.5.2 Tying node embeddings across layers

In some cases graphs have multiple “layers” that contain copies of the same nodes (Figure 8.A). For example, in protein-protein interaction networks derived from different tissues (*e.g.*, brain or liver tissue), some proteins occur across multiple tissues. In these cases it can be beneficial to share information across layers, so that a node’s embedding in one layer can be informed by its embedding in other layers. Zitnik et al. [60] offer one solution to this problem, called OhmNet, that combines node2vec with a regularization penalty that ties the embeddings across layers. In particular, assuming that we have a node v_i, which belongs to two distinct layers $\mathcal{G}_1$ and $\mathcal{G}_2$, we can augment the standard embedding loss on this node as follows:

$$\mathcal{L}(v_i)' = \mathcal{L}(v_i) + \lambda \|\mathbf{z}_i^{\mathcal{G}_1} - \mathbf{z}_i^{\mathcal{G}_2}\| \tag{18}$$

where $\mathcal{L}$ denotes the usual embedding loss for that node (*e.g.*, from Equation 8 or 11), λ denotes the regularization strength, and $\mathbf{z}_i^{\mathcal{G}_1}$ and $\mathbf{z}_i^{\mathcal{G}_2}$ denote v_i’s embeddings in the two different layers, respectively.

Zitnik et al. further extend this idea by exploiting hierarchies between graph layers (Figure 8.B). For example, in protein-protein interaction graphs derived from various tissues, some layers correspond to interactions throughout large regions (*e.g.*, interactions that occur in any brain tissue) while other interaction graphs are more fine-grained (*e.g.*, only interactions that occur in the frontal lobe). To exploit this structure, embeddings can be learned at the various levels of the hierarchy, and the regularization in Equation (18) can recursively applied between layers that have a parent-child relationship in the hierarchy.

2.6 Embedding structural roles

So far, all the approaches we have reviewed optimize node embeddings so that nearby nodes in the graph have similar embeddings. However, in many tasks it is more important to learn representations that correspond to the structural roles of the nodes, independent of their global graph positions (*e.g.*, in communication or transportation networks) [29]. The node2vec approach introduced in Section 2.2.2 offers one solution to this problem, as Grover et al. found that biasing the random walks allows their model to better capture structural roles (Figure 5). However, more recently, Ribeiro et al. [50] and Donnat et al. [20] have developed node embedding approaches that are specifically designed to capture structural roles.

Ribeiro et al. propose struc2vec, which involves generating a a series of weighted auxiliary graphs $\mathcal{G}'_k, k = \{1, 2, ...\}$ from the original graph $\mathcal{G}$, where the auxiliary graph $\mathcal{G}'_k$ captures structural similarities between nodes’ k-hop neighborhoods. In particular, letting $R_k(v_i)$ denote the ordered sequence of degrees of the nodes that are exactly k-hops away from v_i, the edge-weights, $w_k(v_i, v_j)$, in auxiliary graph G'_k are recursively defined as

$$w_k(v_i, v_j) = w_{k-1}(v_i, v_j) + d(R_k(v_i), R_k(v_j)), \tag{19}$$

where $w_0(v_i, v_j) = 0$ and $d(R_k(v_i), R_k(v_j))$ measures the “distance” between the ordered degree sequences $R_k(v_i)$ and $R_k(v_j)$ (*e.g.*, computed via dynamic time warping [50]). After computing these weighted auxillary graphs, struc2vec runs biased random walks over them and uses these walks as input to the node2vec optimization algorithm.

Donnat et al. take a very different approach to capturing structural roles, called GraphWave, which relies on spectral graph wavelets and heat kernels [20]. In brief, we let $\mathbf{L}$ denote the graph Laplacian—*i.e.*, $\mathbf{L} = \mathbf{D} - \mathbf{A}$ where $\mathbf{D}$ contains node degrees on the diagonal and $\mathbf{A}$ is the adjacency matrix—and we let $\mathbf{U}$ and $\lambda_i, i = 1...|\mathcal{V}|$ denote the eigenvector matrix and eigenvalues of $\mathbf{L}$, respectively. Finally, we assume that we have a heat kernel, $g(\lambda) = e^{-s\lambda}$, with pre-defined scale s. Using $\mathbf{U}$ and $g(\lambda)$, GraphWave computes a vector, $\boldsymbol{\psi}_{v_i}$, corresponding to the structural role of node, $v_i \in \mathcal{V}$, as

$$\boldsymbol{\psi}_{v_i} = \mathbf{U}\mathbf{G}\mathbf{U}^\top \mathbf{v}_i \tag{20}$$

[7]We do not review this literature in detail here, and refer the reader to Nickel et al. [42] for a recent review.

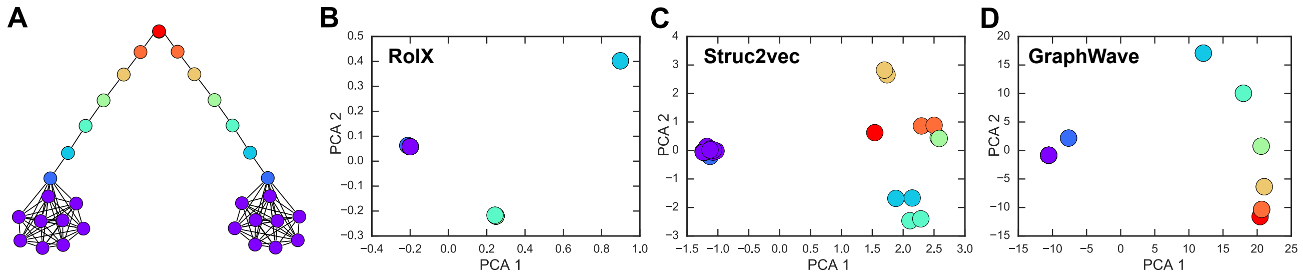


Figure 9: **A**, Synthetic barbell graph used as a test dataset for detecting structural roles, where nodes are colored according to their structural roles. In this case, the structural roles (*i.e.*, colors) are computed by examining the degrees of each node's immediate neighbors, and their 2-hop neighbors, and so on (up to $|\mathcal{V}|$-hop neighborhoods). **B-D**, Visualization of the output of three role-detection algorithms on the barbell graph, where the model outputs are projected using principal components analysis. RolX (**B**) [29] is a baseline approach based upon hand-designed features, while struc2vec (**C**) and GraphWave (**D**) use different representation learning approaches. Note that all methods correctly differentiate the ends of the barbells from the rest of the graph, but only GraphWave is able to correctly differentiate all the various roles. Note also that there are fewer visible nodes in part **D** compared to **A** because GraphWave maps identically colored (*i.e.*, structurally equivalent) nodes to the exact same position in the embedding space. Reprinted from [20].

where $\mathbf{G} = \text{diag}([g(\lambda_1), ..., g(\lambda_{|\mathcal{V}|})])$ and $\mathbf{v}_i$ is a one-hot indicator vector corresponding to v_i's row/column in the Laplacian.[8] Donnat et al. show that these $\boldsymbol{\psi}_{v_i}$ vectors implicitly relate to topological quantities, such as v_i's degree and the number of k-cycles v_i is involved in. They find that—with a proper choice of scale, s—WaveGraph is able to effectively capture structural information about a nodes role in a graph (Figure 9).

2.7 Applications of node embeddings

The most common use cases for node embeddings are for visualization, clustering, node classification, and link prediction, and each of these use cases is relevant to a number of application domains, ranging from computational social science to computational biology.

Visualization and pattern discovery. The problem of visualizing graphs in a 2D interface has a long history, with applications throughout data mining, the social sciences, and biology [17]. Node embeddings offer a powerful new paradigm for graph visualization: because nodes are mapped to real-valued vectors, researchers can easily leverage existing, generic techniques for visualization high-dimensional datasets [56, 54]. For example, node embeddings can be combined with well-known techniques such as t-SNE [56] or principal components analysis (PCA) in order to generate 2D visualizations of graphs [46, 53], which can be useful for discovering communities and other hidden structures (Figures 2 and 8).

Clustering and community detection. In a similar vein as visualization, node embeddings are a powerful tool for clustering related nodes, a task that has countless applications from computational biology (*e.g.*, discovering related drugs) to marketing (*e.g.*, discovering related products) [23]. Again, because each node is associated with real-valued vector embedding, it is possible to apply any generic clustering algorithm to the set of learned node embeddings (*e.g.*, k-means or DB-scan [22]). This offers an open-ended and powerful alternative to traditional community detection techniques, and it also opens up new methodological opportunities, since node embeddings can capture the functional or structural roles played by different nodes, rather than just community structure.

Node classification and semi-supervised learning. Node classification is perhaps the most common benchmark task used for evaluating node embeddings. In most cases, the node classification task is a form of semi-supervised learning, where labels are only available for a small proportion of nodes, with the goal being to label the full graph based only on this small initial seed set. Common applications of semi-supervised node classification include classifying proteins according to their biological function [27] and classifying documents,

[8]Note that Equation (20) can be efficiently approximated via Chebyshev polynomials [20].

videos, web pages, or individuals into different categories/communities [27, 34, 46, 53]. Recently, Hamilton et al. [28] introduced the task of inductive node classification, where the goal is to classify nodes that were not seen during training, *e.g.* classifying new documents in evolving information graphs or generalizing to unseen protein-protein interaction networks.

Link prediction. Node embeddings are also extremely useful as features for link prediction, where the goal is to predict missing edges, or edges that are likely to form in the future [3]. Link prediction is at the core of recommender systems and common applications of node embeddings reflect this deep connection, including predicting missing friendship links in social networks [53] and affinities between users and movies [55]. Link prediction also has important applications in computational biology. Many biological interaction graphs (*e.g.*, between proteins and other proteins, or drugs and diseases) are incomplete, since they rely on data obtained from costly lab experiments. Predicting links in these noisy graphs is an important method for automatically expanding biological datasets and for recommending new directions for wet-lab experimentation [40]. More generally, link prediction is closely related to statistical relational learning [24], where a common task is to predict missing relations between entities in a knowledge graph [42].

3 Embedding subgraphs

We now turn to the task of representation learning on (sub)graphs, where the goal is to encode a set of nodes and edges into a low-dimensional vector embedding. More formally, the goal is to learn a continuous vector representation, $\mathbf{z}_{\mathcal{S}} \in \mathbb{R}^d$, of an induced subgraph $\mathcal{G}[\mathcal{S}]$ of the full graph $\mathcal{G}$, where $\mathcal{S} \subseteq \mathcal{V}$. (Note that these methods can embed both subgraphs ($\mathcal{S} \subset \mathcal{V}$) as well as entire graphs ($\mathcal{S} = \mathcal{V}$).) The embedding, $\mathbf{z}_{\mathcal{S}}$, can then be used to make predictions about the entire subgraph; for example, one might embed graphs corresponding to different molecules to predict their therapeutic properties [21].

Representation learning on subgraphs is closely related to the design of graph kernels, which define a distance measure between subgraphs [57]. That said, we omit a detailed discussion of graph kernels, which is a large and rich research area of its own, and refer the reader to [57] for a detailed discussion. The methods we review differ from the traditional graph kernel literature primarily in that we seek to learn useful representations from data, rather than pre-specifying feature representations through a kernel function.

Many of the methods in this section build upon the techniques used to embed individual nodes, introduced in Section 2. However, unlike the node embedding setting, most subgraph embedding approaches are fully-supervised, being used for subgraph classification, where the goal is to predict a label associated with a particular subgraph. Thus, in this section we will focus on the various different approaches for generating the $\mathbf{z}_{\mathcal{S}}$ embeddings, with the assumption that these embeddings are being fed through a cross-entropy loss function, analogous to Equation (16).

3.1 Sets of node embeddings and convolutional approaches

There are several subgraph embedding techniques that can be viewed as direct extensions of the convolutional node embedding algorithms (described in Section 2.3.2). The basic intuition behind these approaches is that they equate subgraphs with sets of node embeddings. They use the convolutional neighborhood aggregation idea (*i.e.*, Algorithm 1) to generate embeddings for nodes and then use additional modules to aggregate sets of node embeddings corresponding to subgraphs. The primary distinction between the different approaches in this section is how they aggregate the set of node embeddings corresponding to a subgraph.

3.1.1 Sum-based approaches

For example, "convolutional molecular fingerprints" introduced by Duvenaud et al. [21] represent subgraphs in molecular graph representations by summing all the individual node embeddings in the subgraph:

$$\mathbf{z}_{\mathcal{S}} = \sum_{v_i \in \mathcal{S}} \mathbf{z}_i, \tag{21}$$

where the embeddings, $\{\mathbf{z}_i, \forall v_i \in \mathcal{S}\}$, are generated using Algorithm 1.

Dai et al. [16] employ an analogous sum-based approach but note that it has conceptual connections to mean-field inference: if the nodes in the graph are viewed as latent variables in a graphical model, then Algorithm 1 can be viewed as a form of mean-field inference where the message-passing operations have been replaced with differentiable neural network alternatives. Motivated by this connection, Dai et al. [16] also propose a modified encoder based on Loopy Belief Propagation [41]. Using the placeholders and notation from Algorithm 1, the basic idea behind this alternative is to construct intermediate embeddings, $\boldsymbol{\eta}_{i,j}$, corresponding to edges, $(i, j) \in \mathcal{E}$:

$$\boldsymbol{\eta}^k_{i,j} = \sigma(\mathbf{W}^k_{\mathcal{E}} \cdot \text{COMBINE}(\mathbf{x}_i, \text{AGGREGATE}(\boldsymbol{\eta}^{k-1}_{l,i}, \forall v_l \in \mathcal{N}(v_i) \setminus v_j\}))). \tag{22}$$

These edge embeddings are then aggregated to form the node embeddings:

$$\mathbf{z}^i = \sigma(\mathbf{W}^k_{\mathcal{V}} \cdot \text{COMBINE}(\mathbf{x}_i, \text{AGGREGATE}(\{\boldsymbol{\eta}^K_{i,l}, \forall v_l \in \mathcal{N}(v_i)\}))). \tag{23}$$

Once the embeddings are computed, Dai et al. [16], use a simple element-wise sum to combine the node embeddings for a subgraph, as in Equation (21).

3.1.2 Graph-coarsening approaches

Defferrard et al. [18] and Bruna et al. [8] also employ convolutional approaches, but instead of summing the node embeddings for the whole graph, they stack convolutional and "graph coarsening" layers (similar to the HARP approach in Section 2.2.2). In the graph coarsening layers, nodes are clustered together (using any graph clustering approach), and the clustered node embeddings are combined using element-wise max-pooling. After clustering, the new coarser graph is again fed through a convolutional encoder and the process repeats.

Unlike the convolutional approaches discussed in 2.3.2, Defferrard et al. [18] and Bruna et al. [8] also place considerable emphasis on designing convolutional encoders based upon the graph Fourier transform [15]. However, because the graph Fourier transform requires identifying and manipulating the eigenvectors of the graph Laplacian, naive versions of these approaches are necessarily $O(|\mathcal{V}|^3)$. State-of-the-art approximations to these spectral approaches (e.g., using Chebyshev polynomials) are conceptually similar to Algorithm 1, with some minor variations, and we refer the reader to Bronstein et al. [7] for a thorough discussion of these techniques.

3.1.3 Further variations

Other variants of the convolutional idea are proposed by Neipert et al. [43] and Kearnes et al. [33]. Both advocate alternative methods for aggregating sets of node embeddings corresponding to subgraphs: Kearnes et al. aggregate sets of nodes using "fuzzy" histograms instead of a sum, and they also employ edge embedding layers similar to [16]. Neipart et al. define an ordering on the nodes—*e.g.* using a problem specific ordering or by employing an off-the-shelf vertex coloring algorithm—and using this ordering, they concatenate the embeddings for all nodes and feed this concatenated vector through a standard convolutional neural network architecture.

3.2 Graph neural networks

In addition to the convolution-inspired subgraph embedding approaches discussed above, there is a related—and chronologically prior—line of work on "graph neural networks" (GNNs) [51]. Conceptually, the GNN idea is closely related to Algorithm 1. However, instead of aggregating information from neighbors, the intuition behind GNNs is that subgraphs can be viewed as specifying a "compute graph", *i.e.*, a recipe for accumulating and passing information between nodes.

In the original GNN framework [25, 51] every node, v_i, is initialized with a random embedding, $\mathbf{h}_i^0$ (node attributes are ignored), and at each iteration of the GNN algorithm nodes accumulate inputs from their neighbors using simple neural network layers:[9]

$$\mathbf{h}_i^k = \sum_{v_j \in \mathcal{N}(v_i)} \sigma(\mathbf{W}\mathbf{h}_j^{k-1} + \mathbf{b}), \tag{24}$$

where $\mathbf{W} \in \mathbb{R}^{d \times d}$ and $\mathbf{b} \in \mathbb{R}^d$ are trainable parameters and σ is a non-linearity (*e.g.*, $\tanh$ or a rectified linear unit). Equation (24) is repeatedly applied in a recursive fashion until the embeddings converge, and special care must be taken during initialization to ensure convergence [51]. Once the embeddings have converged, they are aggregated for the entire (sub)graph and this aggregated embedding is used for subgraph classification. Any of the aggregation procedures described in Section 3.1 could be employed, but Scarselli et al. [51] also suggest that the aggregation can be done by introducing a "dummy" super-node that is connected to all nodes in the target subgraph.

Li et al. [38] extend and modify the GNN framework to use Gated Recurrent Units and back propagation through time [14], which removes the need to run the recursion in Equation (24) to convergence. Adapting the GNN framework to use modern recurrent units also allows Li et al. to leverage node attributes and to use the output of intermediate embeddings of subgraphs.

The GNN framework is highly expressive, but it is also computationally intensive compared to the convolutional approaches, due to the complexities of ensuring convergence [51] or running back propagation through time [38]. Thus, unlike the convolutional approaches, which are most commonly used to classify molecular graphs in large datasets, the GNN approach has been used for more complex, but smaller scale, tasks, *e.g.*, for approximate formal verification using graph-based representations of programs [38].

3.3 Applications of subgraph embeddings

The primary use case for subgraph embeddings is for subgraph classification, which has important applications in a number of areas. The most prominent application domain is for classifying the properties of graphs corresponding to different molecules [16, 21, 43, 33]. Subgraph embeddings can be used to classify or predict various properties of molecular graphs, including predicting the efficacy of potential solar cell materials [16], or predicting the therapeutic effect of candidate drugs [33]. More generally, subgraph embeddings have been used to classify images (after converting the image to a graph representation) [8], to predict whether a computer program satisfies certain formal properties [38], and to perform logical reasoning tasks [38].

4 Conclusion and future directions

Representation learning approaches for machine learning on graphs offer a power alternative to traditional feature engineering. In recent years, these approaches have consistently pushed the state of the art on tasks such as node classification and link prediction. However, much work remains to be done, both in improving the performance of these methods, and—perhaps more importantly—in developing consistent theoretical frameworks that future innovations can build upon.

[9]Other parameterizations and variations are discussed in [51].

4.1 Challenges to future progress

In this review, we attempted to unify a number of previous works, but the field as a whole still lacks a consistent theoretical framework—or set of frameworks—that precisely delineate the goals of representation learning on graphs. At the moment, the implicit goal of most works is to generate representations that perform well on a particular set of classification or link prediction benchmarks (and perhaps also generate qualitatively pleasing visualizations). However, the unchecked proliferation of disparate benchmarks and conceptual models presents a real risk to future progress, and this problem is only exacerbated by the popularity of node and graph embedding techniques across distinct, and somewhat disconnected, subfields within the machine learning and data mining communities. Moving forward as a field will require new theoretical work that more precisely describes the kinds of graph structures that we expect the learned representations to encode, how we expect the models to encode this information, and what constraints (if any) should be imposed upon on these learned latent spaces.

More developed theoretical foundations would not only benefit researchers in the field—*e.g.*, by informing consistent and meaningful benchmark tasks—these foundations would also allow application domain-experts to more effectively choose and differentiate between the various approaches. Current methods are often evaluated on a variety of distinct benchmarks that emphasize various different graph properties (*e.g.*, community structures, relationship strengths between nodes, or structural roles). However, many real-world applications are more focused, and it is not necessary to have representations that are generically useful for a wide variety of tasks. As a field, we need to make it clear what method should be used when, and prescribing such use-cases requires a more precise theoretical understanding of what exactly our learned representations are encoding.

4.2 Important open problems

In addition to the general challenges outlined above, there are a number of concrete open problems that remain to be addressed within the area of representation learning on graphs.

Scalability. While most of the works we reviewed are highly scalable in theory (*i.e.*, $O(|\mathcal{E}|)$ training time), there is still significant work to be done in scaling node and graph embedding approaches to truly massive datasets (*e.g.*, billions of nodes and edges). For example, most methods rely on training and storing a unique embedding for each individual node. Moreover, most evaluation setups assume that the attributes, embeddings, and edge lists of all nodes used for both training and testing can fit in main memory—an assumption that is at odds with the reality of most application domains, where graphs are massive, evolving, and often stored in a distributed fashion. Developing representation learning frameworks that are truly scalable to realistic production settings is necessary to prevent widening the disconnect between the academic research community and the application consumers of these approaches.

Decoding higher-order motifs. While much work in recent years has been dedicated to refining and improving the encoder algorithm used to generate node embeddings, most methods still rely on basic pairwise decoders, which predict pairwise relations between nodes and ignore higher-order graph structures involving more than two nodes. It is well-known that higher-order structural motifs are essential to the structure and function of complex networks [5], and developing decoding algorithms that are capable of decoding complex motifs is an important direction for future work.

Modeling dynamic, temporal graphs. Many application domains involve highly dynamic graphs where timing information is critical—*e.g.*, instant messaging networks or financial transaction graphs. However, we lack embedding approaches that can cope with the unique challenges presented by temporal graphs, such as the task of incorporating timing information about edges. Temporal graphs are becoming an increasingly important object of study [45], and extending graph embedding techniques to operate over them will open up a wide range of exciting application domains.

Reasoning about large sets of candidate subgraphs. A major technical limitation of current subgraph embedding approaches is that they require the target subgraphs to be pre-specified before the learning process.

However, many applications seek to *discover* subgraphs with certain properties, and these applications require models that can reason over the combinatorially large space of *possible* candidate subgraphs. For example, one might want to discover central subgraphs in a gene regulatory network, or uncover nefarious sub-communities in a social network. We need improved subgraph embedding approaches that can efficiently reason over large sets of candidate subgraphs, as such improvements are critical to expand the usefulness of subgraph embeddings beyond the task of basic subgraph classification.

Improving interpretability. Representation learning is attractive because it relieves much of the burden of hand designing features, but it also comes at a well-known cost of interpretability. We know that embedding-based approaches give state-of-the-art performance, but the fundamental limitations—and possible underlying biases—of these algorithms are relatively unknown. In order to move forward, care must be taken to develop new techniques to improve the interpretability of the learned representations, beyond visualization and benchmark evaluation. Given the complexities and representational capacities of these approaches, researchers must be ever vigilant to ensure that their methods are truly learning to represent relevant graph information, and not just exploiting statistical tendencies of benchmarks.

Acknowledgments

The authors thank Marinka Zitnik, Zoubin Ghahramani, Richard Turner, Stephen Bach, and Manan Ajay Shah for their helpful discussions and comments on early drafts. This research has been supported in part by NSF IIS-1149837, DARPA SIMPLEX, Stanford Data Science Initiative, and Chan Zuckerberg Biohub. W.L.H. was also supported by the SAP Stanford Graduate Fellowship and an NSERC PGS-D grant. The views and conclusions expressed in this material are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above funding agencies, corporations, or the U.S. and Canadian governments.

References

[1] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A.J. Smola. Distributed large-scale natural graph factorization. In *WWW*, 2013.

[2] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1, 2008.

[3] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *WSDM*, 2011.

[4] M. Belkin and P. Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NIPS*, 2002.

[5] A.R. Benson, D.F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[6] S. Bhagat, G. Cormode, and S. Muthukrishnan. Node classification in social networks. In *Social Network Data Analytics*, pages 115–148. 2011.

[7] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[8] J. Bruna, W. Zaremba, and Y. Szlam, A.and LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014.

[9] S. Cao, W. Lu, and Q. Xu. Grarep: Learning graph representations with global structural information. In *KDD*, 2015.

[10] S. Cao, W. Lu, and Q. Xu. Deep neural networks for learning graph representations. In *AAAI*, 2016.

[11] B.P. Chamberlain, J. Clough, and M.P. Deisenroth. Neural embeddings of graphs in hyperbolic space. *arXiv preprint arXiv:1705.10359*, 2017.

[12] S. Chang, W. Han, J. Tang, G. Qi, C.C. Aggarwal, and T.S. Huang. Heterogeneous network embedding via deep architectures. In *KDD*, 2015.

[13] H. Chen, B. Perozzi, Y. Hu, and S. Skiena. Harp: Hierarchical representation learning for networks. *arXiv preprint arXiv:1706.07845*, 2017.

[14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

[15] Fan RK Chung. *Spectral Graph Theory*. Number 92. American Mathematical Soc., 1997.

[16] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *ICML*, 2016.

[17] M.C.F. De Oliveira and H. Levkowitz. From visual data exploration to visual data mining: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):378–394, 2003.

[18] M. Defferrard and P. Bresson, X.and Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, 2016.

[19] Y. Dong, N.V. Chawla, and A. Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *KDD*, 2017.

[20] C. Donnat, M. Zitnik, D. Hallac, and J. Leskovec. Graph wavelets for structural role similarity in complex networks. *Under review*, 2017.

[21] D. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R.P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2015.

[22] M. Ester, H. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[23] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[24] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT press, 2007.

[25] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, 2005.

[26] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *arXiv preprint arXiv:1605.09096*, 2017.

[27] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, 2016.

[28] W.L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *arXiv preprint, arXiv:1603.04467*, 2017.

[29] K. Henderson, B. Gallagher, T. Eliassi-Rad, H. Tong, S. Basu, L. Akoglu, D. Koutra, C. Faloutsos, and L. Li. Rolx: structural role extraction & mining in large graphs. In *KDD*, 2012.

[30] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[31] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[32] P. Hoff, A.E. Raftery, and M.S. Handcock. Latent space approaches to social network analysis. *JASA*, 97(460):1090–1098, 2002.

[33] S. Kearnes, K. McCloskey, M. Berndl, V. Pande, and P. Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of Computer-Aided Molecular Design*, 30(8):595–608, 2016.

[34] T.N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2016.

[35] T.N. Kipf and M. Welling. Variational graph auto-encoders. In *NIPS Workshop on Bayesian Deep Learning*, 2016.

[36] J.B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[37] J.A. Lee and M. Verleysen. *Nonlinear dimensionality reduction*. Springer Science & Business Media, 2007.

[38] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *ICLR*, 2015.

[39] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology*, 58(7):1019–1031, 2007.

[40] Q. Lu and L. Getoor. Link-based classification. In *ICML*, volume 3, pages 496–503, 2003.

[41] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *UAI*, 1999.

[42] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

[43] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *ICML*, 2016.

[44] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu. Asymmetric transitivity preserving graph embedding. In *KDD*, 2016.

[45] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in temporal networks. In *WSDM*, 2017.

[46] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *KDD*, 2014.

[47] B. Perozzi, V. Kulkarni, and S. Skiena. Walklets: Multiscale graph embeddings for interpretable network classification. *arXiv preprint arXiv:1605.02115*, 2016.

[48] Bryan Perozzi. *Local Modeling of Attributed Graphs: Algorithms and Applications*. PhD thesis, Stony Brook University, 2016.

[49] T. Pham, T. Tran, D.Q. Phung, and S. Venkatesh. Column networks for collective classification. In *AAAI*, 2017.

[50] L.F.R. Ribeiro, P.H.P. Saverese, and D.R. Figueiredo. struc2vec: Learning node representations from structural identity. In *KDD*, 2017.

[51] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[52] M. Schlichtkrull, T.N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. *arXiv preprint arXiv:1703.06103*, 2017.

[53] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In *WWW*, 2015.

[54] J. Tenenbaum, V. De Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[55] R. van den Berg, T.N. Kipf, and M. Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*, 2017.

[56] L. van der Maaten and G. Hinton. Visualizing data using t-sne. *JMLR*, 9:2579–2605, 2008.

[57] S.V.N. Vishwanathan, N.N. Schraudolph, R. Kondor, and K.M. Borgwardt. Graph kernels. *JMLR*, 11:1201–1242, 2010.

[58] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. In *KDD*, 2016.

[59] Z. Yang, W. Cohen, and R. Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *ICML*, 2016.

[60] M. Zitnik and J. Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 2017.

On Summarizing Large-Scale Dynamic Graphs

Neil Shah[*], Danai Koutra[†], Lisa Jin[‡], Tianmin Zou[§], Brian Gallagher[¶], Christos Faloutsos[*]

[*] Carnegie Mellon University, [†] University of Michigan, [‡] University of Rochester
[§] Google, [¶] Lawrence Livermore National Lab

Abstract

How can we describe a large, dynamic graph over time? Is it random? If not, what are the most apparent deviations from randomness – a dense block of actors that persists over time, or perhaps a star with many satellite nodes that appears with some fixed periodicity? In practice, these deviations indicate patterns – for example, research collaborations forming and fading away over the years. Which patterns exist in real-world dynamic graphs, and how can we find and rank their importance? These are exactly the problems we focus on. Our main contributions are (a) formulation*: we show how to formalize this problem as minimizing an information theoretic encoding cost, (b)* algorithm*: we propose* TIMECRUNCH*, an effective and scalable method for finding coherent, temporal patterns in dynamic graphs and (c)* practicality*: we apply our method to several large, diverse real-world datasets with up to* 36 million *edges and introduce our auxiliary* ECOVIZ *framework for visualizing and interacting with dynamic graphs which have been summarized by* TIMECRUNCH*. We show that* TIMECRUNCH *is able to compress these graphs by summarizing important temporal structures and finds patterns that agree with intuition.*

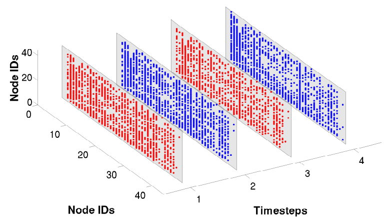
1 Introduction

Given a large phonecall network over time, how can we describe it to a practitioner with just a few phrases? Other than the traditional assumptions about real-world graphs involving degree skewness, what can we say about the connectivity? For example, is the dynamic graph characterized by many large cliques which appear at fixed intervals of time, or perhaps by several large stars with dominant hubs that persist throughout? Our work aims to answer these questions, and specifically, we focus on constructing concise summaries of large, real-world dynamic graphs in order to better understand their underlying behavior.

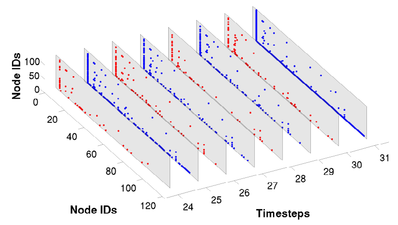
This problem has numerous practical applications. Dynamic graphs are ubiquitously used to model the relationships between various entities over *time*, which is a valuable feature in almost all applications in which nodes represent users or people. Examples include online social networks, phone-call networks, collaboration and coauthorship networks and other interaction networks.

Though numerous graph algorithms suitable for static contexts such as modularity, spectral and cut-based partitioning exist, they do not offer direct dynamic counterparts. Furthermore, the traditional goals of clustering and community detection tasks are not quite aligned with our goal. These algorithms typically produce groupings

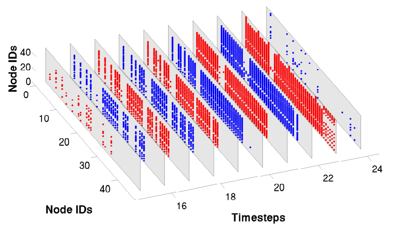
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



(a) 40 users of Yahoo! Messenger forming a *constant near clique* with unusually high 55% density, over 4 weeks in April 2008.



(b) 111 callers in a large phonecall network, forming a *periodic star*, over the last week of December 2007 – note the heavy activity on holidays.



(c) 43 collaborating biotechnology authors forming a *ranged near clique* in the DBLP network, jointly publishing through 2005-2012.

Figure 1: **TIMECRUNCH finds coherent, interpretable temporal structures**. We show the reordered subgraph adjacency matrices, over the timesteps of interest, each outlined in gray; edges are plotted in alternating red and blue, for discernibility.

of nodes which satisfy or approximate some optimization function. However, they do not offer interpretation or characterization of the outputs.

In this work, we propose TIMECRUNCH, an approach for concisely summarizing large, dynamic graphs which extend beyond traditional dense, isolated "cavemen" communities. Our method works by leveraging MDL (Minimum Description Length) in order to represent graphs over time using a lexicon of *temporal phrases* which describe temporal connectivity behavior. Figure 1 shows several interesting results found from applying TIMECRUNCH to real-world dynamic graphs.

- Figure 1a shows a *constant near-clique* of likely bots on Yahoo! Messenger.
- Figure 1b depicts a *periodic star* of possible telemarketers on a phonecall network.
- Lastly, Figure 1c shows a *ranged near clique* of many authors jointly publishing in a biology journal.

In this work, we seek to answer the following informally posed problem:

Problem 1 (Informal): Given a dynamic graph, **find** a set of possibly overlapping temporal subgraphs to **concisely describe** the given dynamic graph in a **scalable** fashion.

Our main contributions are as follows:

1. **Formulation:** We define the problem of dynamic graph understanding in in a compression context.
2. **Algorithm:** We develop TIMECRUNCH, a fast algorithm for dynamic graph summarization.
3. **Practicality:** We show quantitative and qualitative results of TIMECRUNCH on several real graphs, and also discuss ECOVIZ for interactive dynamic graph visualization.

Reproducibility: Our code for TIMECRUNCH is open-sourced at `www.cs.cmu.edu/~neilshah/code/timecrunch.tar`.

2 Related Work

The related work falls into three main categories: static graph mining, temporal graph mining, and graph compression and summarization.

Static Graph Mining. Most works find specific, tightly-knit structures, such as (near-) cliques and bipartite cores: eigendecomposition [24], cross-associations [7], modularity-based optimization methods [20, 5]. Dhillon et al. [10] propose information theoretic co-clustering based on mutual information optimization. However, these approaches have limited structural vocabularies. [14, 17] propose cut-based partitioning, whereas [3] suggests spectral partitioning using multiple eigenvectors – these schemes seek hard clustering of all nodes as opposed to identifying communities, and require parameters. Subdue [8] and other fast frequent-subgraph mining algorithms [12] operate on labeled graphs. Our work involves unlabeled graphs and lossless compression.

Temporal Graph Mining. Most work on temporal graphs focuses on the evolution of specific properties, change detection, or community detection. For example, [2] aims at change detection in streaming graphs using projected clustering. This approach focuses on anomaly detection rather than mining temporal patterns. GraphScope [27] uses graph search for hard-partitioning of temporal graphs to find dense temporal cliques and bipartite cores. Com2 [4] uses CP/PARAFAC decomposition with MDL for the same. [11] uses incremental cross-association for change detection in dense blocks, whereas [22] proposes an algorithm for mining atemporal cross-graph quasi-cliques. These approaches have limited vocabularies and no temporal interpretability. Dynamic clustering [29] finds stable clusters over time by penalizing deviations from incremental static clustering. Our work focuses on interpretable structures, which may not appear at every timestep.

Graph Compression and Summarization. Work on summarization and compression of time-evolving graphs is quite limited [30]. Some examples for compressing *static* graphs include SlashBurn [13], which is a recursive node-reordering approach to leverage run-length encoding, weighted graph compression [28] that uses structural equivalence to collapse nodes/edges to simplify graph representation, two-part MDL representation with error bounds [31], and domain-specific summarization using graph invariants [32]. VoG [16] uses MDL to label subgraphs in terms of a vocabulary on static graphs, consisting of stars, (near) cliques, (near) bipartite cores and chains. This approach only applies to static graphs and does not offer a clear extension to dynamic graphs. Our work proposes a suitable lexicon for dynamic graphs, uses MDL to label *temporally coherent* subgraphs and proposes an effective and scalable algorithm for finding them. More recent works on time-evolving networks include graph stream summarization [34] for query efficiency, and influence-based graph summarization [35, 33], which aim to summarize network propagation processes.

3 Problem Formulation

In this section, we give the first main contribution of our work: formulation of dynamic graph summarization as a compression problem.

The Minimum Description Length (MDL) principle aims to be a practical version of Kolmogorov Complexity [19], often associated with the motto *Induction by Compression*. MDL states that given a model family $\mathcal{M}$, the best model $M \in \mathcal{M}$ for some observed data $\mathcal{D}$ is that which minimizes $L(M) + L(\mathcal{D}|M)$, where $L(M)$ is the length in bits used to describe M and $L(\mathcal{D}|M)$ is the length in bits used to describe $\mathcal{D}$ encoded using M. MDL enforces lossless compression for fairness in the model selection process.

We focus on analysis of undirected dynamic graphs using fixed-length, discretized time intervals. We consider a dynamic graph $G(\mathcal{V}, \mathcal{E})$ with $n = |\mathcal{V}|$ nodes, $m = |\mathcal{E}|$ edges and t timesteps, without self-loops. Here, $G = \cup_x G_x(\mathcal{V}, \mathcal{E}_x)$, where G_x and E_x correspond to the graph and edge-set for the x^{th} timestep.

For our summary, we consider the set of temporal phrases $\Phi = \Delta \times \Omega$, where Δ corresponds to the set of temporal signatures, Ω corresponds to the set of static structure identifiers and $\times$ denotes Cartesian set product. Though we can include arbitrary temporal signatures and static structure identifiers into these sets depending on the types of temporal subgraphs we expect to find in a given dynamic graph, we choose 5 temporal signatures which we anticipate to find in real-world dynamic graphs: oneshot (*o*), ranged (*r*), periodic (*p*), flickering (*f*) and constant (*c*):

- *Oneshot* structures appear at only one timestep
- *Ranged* structures appear for a series of consecutive timesteps
- *Periodic* structures appear at fixed intervals of time
- *Flickering* structures do not have discernible periodicity, but occur multiple times
- *Constant* structures appear at every timestep

and 6 very common structures found in real-world static graphs [15, 24, 16] – stars (*st*), *full* and *near* cliques (*fc*, *nc*), *full* and *near* bipartite cores (*bc*, *nb*) and chains (*ch*):

- *Stars* are characteristic of a hub node connected to 2 or more "spokes"
- *(Near) Cliques* are sets of nodes with very dense interconnectivity
- *(Near) Bipartite Cores* consist of non-intersecting node sets L and R for which there exist only edges between L and R but not within
- *Chains* are a series of nodes in which each node is connected to the next

Summarily, we have the signatures $\Delta = \{o, r, p, f, c\}$, static identifiers $\Omega = \{st, fc, nc, bc, nb, ch\}$ and temporal phrases $\Phi = \Delta \times \Omega$.

In order to use MDL for dynamic graph summarization using these temporal phrases, we next define the model family $\mathcal{M}$, the means by which a model $M \in \mathcal{M}$ describes our dynamic graph and how to quantify the cost of encoding in terms of bits.

3.1 Using MDL for Dynamic Graph Summarization

We consider models $M \in \mathcal{M}$ to be composed of ordered lists of temporal graph structures with node, but not edge overlaps. Each $s \in M$ describes a certain region of the adjacency tensor $\mathbf{A}$ in terms of the interconnectivity of its nodes – note that nonzero $\mathbf{A}_{i,j,k}$ indicates edge (i, j) exists in timestep k .

Our model family $\mathcal{M}$ consists of all possible permutations of subsets of $\mathcal{C}$, where $\mathcal{C} = \cup_v \mathcal{C}_v$ and $\mathcal{C}_v$ denotes the set of all possible temporal structures of phrase $v \in \Phi$ over all possible combinations of timesteps. That is, $\mathcal{M}$ consists of all possible models M, which are ordered lists of temporal phrases $v \in \Phi$ such as flickering stars (*fst*), periodic full cliques (*pfc*), etc. over all possible subsets of $\mathcal{V}$ and $G_1 \cdots G_t$. Through MDL, we seek $M \in \mathcal{M}$ which best mediates between the length of M and the adjacency tensor $\mathbf{A}$ given M.

Our high-level approach for transmitting the adjacency tensor $\mathcal{A}$ via the model M is described as follows: First, we transmit M. Next, given M, we induce the approximation of the adjacency tensor $\mathbf{M}$ as described by each temporal structure $s \in M$ – for each structure s, we induce the edges described by s in $\mathbf{M}$ accordingly. Given that $\mathbf{M}$ is a summary approximation to $\mathbf{A}$, $\mathbf{M} \neq \mathbf{A}$ most likely. Since MDL requires lossless encoding, we must also transmit the error $\mathbf{E} = \mathbf{M} \oplus \mathbf{A}$, obtained by taking the exclusive OR between $\mathbf{M}$ and $\mathbf{A}$. Given M and $\mathbf{E}$, a recipient can construct the full adjacency tensor $\mathbf{A}$ in a lossless fashion.

Thus, we formalize the problem we tackle as follows:

Problem 2 (Minimum Dynamic Graph Description): Given a dynamic graph G with adjacency tensor $\mathbf{A}$ and temporal phrase lexicon Φ, find the smallest model M which minimizes the total encoding length

$$L(G, M) = L(M) + L(\mathbf{E})$$

where $\mathbf{E}$ is the error matrix computed by $\mathbf{E} = \mathbf{M} \oplus \mathbf{A}$ and $\mathbf{M}$ is the approximation of $\mathbf{A}$ induced by M.

3.2 Encoding the Model and Errors

To fully describe a model $M \in \mathcal{M}$, we have the following:

$$L(M) = L_{\mathbb{N}}(|M| + 1) + log_2\binom{|M| + |\Phi| - 1}{|\Phi - 1|} + \sum_{s \in M} (-log_2 P(v(s)|M) + L(c(s)) + L(u(s)))$$

We begin by transmitting the total number of temporal structures in M using $L_{\mathbb{N}}$, Rissanen's optimal encoding for integers greater than or equal to 1 [23]. Next, we optimally encode the number of temporal structures for each phrase $v \in \Phi$ in M. Then, for each structure s, we encode the type $v(s)$ for each structure $s \in M$ using optimal prefix codes [9], the connectivity $c(s)$ and the temporal presence of the s, consisting of the ordered list of timesteps $u(s)$ in which s appears.

In order to have a coherent model encoding scheme, we must define the encoding for each phrase $v \in \Phi$ such that we can compute $L(c(s))$ and $L(u(s))$ for all structures in M. The connectivity $c(s)$ corresponds to the edges which are induced by s, whereas the temporal presence $u(s)$ corresponds to the timesteps in which s is present. We consider the connectivity and temporal presence separately, as the encoding for a temporal structure s described by a phrase v is the sum of encoding costs for the connectivity of the corresponding static structure identifier in Ω and its temporal presence as indicated by a temporal signature in Δ. Due to space constraints, we refer the interested reader to more detailed manuscripts [16, 25] for details regarding encoding processes and costs for the connectivity $L(c(s))$, temporal presence $L(u(s))$ and associated errors. In a nutshell, we have different encoding costs for encoding any subgraph and temporal recurrence pattern using a particular phrase in our lexicon Φ.

Remark: For a dynamic graph G of n nodes, the search space $\mathcal{M}$ for the best model $M \in \mathcal{M}$ is intractable, as it consists of all permutations of all possible temporal structures over the lexicon Φ, over all possible subsets over the node-set $\mathcal{V}$ and over all possible graph timesteps $G_1 \cdots G_t$. Furthermore, $\mathcal{M}$ is not easily exploitable for efficient search. As a result, we propose several practical approaches for the purpose of finding good and interpretable temporal models/summaries for G.

4 Proposed Method: TIMECRUNCH

Thus far, we have described our strategy of formulating dynamic graph summarization as a problem in a compression context for which we can leverage MDL. Specifically, we have described how to encode a model and the associated error which can be used to losslessly reconstruct the original dynamic graph G. Our models are characterized by ordered lists of temporal structures which are further classified as *phrases* from the lexicon Φ – that is, each $s \in M$ is identified by a phrase $p \in \Phi$ – over the node connectivity $c(s)$ (an induced set of edges depending on the static structure identifier st, fc, etc.) and the associated temporal presence $u(s)$ (ordered list of timesteps captured by a temporal signature o, r, etc. and deviations) in which the temporal structure is active, while the error consists of those edges which are not covered by $\mathbf{M}$, or the approximation of $\mathbf{A}$ induced by M.

Next, we discuss how we find good candidate temporal structures to populate the candidate set $\mathcal{C}$, as well as how we find the best model M with which to summarize our dynamic graph. The pseudocode for our algorithm is given in Alg. 1 and the next subsections detail each step of our approach.

4.1 Generating Candidate Static Structures

TIMECRUNCH takes an incremental approach to dynamic graph summarization. Our approach begins by considering potentially useful subgraphs over static graphs $G_1 \cdots G_t$. Section 2 mentions several such algorithms for community detection and clustering including EigenSpokes, METIS, SlashBurn, etc. Summarily, for each $G_1 \cdots G_t$, a set of subgraphs $\mathcal{F}$ is produced.

Algorithm 1 TIMECRUNCH

```
1: Generating Candidate Static Structures: Generate static subgraphs for each $G_1 \cdots G_t$ using traditional static graph decomposition approaches.
2: Labeling Candidate Static Structures: Label each static subgraph as a static structure corresponding to the identifier $x \in \Omega$ which minimizes the local encoding cost.
3: Stitching Candidate Temporal Structures: Stitch static structures from $G_1 \cdots G_t$ together to form temporal structures with coherent connectivity and label them according to the phrase $p \in \Phi$ which minimizes temporal presence encoding cost. Populate the candidate set $\mathcal{C}$.
4: Composing the Summary: Compose a model $M$ of important, non-redundant temporal structures which summarize $G$ using the VANILLA, TOP-10, TOP-100 and STEPWISE heuristics. Choose $M$ associated with the heuristic that produces the smallest total encoding cost.
```

4.2 Labeling Candidate Static Structures

Once we have the set of static subgraphs from $G_1 \cdots G_t$, $\mathcal{F}$, we next seek to label each subgraph in $\mathcal{F}$ according to the static structure identifiers in Ω that best fit the connectivity for the given subgraph. That is, for each subgraph construed as a set of nodes $\mathcal{L} \in \mathcal{V}$ for a fixed timestep, does the adjacency matrix of $\mathcal{L}$ best resemble a star, near or full clique, near or full bipartite core or a chain? To answer this question, we try encoding the subgraph $\mathcal{L}$ using each of the static identifiers in Ω and label it with the identifier $x \in \Omega$ which minimizes the encoding cost.

Consider the model ω which consists of only the subgraph $\mathcal{L}$ and a yet to be determined static identifier. In practice, instead of computing the global encoding cost $L(G, \omega)$ when encoding $\mathcal{L}$ as each static identifier in Ω to find the best fit, we compute the *local* encoding cost defined as $L(\omega) + L(\mathbf{E}_\omega^+) + L(\mathbf{E}_\omega^-)$ where $L(\mathbf{E}_\omega^+)$ and $L(\mathbf{E}_\omega^-)$ indicate the encoding costs for the extraneous and unmodeled edges for the subgraph $\mathcal{L}$ respectively. This is done for purpose of efficiency – intuitively, however, the static identifier that best describes $\mathcal{L}$ is independent of the edges outside of $\mathcal{L}$.

The challenge in this labeling step is that before we can encode $\mathcal{L}$ as any type of identifier, we must identify a suitable permutation of nodes in the subgraph so that our model encodes the correct edges. For example, if $\mathcal{L}$ is a star, which is the hub? Or if $\mathcal{L}$ is a bipartite core, how can we distinguish the parts? We resort to heuristics, as some of these tasks are computationally difficult to perform exactly – for example, finding the best configuration of nodes to form a bipartite core is equivalent to finding the maximum cut which is NP-hard. Details of appropriate configurations for each static structure are given in [16] for space constraints.

4.3 Stitching Candidate Temporal Structures

Thus far, we have a set of static subgraphs $\mathcal{F}$ over $G_1 \cdots G_t$ labeled with the associated static identifiers which best represent subgraph connectivity (from now on, we refer to $\mathcal{F}$ as a set of static *structures* instead of *subgraphs* as they have been labeled with identifiers). From this set, our goal is to find meaningful temporal structures – namely, we seek to *find* static subgraphs which have the same patterns of connectivity over one or more timesteps and *stitch* them together. Thus, we formulate the problem of finding coherent temporal structures in G as a clustering problem over $\mathcal{F}$. Though there are several criteria we could use for clustering static structures together, we employ the following based on their intuitive meaning: two structures in the same cluster should have (a) substantial overlap in the node-sets composing their respective subgraphs, and (b) exactly the same, or similar (full and near clique, or full and near bipartite core) static structure identifiers. These criteria, if satisfied, allow us to find groups of nodes that share interesting connectivity patterns over time.

Conducting the clustering by naively comparing each static structure in $\mathcal{F}$ to the others will produce the desired result, but is *quadratic* on the number of static structures and is thus undesirable from a scalability point of view. Instead, we propose an incremental approach using repeated rank-1 Singular Value Decomposition (SVD) for clustering the static structures, which offers *linear* time complexity on the number of edges m in G.

We begin by defining $\mathbf{B}$ as the *structure-node membership matrix* (SNMM) of G. $\mathbf{B}$ is defined to be of dimensions $|\mathcal{F}| \times |\mathcal{V}|$, where $\mathbf{B}_{i,j}$ indicates whether the ith row (structure) in $\mathcal{F}$ ($\mathbf{B}$) contains node j in its node-set. Thus, $\mathbf{B}$ is a matrix indicating the membership of nodes in $\mathcal{V}$ to each of the static structures in $\mathcal{F}$. We note that any two equivalent rows in $\mathbf{B}$ are characterized by structures that share the same node-set (but possibly different static identifiers). As our clustering criteria mandate that we cluster only structures with the same or similar static identifiers, in our algorithm, we construct 4 SNMMs – $\mathbf{B}_{st}$, $\mathbf{B}_{cl}$, $\mathbf{B}_{bc}$ and $\mathbf{B}_{ch}$ corresponding to the associated matrices for stars, near and full cliques, near and full bipartite cores and chains respectively. Now, any two equivalent rows in $\mathbf{B}_{cl}$ are characterized by structures that share the same-node set and the same, or similar static identifiers, and analogue for the other matrices. Next, we utilize SVD to cluster the rows in each SNMM, effectively clustering the structures in $\mathcal{F}$.

Recall that the rank-k SVD of an $m \times n$ matrix $\mathbf{A}$ factorizes $\mathbf{A}$ into 3 matrices – the $m \times k$ matrix of left-singular vectors $\mathbf{U}$, the $k \times k$ diagonal matrix of singular values $\mathbf{\Sigma}$ and the $n \times k$ matrix of right-singular vectors $\mathbf{V}$, such that $\mathbf{A} = \mathbf{U\Sigma V^T}$. A rank-$k$ SVD effectively reduces the input data into the best k-dimensional representation, each of which can be mined separately for clustering and community detection purposes. However, one major issue with using SVD in this fashion is that identifying the desired number of clusters k upfront is a non-trivial task. To this end, [21] evidences that in cases where the input matrix is sparse, repeatedly clustering using k rank-1 decompositions and adjusting the input matrix accordingly approximates the batch rank-k decomposition. This is a valuable result in our case – as we do not initially know the number of clusters needed to group the structures in $\mathcal{F}$, we eliminate the need to define k altogether by repeatedly applying rank-1 SVD using power iteration and removing the discovered clusters from each SNMM until all clusters have been found (when all SNMMs are fully sparse and thus *deflated*). However, in practice, full deflation is unneeded for summarization purposes, as most "important" clusters are found in early iterations due to the nature of SVD. For each of the SNMMs, the matrix $\mathbf{B}$ used in the $(i+1)^{th}$ iteration of this iterative process is computed as

$$\mathbf{B}^{i+1} = \mathbf{B}^i - I^{\mathcal{G}_i} \circ \mathbf{B}^i$$

where $\mathcal{G}_i$ denotes the set of row ids corresponding to the structures which were clustered together in iteration i, $I^{\mathcal{G}_i}$ denotes the indicator matrix with 1s in rows specified by $\mathcal{G}_i$ and $\circ$ denotes the Hadamard matrix product. This update to $\mathbf{B}$ is needed between iterations, as without subtracting out the previously-found cluster, repeated rank-1 decompositions would find the same cluster ad infinitum and the algorithm would not converge.

Although this algorithm works assuming we can remove a cluster in each iteration, the question of how we find this cluster given a singular vector has yet to be answered. First, we sort the singular vector, permuting the rows by magnitude of projection. The intuition is that the structure (rows) which projects most strongly to that cluster is the best representation of the cluster, and is considered a *base* structure which we attempt to find matches for. Starting from the base structure, we iterate down the sorted list and compute the Jaccard similarity, defined as $J(\mathcal{L}_1, \mathcal{L}_2) = |\mathcal{L}_1 \cap \mathcal{L}_2| / |\mathcal{L}_1 \cup \mathcal{L}_2|$ for node-sets $\mathcal{L}_1$ and $\mathcal{L}_2$, between each structure and the base. Other structures which are composed of the same, or similar node-sets will also project strongly to the cluster, and be stitched to the base. Once we encounter a series of structures which fail to match by a predefined similarity criterion, we adjust the SNMM and continue with the next iteration.

Having stitched together the relevant static structures, we label each temporal structure using the temporal signature in Δ and resulting phrase in Φ which minimizes its encoding cost. We use these temporal structures to populate the candidate set $\mathcal{C}$ for our model.

4.4 Composing the Summary

Given the candidate set of temporal structures $\mathcal{C}$, we next seek to find the model M which best summarizes G. However, actually finding the best model is combinatorial, as it involves considering all possible permutations of subsets of $\mathcal{C}$ and choosing the one which gives the smallest encoding cost. As a result, we propose several heuristics that give fast and approximate solutions without entertaining the entire search space. To reduce the search space, we associate with each temporal structure a metric by which we measure quality, called the *local encoding benefit*. The local encoding benefit is defined as the ratio between the cost of encoding the given temporal structure as error and the cost of encoding it using the best phrase (local encoding cost). Large local encoding benefits indicate high compressibility, and thus meaningful structure in the underlying data. Our proposed heuristics are as follows:

VANILLA: This is the baseline approach, in which our summary contains all the structures from the candidate set, or $M = \mathcal{C}$.

TOP-K: In this approach, M consists of the top k structures of $\mathcal{C}$, sorted by local encoding benefit.

STEPWISE: This approach involves considering each structure of $\mathcal{C}$, sorted by local encoding benefit, and adding it to M if the global encoding cost decreases. If adding the structure to M increases the global encoding cost, the structure is discarded as redundant or not worthwhile for summarization purposes.

In practice, TIMECRUNCH uses each of the heuristics and identifies the best summary for G as the one that produces the minimum encoding cost.

5 Experiments

In this section, we evaluate TIMECRUNCH and seek to answer the following questions: Are real-world dynamic graphs well-structured, or noisy and indescribable? If they are structured, how so – what temporal structures do we see in these graphs and what do they mean?

5.1 Datasets and Experimental Setup

For our experiments, we use 5 real dynamic graph datasets – we briefly describe them below.

Enron: The `Enron` e-mail dataset is publicly available [26]. It contains 20K unique links between 151 users based on e-mail correspondence, over 163 weeks (May 1999 - June 2002).

Yahoo! IM: The `Yahoo-IM` dataset is publicly available [36]. It contains 2.1M sender-receiver pairs between 100K users over 5.7K zip-codes selected from the Yahoo! messenger network over 4 weeks starting from April 1st, 2008.

Honeynet: The `Honeynet` dataset is not publicly available. It contains information about network attacks on *honeypots* (i.e., computers which are left intentionally vulnerable). It contains source and destination IPs, and attack timestamps of 372K (attacker and honeypot) machines with 7.1M unique daily attacks over a span of 32 days starting from December 31st, 2013.

DBLP: The `DBLP` computer science bibliography is publicly available, and contains yearly co-authorship information [1]. We used a subset of DBLP spanning 25 years, from 1990 to 2014, with 1.3M authors and 15M unique author-author collaborations over the years.

Phonecall: The `Phonecall` dataset is not publicly available. It describes the who-calls-whom activity of 6.3M individuals from a large, anonymous Asian city and contains a total of 36.3M unique daily phonecalls. It spans 31 days, starting from December 1st, 2007.

In our experiments, we use SlashBurn [13] for generating candidate static structures, as it is scalable and designed to extract structure from real-world, non-"cavemen" graphs. We note that including other graph decomposition methods can be used for various applications instead of SlashBurn. Furthermore, when clustering each sorted singular vector during the stitching process, we move on with the next iteration of matrix deflation after 10 failed matches with a Jaccard similarity threshold of 0.5 – we choose 0.5 based on experimental results which show that it gives the best encoding cost and balances between excessively terse and overlong (error-prone) models. Lastly, we run TIMECRUNCH for a total of 5000 iterations for all graphs (each iteration uniformly selects one SNMM to mine, resulting in 5000 total temporal structures), except for the `Enron` graph which is fully deflated after 563 iterations and the `Phonecall` graph which we limit to 1000 iterations for efficiency.

5.2 Quantitative Analysis

In this section, we use TIMECRUNCH to summarize each of the real-world dynamic graphs discussed above and report the resulting encoding costs. Specifically, evaluation is done by comparing the compression ratio between encoding costs of the resulting models to the null encoding (ORIGINAL) cost, which is obtained by encoding the graph using an empty model.

We note that although we provide results in a compression context, compression is *not* our main goal for TIMECRUNCH, but rather the means to our end for identifying suitable structures with which to summarize

Graph	ORIGINAL (bits)	TIMECRUNCH			
		VANILLA	TOP-10	TOP-100	STEPWISE
`Enron`	$86,102$	89% (563)	88%	81%	**78%** (130)
`Yahoo-IM`	$16,173,388$	97% (5000)	99%	98%	**93%** (1523)
`Honeynet`	$72,081,235$	82% (5000)	96%	89%	**81%** (3740)
`DBLP`	$167,831,004$	97% (5000)	99%	99%	**96%** (1627)
`Phonecall`	$478,377,701$	100% (1000)	100%	99%	**98%** (370)

Table 1: **TIMECRUNCH finds temporal structures that compress real graphs.** ORIGINAL denotes cost in bits for encoding each graph with an empty model. Other columns show relative costs for encoding using the respective heuristic (size of model in parentheses). The lowest description cost is bolded.

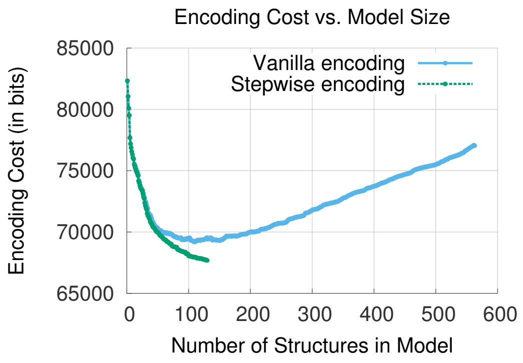


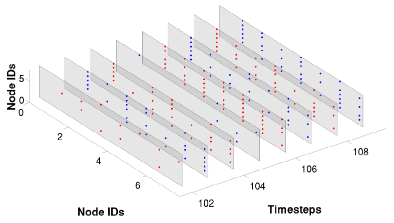
Figure 2: **TIMECRUNCH-STEPWISE summarizes `Enron` using just 78% of ORIGINAL's bits and 130 structures compared to 89% and 563 structures of TIMECRUNCH-VANILLA by pruning unhelpful structures from the candidate set.**

dynamic graphs and route the attention of practitioners. For this reason, we do not evaluate against other, compression-oriented methods which prioritize leveraging any correlation within the data to reduce cost and save bits. Other temporal clustering and community detection approaches which focus only on extracting dense blocks are also not compared to for similar reasons.

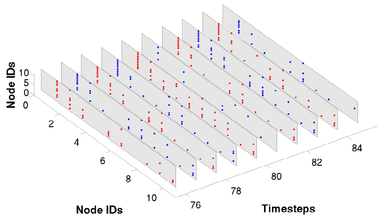
In our evaluation, we consider (a) ORIGINAL and (b) TIMECRUNCH summarization using the proposed heuristics. In the ORIGINAL approach, the entire adjacency tensor is encoded using the empty model $M = \emptyset$. As the empty model does not describe any part of the graph, all the edges are encoded using $L(\mathbf{E}^-)$. We use this as a baseline to evaluate the savings attainable using TIMECRUNCH. For summarization using TIMECRUNCH, we apply the VANILLA, TOP-10, TOP-100 and STEPWISE model selection heuristics. We note that we ignore small structures of < 5 nodes for `Enron` and < 8 nodes for the other, larger datasets.

Table 1 shows the results of our experiments in terms of encoding costs of various summarization techniques as compared to the ORIGINAL approach. Smaller compression ratios indicate better summaries, with more structure explained by the respective models. For example, STEPWISE was able to encode the `Enron` dataset using just 78% of the bits compared to 89% using VANILLA. In our experiments, we find that the STEPWISE heuristic produces models with considerably fewer structures than VANILLA, while giving even more concise graph summaries (Fig. 2). This is because it is highly effective in pruning redundant, overlapping or error-prone structures from the candidate set $\mathcal{C}$, by evaluating new structures in the context of previously seen ones.

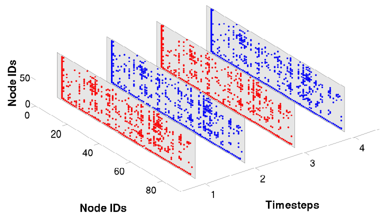
Our results indicate that real-world dynamic graphs are in fact structured, as TIMECRUNCH gives better encoding cost than ORIGINAL.



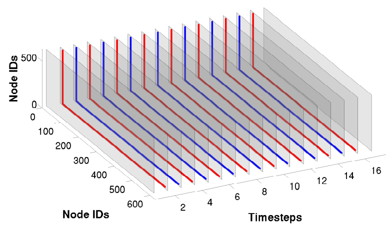
(a) 8 employees of the `Enron` legal team forming a *flickering near clique*



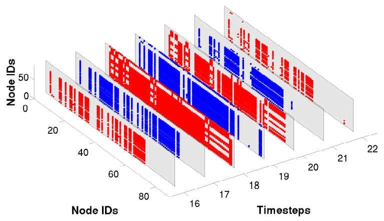
(b) 10 employees of the `Enron` legal team forming a *flickering star* with the boss as the hub



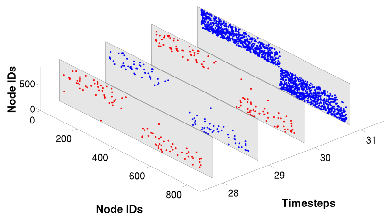
(c) 82 users in `Yahoo-IM` forming a *constant star* over the observed 4 weeks



(d) 589 honeypot machines were attacked on `Honeynet` over 2 weeks, forming a *ranged star*



(e) 82 authors forming a *ranged near clique* on `DBLP`, with burgeoning collaboration from timesteps 18-20 (2007-2009)



(f) 792 callers in `Phonecall` forming a *oneshot near bipartite core* appearing strongly on Dec. 31

Figure 3: **TIMECRUNCH finds meaningful temporal structures in real graphs.** We show the reordered subgraph adjacency matrices over multiple timesteps. Individual timesteps are outlined in gray, and edges are plotted with alternating red and blue color for discernibility.

	st	**fc**	**ch**
r	9	-	-
p	93	7	1
f	3	1	-
c	-	-	-
o	15	1	-

(a) `Enron`

	st	**fc**	**nc**	**bc**	**nb**	**ch**
r	147	43	-	1	45	6
p	59	25	-	-	42	3
f	179	55	-	1	62	3
c	185	118	-	-	66	-
o	295	129	1	2	56	-

(b) `Yahoo-IM`

	st	**bc**
r	56	-
p	125	1
f	39	-
c	-	-
o	3512	7

(c) `Honeynet`

	st	**fc**	**nb**	**ch**
r	43	80	-	5
p	19	26	-	-
f	1	-	-	-
c	-	-	-	-
o	516	840	97	-

(d) `DBLP`

	st	**fc**	**nc**	**bc**
r	15	-	-	-
p	68	-	-	1
f	88	-	-	-
c	5	-	-	-
o	187	4	1	1

(e) `Phonecall`

Table 2: Frequency of each temporal structure type discovered using TIMECRUNCH-STEPWISE for each dataset.

5.3 Qualitative Analysis

In this section, we discuss qualitative results from applying TIMECRUNCH to the real-world datasets.

Enron: The `Enron` graph is characteristic of many periodic, ranged and oneshot stars and several periodic and flickering cliques. Periodicity is reflective of office e-mail communications (e.g. meetings, reminders). Figure 3a shows an excerpt from one flickering clique which corresponds to the several members of Enron's legal team, including Tana Jones, Susan Bailey, Marie Heard and Carol Clair – all lawyers at Enron. Figure 3b shows an excerpt from a flickering star, corresponding to many of the same members as the flickering clique – the center of this star was identified as the boss, Tana Jones (Enron's Senior Legal Specialist). Note that the satellites of the star oscillate over time. Interestingly, the flickering star and clique extend over most of the observed duration. Furthermore, several of the oneshot stars corresponds to company-wide emails sent out by key players John Lavorato (Enron America CEO), Sally Beck (COO) and Kenneth Lay (CEO/Chairman).

Yahoo! IM: The `Yahoo-IM` graph is composed of many temporal stars and cliques of all types, and several smaller bipartite cores with just a few members on one side (indicative of friends who share mostly similar friend-groups but are themselves unconnected). We observe several interesting patterns in this data – Fig. 3c corresponds to a constant star with a hub that communicates with 70 users consistently over 4 weeks. We suspect that these users are part of a small office network, where the boss uses group messaging to notify employees of important updates or events – we notice that very few edges of the star are missing each week and the average degree of the satellites is roughly 4, corresponding to possible communication between employees. Figure 1a depicts a constant clique between 40 users, with an average density over 55% – we suspect that these may be spam-bots messaging each other in an effort to appear normal.

Honeynet: `Honeynet` is a bipartite graph between attacker and honeypot (victim) machines. As such, it is characterized by temporal stars and bipartite cores. Many of the attacks only span a single day, as indicated by the presence of 3512 oneshot stars, and no attacks span the entire 32 day duration. Interestingly, 2502 of these oneshot star attacks (71%) occur on the first and second observed days (Dec. 31 and Jan. 1st) indicating intentional "new-year" attacks. Figure 3d shows a ranged star, lasting 15 consecutive days and targeting 589 machines for the entire duration of the attack.

DBLP: Agreeing with intuition, `DBLP` consists of a large number of oneshot temporal structures corresponding to many single instances of joint publication. However, we also find numerous ranged/periodic stars and cliques which indicate coauthors publishing in consecutive years or intermittently. Figure 1c shows a ranged clique spanning from 2007-2012 between 43 coauthors who jointly published each year. The authors are mostly members of the NIH NCBI (National Institute of Health National Center for Biotechnology Information) and have published their work in various biotechnology journals such as *Nature*, *Nucleic Acids Research* and *Genome Research*. Figure 3e shows another ranged clique from 2005 to 2011, consisting of 83 coauthors who jointly publish each year, with an especially collaborative 3 years (timesteps 18-20) corresponding to 2007-2009 before returning to status quo.

Phonecall: The `Phonecall` dataset is largely comprised of temporal stars and few dense clique and bipartite structures. Again, we have a large proportion of oneshot stars which occur only at single timesteps. Further analyzing these results, we find that 111 of the 187 oneshot stars (59%) are found on Dec. 24, 25 and 31st, corresponding to Christmas Eve/Day and New Year's Eve holiday greetings. Furthermore, we find many periodic and flickering stars typically consisting of 50-150 nodes, which may be associated with businesses regularly contacting their clientele, or public phones which are used consistently by the same individuals. Figure 1b shows one such periodic star of 111 users over the last week of December, with particularly clear star structure on Dec. 25th and 31st and other odd-numbered days, accompanied by substantially weaker star structure on the even-numbered days. Figure 3f shows an oddly well-separated oneshot near-bipartite core which appears on Dec. 31st, consisting of two roughly equal-sized parts of 402 and 390 callers.

6 Application: Leveraging TIMECRUNCH for Interactive Visualization

One promising application of TIMECRUNCH is for dynamic graph visualization. In this section, we overview ECOVIZ (for Evolving COmparative network visualization), an interactive web application which enables pairwise comparison and temporal analysis of TIMECRUNCH's dynamic graph summary output. ECOVIZ aims to (i) adapt TIMECRUNCH to domain-specific requirements and (ii) provide efficient querying and visualization of its summary structures.

Data: We illustrate ECOVIZ using a connectomics application, which we briefly introduce for context. Connectomics involves the inference of functional brain networks from fMRI data [6]. Regions of the brain are discretized into "voxels," between which edges are inferred based on the strength of inter-voxel time series correlations. To obtain sparse brain networks (instead of full cliques), a user-defined threshold is applied to keep only the strongest correlations. Dynamic graphs of these brain networks (obtained by dividing the time series into segments, and generating a brain network per segment) reflect how patients' brain behavior changes over

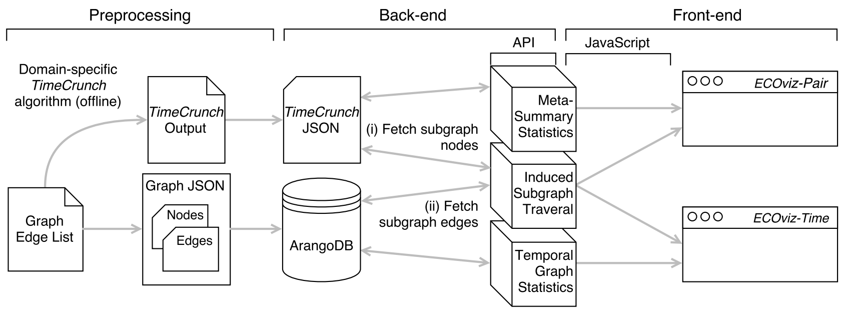


Figure 4: **End-to-end pipeline for our ECOVIZ visualization system.** Major components include offline preprocessing, ArangoDB & Flask API back-end, and web interface (JavaScript) front-end.

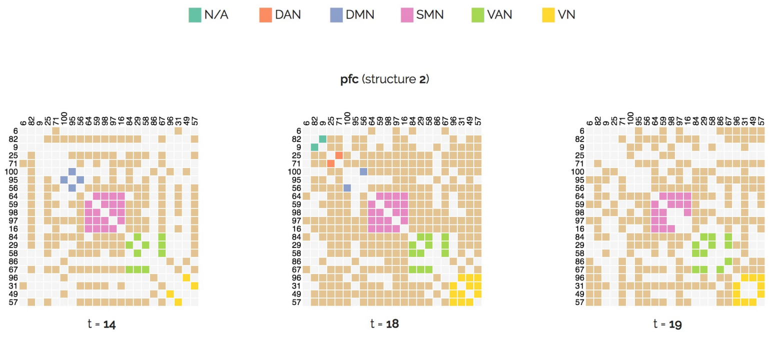


Figure 5: **ECOVIZ-TIME matrix sequence of a periodic full clique (pfc).** The colored resting-state modules show temporal patterns across time steps. Colors correspond to different voxel subnetworks: dorsal attention (DAN), default mode (DMN), sensorimotor (SMN), ventral attention (VAN), and primary visual (VN).

time. Furthermore, each voxel is associated with a subnetwork (e.g., Default Mode Network) reflecting a type of brain behavior (e.g., language, visual,sensorimotor).

ECOVIZ Overview: The first step in the ECOVIZ pipeline is to apply summarization to the input dynamic graph. In order to customize TIMECRUNCH for the connectomics application, we propose an application-specific candidate subgraph extraction routine: Instead of using the original SlashBurn clustering routine designed for real, large-scale graphs with power-law degree distribution, we use the set of all voxel nodes' egonets as candidate subgraphs. These egonets, or induced subgraphs of the central ego node and its neighbors, adjust the focus of TIMECRUNCH from high-degree hub nodes to labeled nodes from known brain regions. Note that ECOVIZ can just as well be used for visualization of large social graphs using the original candidate subgraph generation process. In addition to providing more connectomics-appropriate candidate subgraphs, these egonets also serve as natural communities suited for the small-worldness of brain networks [18].

Upon completing the TIMECRUNCH summarization process, ECOVIZ must interface the structure output format (node and time step participation per structure) with connectivity information in the underlying network. To do so, ECOVIZ receives a list of summary structures, and pairs each structural snapshot with connectivity data from the original network. This aggregation occurs in real time, and is the backbone of ECOVIZ's visualization component. For each temporal snapshot in each summary structure, the application (i) fetches participating node IDs from TIMECRUNCH results (stored in JSON), (ii) queries the graph database (ArangoDB) for an induced subgraph of the edges participating the structure, and (iii) makes asynchronous JavaScript requests to visualize each subgraph. This pipeline, as shown in Figure 4, is the source of two visualization views, ECOVIZ-PAIR and ECOVIZ-TIME, that support separate modes of data analysis.

The ECOVIZ-PAIR view allows end-users to compare pairs of summaries differing in data source (i.e., active and rest state brain behaviors) or preprocessing method (i.e., threshold value and time interval granularity used for the dynamic network generation) for the same subject or underlying phenomenon. Meta-summary charts are also displayed to reveal summary diversity, which may be used to indirectly evaluate graph construction quality. ECOVIZ-TIME, as seen in Figure 5, shows temporal snapshots of a summary structures via panels displaying visualized subgraphs or adjacency matrices over time. Nodes can optionally be colored, reflecting group membership or targeted interest (Figure 5 shows a coloring which reflects voxels' involvement in various types of brain behavior), which aids in detection of inter- and intra-community patterns over time.

7 Conclusion

In this work, we tackle the problem of identifying significant and structurally interpretable temporal patterns in large, dynamic graphs. Specifically, we formalize the problem of finding important and coherent temporal structures in a graph as *minimizing the encoding cost* of the graph from a compression standpoint. To this end, we propose TIMECRUNCH, a fast and effective, incremental technique for building interpretable summaries for dynamic graphs which involves *generating* candidate subgraphs from each static graph, *labeling* them using static identifiers, *stitching* them over multiple timesteps and *composing* a model using practical approaches. Finally, we apply TIMECRUNCH on several large, dynamic graphs and find numerous patterns and anomalies which indicate that real-world graphs *do* in fact exhibit temporal structure. We additionally demo our ECOVIZ framework which enables interactive and domain-specific dynamic network visualization on top of TIMECRUNCH.

References

[1] DBLP network dataset. `konect.uni-koblenz.de/networks/dblp_coauthor`, July 2014.

[2] C. C. Aggarwal and P. S. Yu. Online analysis of community evolution in data streams. In *SDM*, 2005.

[3] C. J. Alpert, A. B. Kahng, and S.-Z. Yao. Spectral partitioning with multiple eigenvectors. *Discrete Applied Mathematics*, 90(1):3–26, 1999.

[4] M. Araujo, S. Papadimitriou, S. Günnemann, C. Faloutsos, P. Basu, A. Swami, E. E. Papalexakis, and D. Koutra. Com2: Fast automatic discovery of temporal ("comet") communities. In *PAKDD*, pages 271–283. Springer, 2014.

[5] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[6] E. Bullmore and O. Sporns. Complex brain networks: Graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009.

[7] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *KDD*, pages 79–88. ACM, 2004.

[8] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *arXiv preprint cs/9402102*, 1994.

[9] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[10] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *Proc. 9th KDD*, pages 89–98, 2003.

[11] J. Ferlez, C. Faloutsos, J. Leskovec, D. Mladenic, and M. Grobelnik. Monitoring network evolution using MDL. *ICDE*, 2008.

[12] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal. Discovering frequent topological structures from graph datasets. In *KDD*, pages 606–611, 2005.

[13] U. Kang and C. Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *ICDM*, pages 300–309. IEEE, 2011.

[14] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[15] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins. The web as a graph: measurements, models, and methods. In *Computing and combinatorics*, pages 1–17. Springer, 1999.

[16] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos. Vog: Summarizing and understanding large graphs.

[17] B. Kulis and Y. Guan. Graclus - efficient graph clustering software for normalized cut and ratio association on undirected graphs, 2008. 2010.

[18] C.-T. Li and S.-D. Lin. Egocentric information abstraction for heterogeneous social networks. In *International Conference on Advances in Social Network Analysis and Mining*. IEEE, 2009.

[19] M. Li and P. M. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2009.

[20] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[21] E. E. Papalexakis, N. D. Sidiropoulos, and R. Bro. From k-means to higher-way co-clustering: Multilinear decomposition with sparse latent factors. *IEEE TSP*, 61(2):493–506, 2013.

[22] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *KDD*, pages 228–238, 2005.

[23] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[24] N. Shah, A. Beutel, B. Gallagher, and C. Faloutsos. Spotting suspicious link behavior with fbox: An adversarial perspective. In *ICDM*. 2014.

[25] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos. Timecrunch: Interpretable dynamic graph summarization. In *KDD*, pages 1055–1064. ACM, 2015.

[26] J. Shetty and J. Adibi. The enron email dataset database schema and brief statistical report. *Inf. sciences inst. TR, USC*, 4, 2004.

[27] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, pages 687–696. ACM, 2007.

[28] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In *KDD*, pages 965–973. ACM, 2011.

[29] K. S. Xu, M. Kliger, and A. O. Hero III. Tracking communities in dynamic social networks. In *SBP*, pages 219–226. Springer, 2011.

[30] Yike Liu and Tara Safavi and Abhilash Dighe and Danai Koutra Graph Summarization: A Survey In CoRR, abs/1612.04883, 2016.

[31] Saket Navlakha and Rajeev Rastogi and Nisheeth Shrivastava Graph Summarization with Bounded Error In SIGMOD, pages 419–432, 2008.

[32] Di Jin and Danai Koutra Exploratory Analysis of Graph Data by Leveraging Domain Knowledge In ICDM, 2017.

[33] Bijaya Adhikari and Yao Zhang and Aditya Bharadwaj and B. Aditya Prakash Condensing Temporal Networks using Propagation In SDM, pages 417–425, 2017.

[34] Nan Tang and Qing Chen and Prasenjit Mitra Graph Stream Summarization: From Big Bang to Big Crunch In SIGMOD, pages 1481–1496, 2016.

[35] Lei Shi and Hanghang Tong and Jie Tang and Chuang Lin VEGAS: Visual InfluEnce GrAph Summarization on Citation Networks In TKDE, pages 3417–3431, 2015.

[36] Yahoo! Webscope. `webscope.sandbox.yahoo.com`.

Billion-Node Graph Challenges

Yanghua Xiao, Bin Shao
Fudan University, Micorosoft Research Asia
shawyh@fudan.edu.cn, binshao@microsoft.com

Abstract

Graph is a universal model in big data era and finds its wide applications in a variety of real world tasks. The recent emergence of big graphs, especially those with billion nodes, poses great challenges for the effective management or mining of these big graphs. In general, a distributed computing paradigm is necessary to overcome the billion-node graph challenges. In this article, we elaborate the challenges in the management or mining on billon-node graphs in a distributed environment. We also proposed a set of general principles in the development of effective solutions to handle billion-node graphs according to our previous experiences to manage billion-node graphs. The article is closed with a brief discussion of open problems in billion-node graph management.

1 Introduction

Many large graphs have emerged in recent years. The most well known graph is the WWW, which now contains more than 50 billion web pages and more than one trillion unique URLs [5]. A recent snapshot of the friendship network of Facebook contains 800 million nodes and over 100 billion links [6]. LinkedData is also going through exponential growth, and it now consists of 31 billion RDF triples and 504 million RDF links [7]. In biology, the genome assembly problem has been converted into a problem of constructing, simplifying, and traversing the de Brujin graph of the read sequence [8]. Each vertex in the de Brujin graph represents a k-mer, and the entire graph in the worst case contains as many as 4^k vertices, where k generally is at least 20.

We are facing challenges at all levels from system infrastructures to programming models for managing and analyzing large graphs. We argue that a distributed memory system has the potential to meet both the memory and computation requirements for large graph processing [4]. The reasons are as follows. One distinguishing characteristic of graphs, as compared to other forms of data, is that data accesses on graphs have no locality: As we explore a graph, we invoke random, instead of sequential data accesses, no matter how the graph is stored. To address this problem, a simple solution is to put the graph in the main memory. The size of required memory space to host the topology of a web-scale graph is usually on the terabyte scale. It is still unlikely that it can be stored in the memory of a single machine. Thus, a distributed system is necessary. On the other hand, the computation power required by applications on web-scale graphs are often far beyond the capacity of a single machine. A distributed system is beneficial as graph analytics is often computation intensive. Clearly, both memory and computation demand an efficient distributed computing platform for large graphs.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In this paper, we advocate the use of distributed memory systems as a platform for online graph query processing and offline graph analytics. However, management of billion-node graphs in a distributed paradigm is still very challenging. In this article, we will first elaborate the challenges to manage or mine billion-node graphs in a distributed environment, especially for the distributed memory systems. We further propose a set of general principle to overcome the billion-node graph challenges according to our previous experience to develop scalable solutions on billion-node graphs.

2 Challenges

Even given an efficient distributed memory system, the management of billion-node graph is still challenging. There are two major challenges: *scalability* and *generality*. The two challenges are even more remarkable when the graphs are complicated. However, most real graphs are usually complicated and notoriously known as complex networks. Next, we will first elaborate the two challenges. Then we discuss the difficulties caused by the complicated structure of real graphs.

2.1 Billion-node Graph Challenges

Scalability Most current graph algorithms are designed for small, memory-based graphs. Most of memory based graph algorithms rely one subtile/tricky strategies to produce an optimal or near optimal solution. Consider the graph partitioning problem. A class of local refinement algorithms, most of which originated from the Kerninghan-Lin (KL) algorithm [9], bisect a graph into even size partitions. The KL algorithm incrementally swaps vertices among partitions of a bisection to reduce the edge-cut of the partitioning, until the partitioning reaches a local minimum. The local refinement algorithms are costly, and are designed for memory-based graphs only.

There are two general ideas to improve the scalability of the well-turned memory based algorithm for small graphs. The first is *multiple level* processing following the *coarsening-then-conquer* idea [12]. In *coarsening-then-conquer* idea, we repeatedly coarsen a graph until it is small enough then run corresponding heavyweight well-tuned algorithms on the small coarsened graphs. The second is leveraging parallelism. That is distributing the computation and data onto different machines and use multiple machines to achieve a speedup. However, both of two ideas face great challenges when processing billion-node graphs.

- First, consider the coarsening-then-conquer idea. We use graph partitioning as an example problem to discuss the challenge this idea is confronted with when handling billion-node graphs. Multi-level partitioning algorithms, such as METIS [11] coarsens a large graph by *maximal match* and apply algorithms such as KL and FM on the small graph. However, the assumption that (near) optimal partitions on coarsened graphs implies a good partitioning in the original graph may not be valid for real graphs, when maximal match is used as the underlying coarsening mechanism. Metis actually spent grate efforts refine the partitioning on coarsened graphs. In general, it is not trivial design an appropriate coarsening mechanisms to coarsen a graph without sacrificing useful information in the network.

- Second, let's investigate the challenge to use parallelism when processing billion-node graphs. Both data parallelism and computation parallelism is not trivial for graph data. To partition a graph into k machines to minimize the number of cut edges (i..e, communication cost) is a classical NP-hard problem. Thus, data parallelism is not trivial to be used. To distribute computation over different machines is neither easy. Because the logics of most graph operations or algorithms is inherently dependent, leading to poor computation parallelism. Many graph problems are proved to be P-Complete (i.e. hard to parallelize) problems, such as DFS (depth-first search).

Generality It is important to develop a general purpose infrastructure where graphs can be stored, served, and analyzed, especially for web-scale graphs. Current graph solutions are not built on top of a general-purpose graph infrastructure. Instead, they are designed exclusively and subtly for the purpose of specific tasks. Many solutions are sufficiently tuned for the specific problem settings and specific graph data. As a result, the assumptions, principles or strategies that achieve good performance on certain tasks or graphs are not necessarily effective on other problems or graphs. For example, for the graph partitioning problem, ParMetis [13] can work on graphs of tens of millions of nodes. However, it requires that the graph is partitioned *two-dimensionally*, that is, the adjacency list of a single vertex is divided and stored in multiple machines. This helps reduce the communication overhead when coarsening a graph. However, such a design may be disruptive to other graph algorithms. Even if we adopt this approach, the cost of converting data back and forth is often prohibitive for web-scale graphs.

Then, the question is, is any infrastructure currently available appropriate for web-scale graphs? MapReduce is an effective paradigm for large-scale data processing. However, MapReduce is not the best choice for graph applications [14, 22]. Besides the fact that it does not support online graph query processing, many graph algorithms for offline analytics cannot be expressed naturally and intuitively. Instead, they need a total rethinking in the "MapReduce language." For example, graph exploration, i.e., following links from one vertex to its neighbors, is implemented by MapReduce iterations. Each iteration requires large amount of disk space and network I/O, which is exacerbated by the random access pattern of graph algorithms. Furthermore, the irregular structures of the graph often lead to varying degrees of parallelism over the course of execution, and overall, parallelism is poorly exploited [14, 22]. Only algorithms such as PageRank, shortest path discovery that can be implemented in vertex-centric processing and run in a fixed number of iterations can achieve good efficiency. The Pregel system [15] introduced a vertex-centric framework to support such algorithms. However, graph partitioning is still a big challenge. We are not aware of any effective graph partitioning algorithms in MapReduce or even in the vertex-centric framework.

The generality issue deteriorates when the diversity of graph data and graph computations is taken into account. There are many kinds of graphs, such as regular graph, planar graphs, ER random graphs [31], small-world graphs, scale-free graphs [19]. Graph algorithms' performance may vary a lot on different types of graphs. On the other hand, there are a variety of graph computations such as path finding, subgraph matching, community detection, and graph partitioning. Each graph computation itself even deserves dedicated research; it is nearly impossible to design a system that can support all kinds of graph computations. There are also two completely different scenarios for graph computations: offline analytics and online query answering. They have different requirements on the performance and effectiveness of the graphs algorithms or solutions. All these diversity factors interleave with each other to make the situation even worse.

2.2 Complicated Structures of Real Big Graphs

Most real graphs are notoriously known as complex networks. Complex network means that the real graphs have a structure that is far away from the regularly simple structure that can be explained by simple network growth dynamics or modeled by simple models such as regular graphs, or ER networks. Next, we will discuss some of the complex structural characteristics of real graphs including scale free (power-law degree distribution) [19], small world *small-world* and *community structure*, as well as their influence on the billion-node graph management.

- *Scale free.* Most real graphs have a power law degree distribution, which is known as the scale free property of real graphs. The power law degree distribution implies that most vertices have small degrees, while some vertices of large degree do have a great opportunity to exist. These vertices of highest degree are usually referred to as *hubs*. There are two implications of power law degree distribution on big graph management. First, a distributed solution is easily trapped in load unbalance. If the hub vertices are not carefully taken care of, the machines hosting them tend to be the bottleneck of the system since hubs have more logical connections to other vertices and have a large probability to receive or send message from or to neighbors. The second implication is that the graph is more heterogeneous in terms of degree when compared to synthetic graph models such as ER model [31], where most vertices tend to have the average degree. The heterogeneous degrees make it difficult to design a universal strategy to process vertices of different degrees.

- *Small world.* Most real life networks are small-world networks, i.e., the average shortest distance is quite small. In most social networks, any two persons can reach each other within six steps. It is also shown that real life networks *shrink* when they grow in size, that is, their diameters become smaller [20]. As a result, the graph exploration in a big graph becomes extremely challenging since exploration of six steps or less will need to traverse almost the entire graph. The traverse of a billion-node graph usually costs serval hours. As a result, any online query answering that needs to explore the graphs on a billion-node graph might incur unaffordable cost.
- *Community structure.* Most real-life complex networks, including the Internet, social networks, and biological neural networks, contain community structures. That is, the networks can be partitioned into groups within which connections are dense and between which connections are sparse. The community structure makes the divide-and-conquer possible. However, partitioning a billion-node graph itself is non-trivial. And in many cases, the communities are not necessarily of equal size, which makes the balanced load distributions harder. On the other hand, real graphs usually tend to have an unclear community structure, that is the boundaries of communities are not clear and two communities in many real cases are hard to be separated. As a result, although the existence of community structure provides an opportunity, it is still challenging to leverage the community structure to develop effective solutions.

There are also many other properties of real graphs that pose great challenges to graph management, such as dynamically evolving, containing heterogeneous information. This article is only a limited discussion of these challenges. The complicated structures not only poses great challenge for us to manage big graph but also provide us new opportunities to build effective solutions to manage billion-node graphs if we are well aware of their existence and fully leverage these properties. Some recent approaches take into consideration the power-law degree distribution and community structure exhibited by many real life networks to develop effective distributed storage solutions [3] or reduce communication costs [2] in distributed graph processing.

3 Principles

Next, we present a set of general principles to overcome the billion-node graph challenges according to our previous experience to manage billion-node graphs. All these principles reflect our choice of different options in developing solutions. Most of our principles are intuitively correct. However, how to realize them in real problems and on big graphs is non-trivial. Clearly, it is only a limited set of effective principles. More effective principles are coming when we have more practices to process billion-node graphs. In the following text, we use detailed examples to showcase each principle. All the principles are discussed with respect to billion-node graph challenges.

3.1 Principle 1: Lightweight models are preferred

Billion-node scale demands lightweight models. In many real applications, the real requirement can be satisfied by problems models of different complexity. In general, when the graph is big, the models of less complexity is preferred. However, no free lunch in general. The lightweight models usually come with the cost of the sacrifice of effectiveness, which however is usually acceptable when processing billion-node graphs. For most heavyweight problems on graphs, there are usually corresponding lightweight models. For example, to answer the shortest distance query, distance oracle that uses a precomputed data structure to estimate the shortest distance approximately is usually more lightweight compared to the exact shortest distance query. To find a community around a vertex, the community search [1] is more lightweight than community mining. To find the landmarks of a graph, approximate betweenness is more lightweight than the exact betweenness. Next, we use shortest distance query as an example problem to elaborate this principle.

Distance oracle. Shortest distance queries are one of the fundamental queries on graphs. Exact shortest distance query answering on billion node graphs either by offline pre-computation or online computation are unaffordable. For the online computation, it takes hours for the Dijkstra algorithm (on weighted graph) or bread-first search (on unweighted graphs) to find the shortest distance between two nodes [25] in a billion-node graph. Alternatively, by offline computation we can pre-compute and store the shortest distances for all pairs of nodes, and use table lookups to answer shortest distance queries at the time of the query. Clearly, this approach requires quadratic space, which is prohibitive on large graphs.

Our experience shows that rather than exact shortest distance query, *distance oracle* is a more realistic model to answer shortest distance queries on billion node graphs. A distance oracle is a pre-computed data structure that enables us to find the (approximate) shortest distance between any two vertices in constant time. A distance oracle is feasible for billion node graphs if it satisfies the following criteria:

1. Its pre-computed data structure has a small *space complexity*. For billion-node graphs, we can only afford linear, or sub-linear pre-computed data structures.

2. Its construction has a small *time complexity*. Although distance oracles are created offline, for billion-node graphs, we cannot afford algorithms of quadratic time complexity.

3. It answers shortest distance queries in *constant time*.

4. It answers shortest distance queries with high *accuracy*.

We built a distance oracle by an embedding based approach which will be elaborated in the next principle. The result shows that our distance oracle is a realistic solution for the online shortest distance query on billion-node graphs. Our distance oracle takes about 20 to 70 hours to construct distance oracle on billion node graphs. Since it is built offline, this performance is acceptable. The query response is also quite efficient. It only takes less than 1 ms to answer the shortest distance query on a billion node graph. The accuracy results show that for geodesic distances less than 14 (which account for the majority of the shortest paths), our distance oracles generate distances with absolute error consistently less than 0.3 and relative error consistently less than 8%.

3.2 Principle 2: Approximate solutions are usually acceptable

For billion-node graph, we can not accept the algorithms or solutions with super-linear complexity. In general, only linear or near-linear (such as $O(N \log N)$) is acceptable for billion-node graphs. However, the efficiency usually comes with the sacrifice of accuracy. For billion-node graphs, minor sacrifice of accuracy usually is acceptable. Next, we will show how we use embedding based solution to answer shortest distance queries.

Embedding based distance oracle Graph embedding projects the elements of graphs (such as nodes or edges) into a geometric space so that the key properties of the original graph is preserved in the geometric space. There are a lot of graph embedding solutions due to the popularity of deep learning models which usually requires graphs as input. In this paper, we only consider the embeddings that is critical for big graph management.

Specially, we will discuss how to use embedding to build an efficient-yet-effective distance oracle. By this case study, we show that embedding is a promising transformation operation that can be used as an offline step to derive the essential information of the graph.

To build a distance oracle, we embed a graph into a geometric space so that shortest distances in the embedded geometric space preserve the shortest distances in the graph. Then, the shortest distance query can be answered by the coordinate distance. Let c be the dimensionality of the embedded space. The space complexity of the coordinate based solution is $\Theta(c \cdot |V|)$. The query can be answered in $\Theta(c)$ time, independent of the graph size.

Graph embedding is the key to the success of distance oracles of this kind. State-of-the-art approaches [26, 27] first select a set of landmark vertices using certain heuristics (for example, selecting vertices of large degrees). For each landmark, BFS is performed to calculate the shortest distance between each landmark and each vertex. Then, we fix the coordinates of these landmarks using certain global optimization algorithms based on the accurate distances between landmarks. Finally, for each non-landmark vertex, we compute its coordinates according to its distances to the landmarks. Finding the coordinates of landmarks and non-landmark vertices is formulated as an optimization problem with the objective of minimizing the sum of squared errors:

$$\sqrt{\sum_{(u,v)} (|c(u) - c(v)| - d(u,v))^2} \tag{1}$$

where $c(u)$ is the coordinates of vertex u. Usually, this kind of optimization problem can be solved by the *simplex downhill* method [28]. Given the coordinates of two vertices, their shortest distance can be directly estimated by the geometric distance between them. The distance calculation can be further improved by certain low-bound or upper-bound estimation.

Experimental results on real graphs show that embedding based distance oracle can find shortest distance for an arbitrarily given pair of nodes with the absolute error almost consistently less than 0.5, which is significantly better than other competitors [2].

3.3 Principle 3: Local information is usually helpful

A graph can be logically or physically organized. In either way, locality of the graph does matters in handling billion-node graphs. The subgraphs aground a certain vertex is a logically local graph. In distributed graph computing, the subgraphs induced by the vertices and their adjacent list in a certain machine is a physically local graph. Fully leveraging the locality of these graphs alleviates us from the complexity of the entire graph. By focusing on the local graphs, we can build effective and scalable solutions. To illustrate this principle, we will showcase how to use local graphs in local machines to estimate the betweenness of vertex.

Betweenness computation by Local graph based vertex Betweenness is one of the important measures of vertex in a graph. It is widely used as the criteria to find important nodes in a graph. Betweenness has been empirically shown to be the best measure to select landmarks for shortest distance query or shortest path query [29]. For a vertex v, its betweenness is the fraction of all shortest paths in the graph that pass through v. Formally, we have:

$$bc(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

where σ_{st} is the number of shortest paths between s and t, and $\sigma_{st}(v)$ is the number of shortest paths in σ_{st} that pass through v.

It is not difficult to see that the exact calculation of betweenness requires to enumerate all pairs of shortest path. The fastest algorithm needs $O(|V||E|)$ time to compute exact betweenness [30]. It is prohibitive for a billion-node graph. Thus an *approximate betweenness* is a more appropriate choice than the exact betweenness on big graphs. However, it is non-trivial to develop an effective-yet-efficient approximate betweenness estimation solution in a distributed environment.

We propose an efficient and effective solution to compute approximate betweenness for large graphs distributed across many machines. A graph is usually distributed over a set of machines by hashing on the vertex id. The hash function distribute a vertex as well as its adjacent list to a certain machine, which allows to build a local graph consisting of vertices and their relationship that can be found from the same machine. We show a local graph in Example 1. Instead of finding the shortest paths in the entire graph, we find the shortest paths in each machine. Then, we use the shortest paths in each machine to compute the betweenness of vertices on that machine. We call betweenness computed this way *local* betweenness. Clearly, the computation does not incur any network communication. After we find local betweenness for all vertices, we use a single round of communication to find the top-k vertices that have the highest local betweenness value, and we use these vertices as landmarks.

Theoretical results show that the shortest distance estimated from local graphs has a upper bound on the error of the exact shortest distance. This allows us to use the local shortest distance to approximate exact betweenness. Experimental results on real graphs shows that contrary to the perception that local betweenness is very inaccurate because each machine only contains a small portion of the entire graph, it turns out to be a surprisingly good alternative for the exact betweenness measure. Please refer to for the detailed results.

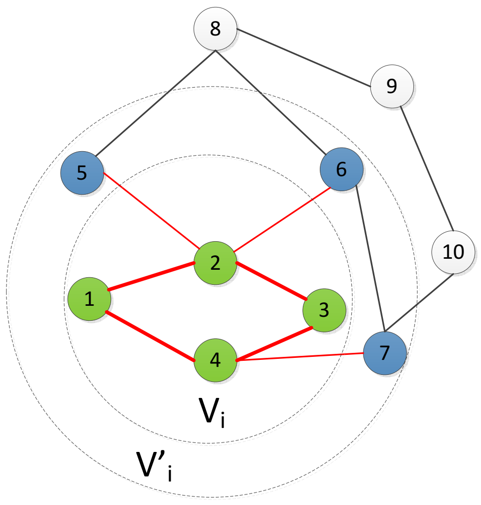


Figure 1: The local graph, the extended local graph, and the entire graph, from the perspective of machine i.

Example 1 (Local graphs): Consider a graph with 10 vertices, as shown in Figure 1. Assume machine i contains 4 vertices, that is, $V_i = \{1, 2, 3, 4\}$. The graph in the innermost circle is the *local graph*. Machine i also has information about vertices $5, 6$, and 7, as they are in the adjacency lists of vertex 2 and 4. The graph inside the second circle, except for the edge $(6, 7)$, is the *extended local graph*. Machine i does not realize the existence of vertices $8, 9$, and 10. Note that edge $(6, 7)$ does not belong to the extended local graph since none of its ends belongs to V_i.

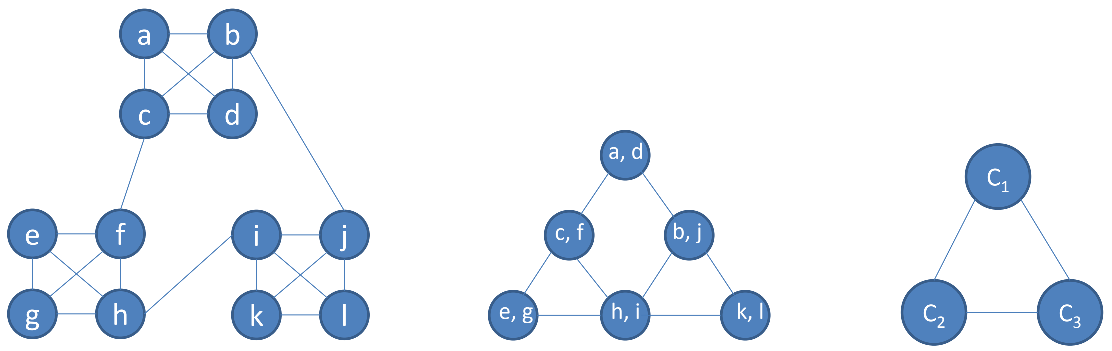
3.4 Principle 4: Fully leverage the properties of real graphs

We have shown that real big graphs usually exhibit a lot of structural characteristics. These characteristics on the one hand pose great challenge to build scalable solutions on graphs. On the other hand, if we are fully aware of these characteristics, we have a great opportunity to boost the performance of the solutions. Next, we show two cases in which we build effective solutions by carefully leveraging the properties of real graphs. The first is partitioning a billion graph by employing the community structure of real big graphs. The second is reducing the communication cost for a distributed bread-first-search by employing the power-law degree distribution of real graphs.

3.4.1 Graph partitioning based on community structure

As we have claimed that a distributed system is necessary to manage a billion-node graph. To deploy a graph on a distributed system, we need to first divide the graph into multiple partitions, and store each partition in one machine. How the graph is partitioned may cause significant impact on load balancing and communication. Graph partitioning problem thus is proposed to distribute a big graph onto different machines. Graph partitioning problem seeks solution to divide a graph into k parts with approximately identical size so that the edge cut size or the total communication volume is minimized in a distributed system. The problem of finding an optimal partition is NP-Complete [21]. As a result, many approximate solutions have been proposed [9, 10], such as KL [9] and FM [10]. However, these algorithms in general are only effective for small graphs. For a large graph, a widely adopted approach is to "coarsen" the graph until its size is small enough for KL or FM. The idea is known as multi-level graph partitioning, and a representative approach is METIS [11].

METIS uses maximal match to coarsen a graph. A maximal match is a maximal set of edges where no two edges share a common vertex. After a maximal match is found, METIS collapses the two ends of each edge into one node, and as a result, the graph is "coarsened.". Maximal match based coarsening however is unaware of the community structure of real graphs. Thus it is quite possible to break the community structure in the coarsened graphs. As a result, a maximal match may fail to serve as a good coarsening scheme in graph partitioning. For example, the coarsened graph shown in Figure 2(b) no longer contains the clear structure of the original graph shown in Figure 2(a). Thus, partitions on the coarsened graph cannot be optimal for the original graph. METIS use subtle refinement to compensate for the information loss due to coarsening, leading to extra cost.



(a) A graph (b) Coarsened by maximal match (c) Coarsened by LP

Figure 2: An example graph and its coarse-grained graph

A community-structure aware coarsening is proposed in the billion-graph partitioning solution MLP (multiple-level propagation). MLP uses *label propagation* (LP) [23, 24] to find the community of a graph and then coarsen a big graph. Compared to maximal match, LP is more semantic-aware and is able to find the inherent community structure of the graph, as illustrated in Figure 2(c). Thus, the coarsened graph preserves the community structure of the original graph. Consequently, the locally closed nodes tend to be partitioned into the same machine. MLP thus has no necessary for extra refinement, which saves time cost and makes MLP a good choice to partition a billion-node graph.

The experimental results on the synthetic graphs with embedded communities show that MLP can effectively leverage the community structure of graphs to generate a good partitioning with less memory and time [3]. In contrast, METIS, which is based on the maximal matching method, is not community-aware when it coarsens a graph, thus heavily relying on costly refinement in the uncoarsening phase to ensure the solution quality. As a result, METIS incurs more time and space costs.

3.4.2 Network IO reduction based on the power-law degree distribution

Bread-first search is one of the most important operations on graphs. Billion-node graph management calls for distribute BFS algorithms. *Level-synchronized BFS* is one of typical distributed algorithm to compute shortest distances in a distributed environment. Level-synchronized BFS proceeds level by level starting from a given vertex. At level l, each machine i finds vertices of distance l to r. Then, we locate their neighboring vertices. If a neighboring vertex has not been visited before, then it has distance $l+1$ to r. However, since the neighboring vertices are distributed across all machines, only their host machines (the machines they reside on) know their current distances to r. Thus, we ask their host machines to update the distance of these neighboring vertices. Each remote machine i, upon receiving messages from all of other machines, updates their distances to either $l+1$ if they have not been visited before, or keeps their current distances (equal to or less than l) unchanged. After each level, all the machines synchronize, to ensure that vertices are visited level by level.

In the above naive BFS algorithm, the cost is dominated by sending messages to remote machines to update vertices' distances to landmarks. We reduce the communication cost by caching a subset of vertices on each machine. The opportunity comes when we notice that many distance update requests in level-synchronized BFS actually are wasteful. Specifically, a distance update request for vertex u is wasteful if the shortest distance of u to the starting node is already known. Avoiding such wasteful requests can reduce the communication cost. A straightforward way to do this is to cache each vertex's current distance on each machine. Then, we only need to send messages for those vertices whose current distances to r are ∞. In this manner, we can reduce the communication cost. However, for billion node graphs, it is impossible to cache every vertex on every machine. Then, the problem is: *which vertices to cache?*

Intuitively, the number of remote messages we saved by caching vertex v is closely related to the degree of v. The larger the degree of v, the more messages can be saved. Since real networks are usually scale free, that is, only a small fraction of vertices have large degree. This allows us to save a significant number of network messages by caching the top-K vertices of highest degree. We use an empirical study [2] to obtain a more intuitive understanding about the effectiveness to save network IOs by caching the hub vertices. The study uses the scale-free graphs [32] whose degree follows the distribution as follows:

$$deg(v) \propto r(v)^R \tag{3}$$

where $r(v)$ is the degree rank of vertex v, i.e., v is the $r(v)$-th highest-degree vertex in the graph, and $R < 0$ is the rank exponent.

The simulation results are given in Figure 3 with R varying from -0.6 to -0.9 (many real networks' rank exponent is in this range) [2]. From the simulation, we can see that by caching a small number of hub vertices, a significant number of communications can be saved. For example, when $R = -0.9$, caching 20% of the top-degree vertices can reduce communications by 80%.

4 Conclusion

In this paper, we elaborate the challenges to mange big graphs of billion nodes in the distributed memory system. We propose a set of general principles to develop efficient algorithmic solutions to manage billion-node graphs based on our previous experience to process billion-node graphs. Although these principles are effective, we

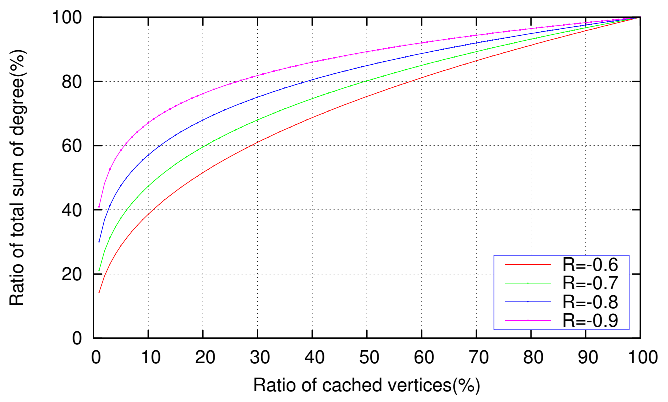


Figure 3: Benefits of caching hub vertices on scale free graphs

still have many open problems in billion-node graph management. The most challenging open problem might be the design of a universal solution to manage big graphs. Currently, we can only develop efficient solutions that are well tuned for specific tasks or specific graphs. In general, we still have a long way to overcome the billion-node graph challenges.

Acknowledgement This paper was partially supported by National Key Basic Research Program of China under No.2015CB358800, by the National Natural Science Foundation with No.61472085, U1509213, by Shanghai Municipal Science and Technology Commission foundation key project under No.15JC1400900, by Shanghai Municipal Science and Technology project under No.16511102102, No.16JC1420401.

References

[1] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 277–288.

[2] Z. Qi, Y. Xiao, B. Shao, and H. Wang, "Toward a distance oracle for billion-node graphs," *Proc. VLDB Endow.*, vol. 7, no. 1, pp. 61–72, Sep. 2013.

[3] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, Eds. IEEE, 2014, pp. 568–579.

[4] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516.

[5] http://www.worldwidewebsize.com/.

[6] http://www.facebook.com/press/info.php?statistics.

[7] http://www.w3.org/.

[8] D. R. Zerbino et al., "Velvet: algorithms for de novo short read assembly using de bruijn graphs." *Genome Research*, vol. 18, no. 5, pp. 821–9, 2008.

[9] B. Kernighan et al., “An efficient heuristic procedure for partitioning graphs,” *Bell Systems Journal*, vol. 49, pp. 291–307, 1972.

[10] C. M. Fiduccia et al., “A linear-time heuristic for improving network partitions,” in *DAC ’82*.

[11] G. Karypis et al., “Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0,” Tech. Rep., 1995.

[12] G. Karypis et al., “Analysis of multilevel graph partitioning,” in *Supercomputing ’95*.

[13] G. Karypis et al., “A parallel algorithm for multilevel graph partitioning and sparse matrix ordering,” *J. Parallel Distrib. Comput.*, vol. 48, pp. 71–95, 1998.

[14] A. Lumsdaine et al., “Challenges in parallel graph processing,” *Parallel Processing Letters*, pp. 5–20, 2007.

[15] G. Malewicz et al., “Pregel: a system for large-scale graph processing,” in *SIGMOD ’10*.

[16] A. Abou-Rjeili et al., “Multilevel algorithms for partitioning power-law graphs,” in *IPDPS ’06*.

[17] B.Shao et al., ”The trinity graph engine,” *Microsoft Technique Report, MSR-TR-2012-30*.

[18] P. Erdős et al., “Graphs with prescribed degrees of vertices (hungarian).” *Mat. Lapok*, vol. 11, pp. 264–274, 1960.

[19] A.-L. Barabasi et al., “Emergence of scaling in random networks,” vol. 286, pp. 509–512, 1999.

[20] J. Leskovec et al., “Graphs over time: Densification laws, shrinking diameters and possible explanations,” in *KDD ’05*.

[21] M. R. Garey et al., “Some simplified np-complete problems,” in *STOC ’74*.

[22] K. Munagala et al., “I/O-complexity of graph algorithms,” in *SODA ’99*.

[23] M. J. Barber et al., “Detecting network communities by propagating labels under constraints,” *Phys.Rev.E*, vol. 80, p. 026129, 2009.

[24] X. Liu et al., “How does label propagation algorithm work in bipartite networks?” in *WI-IAT ’09*.

[25] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy, “A sketch-based distance oracle for web-scale graphs,” in *WSDM ’10*.

[26] X. Zhao, A. Sala, H. Zheng, and B. Y. Zhao, “Fast and scalable analysis of massive social graphs,” *CoRR*, 2011.

[27] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao, “Orion: shortest path estimation for large social graphs,” in *WOSN’10*.

[28] J. A. Nelder and R. Mead, “A simplex method for function minimization,” *Computer Journal*, 1965.

[29] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, “Fast shortest path distance estimation in large networks,” in *CIKM ’09*.

[30] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of Mathematical Sociology*, 2001.

[31] aErdős, P. and Gallai, T., “Graphs with prescribed degrees of vertices (Hungarian).,” *Mat. Lapok*,1960.

[32] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the internet topology,” *SIGCOMM Comput. Commun. Rev.*, 1999.

Mining Social Streams: Models and Applications

Karthik Subbian, Charu C. Aggarwal
University of Minnesota, IBM T. J. Watson Research Center
karthik@cs.umn.edu, charu@us.ibm.com

Abstract

Online social networks are rich in terms of structural connections between entities and the content propagated between them. When this data is available as a continuous stream of interactions on the social graph, it is referred to as a social stream. *Social streams have several challenges: (1) size of the underlying graph, (2) volume of the interactions, and (3) heterogeneity of the content and type of interactions between the entities. Mining social streams incorporates discovering patterns and trends using* both structure and interaction data.

In this work, we discuss two important social stream applications: (1) event detection and (2) influence analysis. Event detection is seen as a social stream clustering problem that can be either supervised and unsupervised, depending on the availability of labeled data. While influence analysis is an unsupervised problem modeled in a query-based framework. We discuss the key characteristics of these two problems, their computational difficulties and efficient modeling techniques to address them. Finally, we highlight several challenges and future opportunities for research in this area.

1 Introduction

The prominence of social network and the rise of cloud- and web-based applications, in the last decade, has enabled greater availability of streaming social data. The data available in the stream could be a continuous stream of edges as in the case of new friendships in Facebook or streaming interactions between users such as stream of tweets in Twitter. There are several challenges in mining and discovering patterns in streaming social data. Some of them are (a) processing the data in single pass (i.e., one-pass constraint), (b) continuous maintenance of the model, and (c) its efficient representation in memory through hashing, sampling, or latent factors. Depending on the availability of supervised knowledge, the learned model can be either supervised or unsupervised. characteristics of such social streams and their applications.

Social streams consist of *content-based interactions between structurally connected entities* in the data. Let us assume that the structure of the social network is denoted by the graph $G = (N, A)$. The node set is denoted by N and edge set is denoted by A. The concept of a social stream is overlaid on this *social network structure* G.

Definition 1 (Social Stream): A social stream is a continuous and temporal sequence of objects $S_1 \dots S_r \dots$, such that each object S_i is a tuple of form (q_i, R_i, T_i) where,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- $q_i \in N$ is the sender of the message T_i to other nodes.
- T_i corresponds to the content of the interaction.
- $R_i \subseteq N$ is a *set of one or more* receiver nodes, which correspond to all recipients of the message T_i from node q_i. It is assumed that each edge (q_i, r) belongs to the set A, for all $r \in R_i$.

One can also view a social stream as an enriched version of a graph stream, which does not have associated content [1].

1.1 Social Stream Applications

There are numerous applications that are popular in the context of social streams. We discuss some of them below and with their key characteristics.

- **Event Detection** The problem of event detection [3, 22] is to identify the most important set of keywords that describe a trending event. Such trending events often occur in temporal bursts in specific network localities. This problem is typically unsupervised where the new events are detected with no prior knowledge. In some cyber-security applications, such as detecting terrorist attacks or civilian unrest, previous occurrences of such instance can be used to supervise and detect their repeating occurrence.
- **Influence Analysis** The diffusion of information in a social network via re-sharing is referred to as cascades. Identifying influential users [24] in these information cascades in social streams is a challenging problem. The streaming nature of the data makes it harder to track the diffusion of information across longer paths in the network when there are numerous messages propagated. Moreover, to be able to query and understand the influential users in a time-sensitive manner requires efficient data structures and their maintenance.
- **Link Prediction** Discovering links in streaming graph data can be important in a variety of applications where the graph is changing rapidly. For instance camera-first applications, like Snapchat or Instagram, where an user visits the application several times a day, recommending the right group or user to follow is critical to maintain a low churn rate. In such scenarios computing centrality measures, such as Jaccard coefficient, common neighbors, and Adar-Adamic in streaming graphs can be computationally hard. There are some recent approaches [30] that tackle this problem using cost-effective graph sketches based on hashing and sampling techniques.

In this paper, we will discuss the first two applications: (1) Event detection and (2) Influence Analysis. For event detection [3], we discuss an online clustering algorithm which tracks both content and network structure efficiently using hashing schemes. In the case of influence analysis [26], we consider a more flexible querying model to query influence scores either using the set of influencers, set of users influenced, the context or time. Event detection is discussed in both supervised and unsupervised settings. While influence analysis is an unsupervised problem modeled in a query-based framework.

2 Event Detection

The problem of event detection is closely related to that of *topic detection and tracking* [5, 4, 7, 29]. This problem is also closely related to *stream clustering*, and attempts to determine new *topical trends in the text stream* and their significant evolution. The idea is that important and newsworthy events in real life are often captured in the form of *temporal bursts of closely related messages in a network locality*. Clearly, messages which are sent between a tightly knit group of actors may be more indicative of a particular event of social interest, than a set of messages which are more diffusely related from a structural point of view. At the same time, the *content and topics* of the documents should also play a strong role in the event detection process. Thus, both network locality and content of interaction need to be leveraged in a *dynamic streaming scenario* for the event detection process [3].

2.1 Social Stream Clustering

We begin with describing an unsupervised technique for event detection. This approach continuously characterizes the incoming interactions in the form of clusters, and leverages them in order to report events in the social stream. Formally, the problem of social stream clustering can be defined as follows:

Definition 2 (Social Stream Clustering): A social stream $S_1 \dots S_r \dots$ is continuously partitioned into k *current* clusters $\mathcal{C}_1 \dots \mathcal{C}_k$, such that:

- Each object S_i belongs to at most one of the current clusters $\mathcal{C}_r$.
- The objects are assigned to the different clusters with the use of a similarity function which captures both the content of the interchanged messages, and the dynamic social network structure implied by the different messages.

As the clusters are created dynamically, they may change considerably over time with the addition of new points from an evolving stream. Furthermore, in some cases, an incoming object may be significantly different from the current clusters. In that case, it may be put into a cluster of its own, and one of the current clusters may be removed from the set $\mathcal{C}_1 \dots \mathcal{C}_k$. Such an event may be an interesting one, especially if the newly created cluster starts a new pattern of activity in which more stream objects are subsequently added. Therefore, there are two kinds of events *novel* and *evolutionary events*.

The arrival of a data point S_i is said to be a *novel event* if it is placed as a single point within a newly created cluster $\mathcal{C}_i$. It is often possible for previously existing clusters to match closely with a sudden burst of objects related to a particular topic. This sudden burst is characterized by a change in fractional presence of data points in clusters. Let $F(t_1, t_2, \mathcal{C}_i)$ be the fractional cluster presence of objects arrived during time period (t_1, t_2) which belong to cluster $\mathcal{C}_i$ normalized by the total number of objects in cluster $\mathcal{C}_i$.

In order to determine *evolutionary events*, we determine the higher rate at which data points points have arrived to this cluster in the previous time window of length H, as compared to the event before it. A parameter α is used in order to measure this evolution rate and $t(\mathcal{C}_i)$ is the creation time for cluster $\mathcal{C}_i$. This condition of evolution rate is formally defined in Eqn. 1. Here it is assumes that the value of $t_c - 2 \cdot H$ is larger than the cluster creation time $t(\mathcal{C}_i)$ in order to define the evolution rate in a stable way.

$$\frac{F(t_c - H, t_c, \mathcal{C}_i)}{F(t(\mathcal{C}_i), t_c - H, \mathcal{C}_i)} \geq \alpha \qquad (1)$$

2.2 Online Model Maintenance

The design of an effective *online clustering* algorithm is the key to event detection. There are two main components that decide the efficiency of online clustering in social streams: (a) representation of clusters and (b) efficient similarity computation using text and structural content.

In order to detect a novel or an evolutionary event it is critical to decide which cluster to assign the incoming social stream object. Hence, the similarity score computed between the incoming object and the clusters, using structure and content information, plays a crucial role. The structure and content of the *incoming object* is usually smaller and easier to represent in-memory. However, the cluster information is extremely large to fit in-memory, as the cluster may represent all the incoming social stream objects until that time. Hence, we need an efficient summarization of clusters. A typical way to summarize the clusters is using bit vectors or normalized frequency counters [3, 2]. However, when the size of the vectors become extremely large it needs to be compressed to fit in-memory for tracking the social stream clusters. In the following we discuss one such efficient summarization and hash-based similarity computation approach for online model maintenance

2.2.1 Cluster Summarization

Each of the social stream clusters for event detection must maintain both text and network information in their summaries. The cluster-summary $\psi_i(\mathcal{C}_i)$ of cluster $\mathcal{C}_i$ is defined as follows:

- It contains the node-summary, which is a set of nodes $V_i = \{j_{i1}, j_{i2} \dots j_{is_i}\}$ together with their frequencies $\eta_i = \nu_{i1} \dots \nu_{is_i}$. The node set V_i is assumed to contain s_i nodes.
- It contains the content-summary, which is a set of word identifiers $W_i = \{l_{i1}, l_{i2}, \dots l_{iu_i}\}$ together with their corresponding word frequencies $\Phi_i = \phi_{i1}, \phi_{i2} \dots \phi_{iu_i}$. The content-summary W_i is assumed to contain u_i words.

The overall summary is $\psi_i(\mathcal{C}_i) = (V_i, \eta_i, W_i, \Phi_i)$.

2.2.2 Efficient Similarity Computation

The size of word frequencies $|\Phi_i|$ is generally smaller compared to $|\eta_i|$. This is because the size of vocabulary (in 100 thousands) is often much smaller than the number of users in the social network (usually in billions). Hence, it requires efficient computation of similarity for the structural part. Note that the content-based similarity computation $SimC(S_i, C_r)$ is straightforward, and is simply the tf-idf based [21] similarity between the content T_i and W_r.

The structural similarity between the nodes V_r and the nodes $R_i \cup \{q_i\}$ in the social stream is computed as follows. Let $B(S_i) = (b_1, b_2, \dots b_{s_r})$ be the bit-vector representation of $R_i \cup \{q_i\}$, which has a bit for each node in V_r, and in the same order as the frequency vector $\eta = (\nu_{r1}, \nu_{r2}, \dots \nu_{rs_r})$ of V_r. The bit value is 1, if the corresponding node is included in $R_i \cup \{q_i\}$ and otherwise it is 0. The structural similarity between the object S_i and the frequency-weighted node set of cluster $\mathcal{C}_r$ is defined as follows:

$$SimS(S_i, \mathcal{C}_r) = \frac{\sum_{t=1}^{s_r} b_t \cdot \nu_{rt}}{\sqrt{||R_i \cup \{q_i\}||} \cdot (\sum_{t=1}^{s_r} \nu_{rt})} \tag{2}$$

Note the use of L_1-norm for the node-frequency vector (as opposed to L_2-norm) in the denominator, in order to to penalize the creation of clusters which are too large. This will result in more balanced clusters. Note that the incoming node set contains both sender and the receivers.

The overall similarity $Sim(S_i, \mathcal{C}_r)$ can be computed as a linear combination of the structural and content-based similarity values.

$$Sim(S_i, \mathcal{C}_r) = \lambda \cdot SimS(S_i, \mathcal{C}_r) + (1 - \lambda) \cdot SimC(S_i, \mathcal{C}_r) \tag{3}$$

The parameter λ is the balancing parameter, and lies in the range $(0, 1)$. This parameter is usually specified by the user.

The numerator of Eqn. 2 is harder to compute as maintaining the vector η_r is expensive. One approach is to compress the size of η_r and estimate the numerator using count min-hash technique [10]. Consider w pairwise independent hash function, each of which resulting in a hash table of size 0 to $h - 1$. Whenever a node is encountered in $q_i \cup R_i$ it is hashed using all w hash functions and the corresponding counts in each of the hash tables are incremented. Since there are collisions in hash table there may be an over-estimate of the exact count of nodes in each hashtable. To upper-bound this over-estimate one can use the minimum value across w hash tables. The numerator of Eqn. 2 can be thus obtained using the sum of minimum counts of hashed incoming nodes in the social stream object. Let this estimated structural similarity be denoted as $EstSimS(S_i, \mathcal{C}_r)$. In Lemma 3 the upper bound of this estimated similarity is shown. We ask the readers to refer to [3] for details of this upper bound proof and [10] for min-hash count technique.

Lemma 3: If a sketch-table with length h and width w is used, then for some small value $\epsilon > \sqrt{|R_i| + 1}/h$, the estimated value of the similarity $EstSimS(S_i, \mathcal{C}_r)$ is bounded to the following range with probability at least $1 - \left(\frac{\sqrt{|R_i|+1}}{h \cdot \epsilon}\right)^w$:

$$SimS(S_i, \mathcal{C}_r) \leq EstSimS(S_i, \mathcal{C}_r) \leq SimS(S_i, \mathcal{C}_r) + \epsilon. \tag{4}$$

The similarity of the social stream object to the assigned cluster is often maintained as a distribution. If the similarity of the newly arrived object is β standard deviations away from the mean, then the object is assigned to its own cluster, resulting in a novel event (See Section 2.1). When β is too small it results in highly unstable clusters and when too large it leads to stale clusters. Note that one must maintain the zeroth, first and second order moments M_0, M_1 and M_2 of the closest similarity values continuously to compute the mean (μ) and standard deviation (σ). These values can be easily maintained in the streaming scenario, because they can be *additively* maintained over the social stream. The mean and standard deviation can be expressed in terms of these moments as follows:

$$\mu = M_1/M_0, \quad \sigma = \sqrt{M_2/M_0 - (M_1/M_0)^2}.$$

2.3 Supervised Event Detection

In several cyber-security applications users look for suspicious events, based on historical occurrences of these instances. This is the case of *supervised event detection*. Here, we assume that we have access to the past history of the stream in which the event $\mathcal{E}$ has been known to have occurred. The *event signature* of a social stream is a k-dimensional vector $V(\mathcal{E})$ containing the (average) relative distribution of event-specific stream objects to clusters. Here k is the number of clusters in the model. Clearly, the event signature provides a useful characterization of the relative topical distribution during an event of significance.

During a period of mideast unrest (the event $\mathcal{E}$), some clusters are likely to be much more active than others, and this can be captured in the vector $V(\mathcal{E})$, as long as ground truth is available to do so. The event signatures can be compared to *horizon signatures*, which are essentially defined in the same way as the event signatures, except that they are defined over the more recent time horizon $(t_c - H, t_c)$ of length H. One can compute the dot product similarity of the horizon signature to the event signature and raise an alarm if its value is above a certain threshold. The tradeoff between false positives and false negatives is determined by the threshold.

3 Influence Analysis

The problem of finding influential actors is important in various domains such as viral marketing [6, 27] and political campaigns. The problem was formally defined by Kempe et al. [16] as an optimization problem over all possible subsets of nodes with cardinality k. Subsequently, a significant amount of work [16, 18, 9, 8, 14, 13] has been done on this area. All these approaches are static in the sense that they work with a fixed model of network structure and edge probabilities. In practice, however, the influence of actors are defined by how their messages are propagated in the social network over time. Such propagation can only be observed from the underlying *social stream*, such as a Twitter feed or the sequence of Facebook activities. Since the problem is dynamic in nature, using the actual flow of information is becoming more popular [25].

A major disadvantage of existing influence analysis methods is that they are not able to *query* the influencers in a *context-specific* fashion. Ideally, one would like to be able to use search terms to determine influencers that are specific to a given context. For example, the top influencers for the search term "*Egypt Unrest*" would be very different from that of the search term "*PGA Golf Tour*". The inflexibility of existing methods is, in part, because the existing methods [16, 18, 9, 8, 14] decouple the problem of influence analysis from learning content-centric influence probabilities [12].

The influencers in a social stream are *time-sensitive* and may rapidly evolve [1], as different external events may lead to changes in the influence patterns over time. For instance, a query such as "*winter boots*" may generally have prominent entities associated with shoe stores as the most influential entities, but an (advertisement) statement from a popular figure in the entertainment industry, such as Justin Bieber, on a specific boot style may change this ordering. Important events can often dynamically change the influencers, and this can only be tracked in a *time-sensitive* and *online fashion* from the underlying activities in the social stream.

3.1 Influence Querying in Social Streams

Influence function is usually composed of four important components: (a) set of influencers, (b) set of users being influenced, (c) context of influence (or keywords), and (d) time of influence. One can compute the influence score as a function of these parameters and use it to query influencers in context-sensitive and time-sensitive manner using social streams.

Let the influence function $\mathcal{I}(S_1, S_2, \mathcal{Q}, t)$ represents the aggregate influence score of actor set S_1 on actor set S_2 with respect to content $\mathcal{Q}$ at time t. Most of the queries are resolved by evaluating this score and then ranking it over various possibilities in the argument. One or more of the arguments in $\mathcal{I}(S_1, S_2, \mathcal{Q}, t)$ can be instantiated to a "*" (don't care) in order to enable more general queries in which all possibilities for the matching are considered. For instance, the queries $\mathcal{I}(\text{“David”}, *, \text{“Egypt unrest”}, t)$ and $\mathcal{I}(\text{“John”}, *, \text{“Egypt unrest”}, t)$ can be used to compare the total influence of David and John on the topic "*Egypt unrest*" at time t. Some examples of useful queries that can be resolved with this approach are as follows:

- For a given query context $\mathcal{Q}$, determining the top-k *influencers* at time t can be formulated as:

$$\max_{X:|X|=k} \mathcal{I}(X, *, \mathcal{Q}, t).$$

  It is also possible to constrain the query to consider a specific subset Y in the second argument, corresponding to the influenced actors. For example, a winter clothing merchant might decide to consider only those actors whose location profiles correspond to colder regions.

- Determining the top-k *influenced* actors at time t, for a given query context $\mathcal{Q}$, can be extremely useful in context-specific subscriptions and in recommending interesting public content to influenced users.

- Influence queries can also be useful in context-sensitive link recommendation, such as finding the top-k influencer-influenced pairs, for a given query context $\mathcal{Q}$.

3.2 Information Flow Based Querying

The context information used in the influence function for a query can be a set of hashtags, tokens or keywords. Such keywords propagated via nodes (or actors) in a social network is considered as an *information flow path*. A flow path must also satisfy a valid path in the network structure. For instance, if there is information reshared from a_1 to a_2 to a_3 and there is no network edge between a_2 to a_3. Then, the flow is only valid until a_1 to a_2. Hence, the valid flow path would be $\mathcal{P} = \langle a_1, a_2 \rangle$ (not $\langle a_1, a_2, a3 \rangle$).

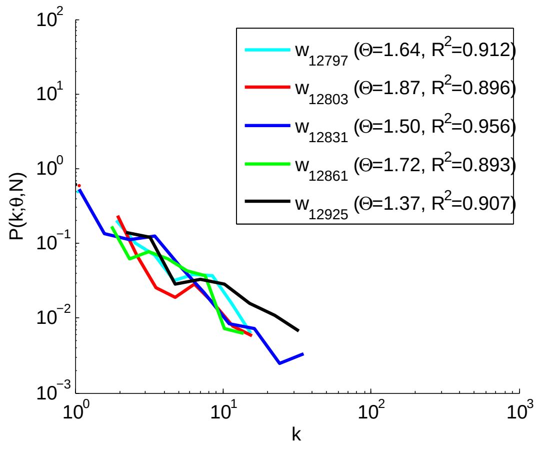


Figure 1: The decayed flow weights in the flow-path trees follow a Zipf distribution. The best-fit estimate of the Zipf parameter θ and corresponding R^2 for few flow-path trees are shown in the legend.

Time-sensitive influence can be computed using an half-life function or exponentially decaying function $exp(-\lambda\delta t)$. Note that the decay rate (λ) is application specific as some social networks may have faster turn around time and hence information propagates quickly. In such cases λ can be set to a higher value to reduce the amount of influence effect. The time difference between the sender (t_o) and the final receiver (t_c) of the message in the flow path $\mathcal{P}$ is denoted by δt. Then the influence score for one flow path $\mathcal{P}$ for carrying a single keyword K_i at time t_c is given by $exp(-\lambda * (t_c - t_o))$ where $\delta t = t_c - t_o$.

There can be many paths through which many keywords may propagate between a pair of actors. Hence, one can accumulate all of that influence between actors a_j and a_k for keywords K_i at time t_c as $\mathcal{V}(a_j, a_k, K_i, t_c)$. When there are several keywords in the query Q, one can compute the atomic influence as the aggregate pairwise influence across all keywords in the query Q. Formally, it is defined as follows:

Definition 4 (Influence Function): The atomic influence function $\mathcal{I}(a_j, a_k, \mathcal{Q}, t)$ for a node a_j to influence a_k, is defined as the sum of the aggregate pairwise flows over all keywords $K_i \in \mathcal{Q}$ in the data: $\mathcal{I}(a_j, a_k, \mathcal{Q}, t) = \sum_{K_i \in \mathcal{Q}} \mathcal{V}(a_j, a_k, K_i, t)$.

3.3 Efficient Tracking of Influencers

In order for one to compute the influence function $I(a_i, a_j, K_i, t_c)$ we need to know all the information flow paths between a_i and a_j, for single keyword K_i, until time t_c. A simple tree based data structure is proposed in [26] called *Flow Path Tree*. The notion of this tree data structure is to have one tree for each keyword and as the social stream objects are encountered the paths are back tracked and the tree data structure is updated. The back tracking may seem exponential in nature, particularly due to the power law degree distribution of the social graphs. However, the keyword K_i is not propagated by all nodes and hence back tracking in practice is much cheaper from a computational perspective.

The main disadvantage of tracking the flow paths using the tree is the size of the tree. The tree grows at an exponentially faster rate as the volume of the stream increases. So, we need an efficient way to maintain the in-memory representation of the flow path tree. The influence weights computed in a single flow path tree for a single keyword K_i using exponentially decaying functions generally follows a skewed Zipf distribution [26]. This is shown in Fig. 1 using a log-log plot. This observation implies that most of the tree weight is generated from a very few important flow paths in the tree. Using this observation, one can trim down the tree by a fraction of $1 - \alpha$, where α is the fraction of nodes retained in the tree after trimming. Also, only leaves of the tree are trimmed when the number of nodes in the tree reaches a maximum threshold, say N. Then the fraction of weights that remain in the tree can be lower-bounded using Theorem 5. We request the readers to refer to [26] for the proof of Theorem 5.

Theorem 5: Let the flow weights on the N nodes in the tree be distributed according to the Zipf distribution $1/i^\theta$ for $\theta \geq 1$. Then, the total fraction $F(N, \alpha)$ of the flow in the top $n = \alpha \cdot N$ nodes of the tree is at least equal to:

$$F(N, \alpha) \geq \log(n)/\log(N) = 1 - \log(1/\alpha)/\log(N). \tag{5}$$

The skew helps significantly in retaining the vast majority of the heavy flows. For example, in a flow path tree with $100,000$ total flow weight, discarding half the nodes ($\alpha = 0.5$) would result in total flow weight reduction of only $1 - \log(50000)/\log(100000) = 0.06$. Thus, the vast majority of the heavy flows (i.e. 94%) are retained, which are the ones most relevant for the influence analysis process. This suggests that the pruning process can be used to significantly reduce the complexity of the flow-path tree, while retaining most of the information needed for the influence analysis task.

3.4 Relationship with Katz Measure

The atomic influence function is closely related to the Katz measure [20]. The Katz measure is defined in terms of the weighted sum of the number of all possible paths between a pair of nodes and the weight decays exponentially with the length of the underlying path. Specifically, if $\mathcal{P}_{ij}$ be the set of paths between nodes a_i and a_j, then the Katz measure $K(a_i, a_j)$ is defined as follows:

$$K(a_i, a_j) = \sum_{P \in \mathcal{P}_{ij}} \gamma^{|P|} \tag{6}$$

Here γ is the discount factor on the path length, which is analogous to the flow-based temporal decay factor. Thus, flow-based approach computes exponentially decayed flow weights across different paths, as a more dynamic, time- and content-sensitive way of measuring the importance of nodes. In an indirect sense, this way of computing node importance can be considered a flow-based analogue to the Katz measure in static networks. Because the Katz measure has been shown to be effective for link recommendation in static networks [20], it lends greater credence to its use in the flow-based streaming scenario. Of course, the Katz measure is used rarely in static networks because of the complexity of enumerating over a large number of possible paths. The important point to understand is that the flow-based measure *significantly* changes in the importance of different paths in the update process, and can also be more efficiently computed in the streaming scenario, such as using the pruning technique discussed in Section 3.3.

4 Future Work and Conclusions

The area of social stream research lies in the intersection of big data, graph mining, and social network analysis. One of the important characteristics of social streams is the availability of high velocity streaming data that includes both structural and content information. Moreover, the sheer volume and the heterogeneity of the underlying social interactions makes the problem much more challenging. For example, a social network could have billions of nodes, trillions of edges, trillion interactions per day and a wide variety of such interactions (e.g. like, share, and comment).

There are numerous social network algorithms that are yet to be developed for various social streaming applications. Community detection falls in the realm of social stream clustering. However, incorporating structure, content and time aspects simultaneously and being able to query the nearest neighbors within a cluster or obtaining cluster assignment probabilities in near real-time is a challenging problem. This is different from traditional clustering in graph streams as it encompasses content and meta information about nodes and edges. Link prediction is similarly another interesting problem, which has been solved in static and dynamic scenarios. However, when given content information propagated by nodes and their meta data, finding relevant links for recommendation based on recent content interactions is quite challenging. Again this problem is very different from link prediction in heterogeneous graphs [11] and streaming link prediction [30], as it does not take into account the content and time of propagation simultaneously.

In several social and online applications users create several implicit signals for analysis, based on their online interaction. For example, listening habits in Last.Fm or Pandora.com, along with the social relationships, can be used to understand an users musical interests and make right recommendations. There are several papers [28, 15, 23, 19, 17] that discuss about making recommendations incorporating the temporal dynamics. However, their models cannot be updated in real-time particularly for such high volume and velocity social data.

The problem of mining social streams is very important to discover real-time trends, using both content and structural information with a wide variety of practical applications. We discussed two important applications: event detection and influence analysis. There are other interesting social network problems that are less studied in the context of social streams, such as link prediction and recommendation. These areas will grow significantly in the next few years with the advent of scalable and distributed infrastructure, availability of multiple social streams, and need for real-time answers.

References

[1] C. Aggarwal and K. Subbian. Evolutionary network analysis: A survey. *ACM Computing Surveys (CSUR)*, 47(1):10, 2014.

[2] C. C. Aggarwal. An introduction to social network data analytics. *Social network data analytics*, pages 1–15, 2011.

[3] C. C. Aggarwal and K. Subbian. Event detection in social streams. In *SDM*, pages 624–635. SIAM, 2012.

[4] J. Allan, V. Lavrenko, and H. Jin. First story detection in tdt is hard. In *CIKM*, pages 374–381. ACM, 2000.

[5] J. Allan, R. Papka, and V. Lavrenko. On-line new event detection and tracking. In *SIGIR*, pages 37–45. ACM, 1998.

[6] S. Bhagat, A. Goyal, and L. V. Lakshmanan. Maximizing product adoption in social networks. In *WSDM*, pages 603–612, 2012.

[7] T. Brants, F. Chen, and A. Farahat. A system for new event detection. In *SIGIR*, pages 330–337. ACM, 2003.

[8] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *KDD*, pages 1029–1038. ACM, 2010.

[9] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–208. ACM, 2009.

[10] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[11] Y. Dong, J. Tang, S. Wu, J. Tian, N. V. Chawla, J. Rao, and H. Cao. Link prediction and recommendation across heterogeneous social networks. In *ICDM*, pages 181–190. IEEE, 2012.

[12] A. Goyal, F. Bonchi, and L. V. Lakshmanan. Learning influence probabilities in social networks. In *WSDM*, pages 241–250. ACM, 2010.

[13] A. Goyal, F. Bonchi, and L. V. Lakshmanan. A data-based approach to social influence maximization. In *VLDB*, pages 73–84, 2011.

[14] A. Goyal, W. Lu, and L. V. Lakshmanan. Simpath: An efficient algorithm for influence maximization under the linear threshold model. In *ICDM*, pages 211–220, 2011.

[15] K. Kapoor, K. Subbian, J. Srivastava, and P. Schrater. Just in time recommendations: Modeling the dynamics of boredom in activity streams. In *WSDM*, pages 233–242. ACM, 2015.

[16] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146. ACM, 2003.

[17] Y. Koren. Collaborative filtering with temporal dynamics. In *Communications of the ACM*, 53(4):89–97. ACM, 2010.

[18] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429. ACM, 2007.

[19] X. Li, J. M. Barajas, and Y. Ding. Collaborative filtering on streaming data with interest-drifting. *Intelligent Data Analysis*, 11(1):75–87, 2007.

[20] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology*, 58(7):1019–1031, 2007.

[21] G. Salton and M. J. McGill. Introduction to modern information retrieval. 1986.

[22] H. Sayyadi, M. Hurst, and A. Maykov. Event detection and tracking in social streams. In *Icwsm*, 2009.

[23] K. Subbian, C. Aggarwal, and K. Hegde. Recommendations for streaming data. In *CIKM*, pages 2185–2190. ACM, 2016.

[24] K. Subbian, C. Aggarwal, and J. Srivastava. Mining influencers using information flows in social streams. *TKDD*, 10(3):26, ACM, 2016.

[25] K. Subbian, C. Aggarwal, and J. Srivastava. Content-centric flow mining for influence analysis in social streams. In *CIKM*, pages 841–846. ACM, 2013.

[26] K. Subbian, C. C. Aggarwal, and J. Srivastava. Querying and tracking influencers in social streams. In *WSDM*, pages 493–502. ACM, 2016.

[27] K. Subbian and P. Melville. Supervised rank aggregation for predicting influencers in twitter. In *SocialCom*, pages 661–665. IEEE, 2011.

[28] J. Z. Sun, K. R. Varshney, and K. Subbian. Dynamic matrix factorization: A state space approach. In *ICASSP*, pages 1897–1900. IEEE, 2012.

[29] Y. Yang, J. Zhang, J. Carbonell, and C. Jin. Topic-conditioned novelty detection. In *KDD*, pages 688–693. ACM, 2002.

[30] P. Zhao, C. Aggarwal, and G. He. Link prediction in graph streams. In *ICDE*, pages 553–564. IEEE, 2016.

33rd IEEE International Conference on Data Engineering 2017
April 19-22, 2017, San Diego, CA

http://icde2017.sdsc.edu/

http://twitter.com/icdeconf #icde17

Call for Participation

The annual ICDE conference addresses research issues in designing, building, managing, and evaluating advanced data systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences.

Venue: ICDE 2017 will be held at the Hilton San Diego Resort and Spa

Conference Events:

- Research, industry and application papers
- ICDE and TKDE posters
- Keynote talks by Volker Markl, Laura Haas, and Pavel Pevzner
- Ph.D. Symposium with keynote speaker Magda Balazinska
- Panels on (i) Data Science Education; (ii) Small Data; and (iii) Doing a Good Ph.D.
- Tutorials on (i) Web-scale blocking, iterative, and progressive entity resolution; (ii) Bringing semantics to spatiotemporal data mining; (iii) Community search over big graphs; (iv) The challenges of global-scale data management; (v) Handling uncertainty in geospatial data.
- Topic-based demo sessions on (i) Cloud, stream, query processing, and provenance; (ii) Graph analytics, social networks, and machine learning; and (iii) Applications, data visualization, text analytics, and integration.

General Chairs:	PC Chairs:
Chaitanya Baru (UC San Diego, USA)	Yannis Papakonstantinou(UC San Diego)
Bhavani Thuraisingham (UT Dallas, USA	Yanlei Diao (Ecole Polytechnique, France)

Affiliated Workshops:

- **HDMM 2017:** 2nd International Workshop on Health Data Management and Mining
- **DESWeb 2017:** 8th International Workshop on Data Engineering Meets the Semantic Web
- **RAMMMNets 2017:** Workshop on Real-time Analytics in Multi-latency, Multy-party, Metro-scale Networks
- **Active'17** Workshop on Data Management on Virtualized Active Systems and Emerging Hardware
- **HardDB 2017** Workshop on Big Data Management on Emerging Hardware
- **WDSE:** Women in Data Science & Engineering Workshop

IEEE computer society

IEEE Advancing Technology for Humanity

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name ____________________

IEEE Member # ____________________

Mailing Address ____________________

____

Country ____________________

Email ____________________

Phone ____________________

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au