ABSTRACT

Title of dissertation: REAL-TIME ANALYTICS ON LARGE DYNAMIC GRAPHS

Jayanta Mondal,
Doctor of Philosophy, 2015

Dissertation directed by: Professor Amol Deshpande
Department of Computer Science

In today's fast-paced and interconnected digital world, the data generated by an increasing number of applications is being modeled as dynamic graphs. The graph structure encodes relationships among data items, while the structural changes to the graphs as well as the continuous stream of information produced by the entities in these graphs make them dynamic in nature. Examples include social networks where users post status updates, images, videos, etc.; phone call networks where nodes may send text messages or place phone calls; road traffic networks where the traffic behavior of the road segments changes constantly, and so on. There is a tremendous value in storing, managing, and analyzing such dynamic graphs and deriving meaningful insights in real-time. However, a majority of the work in graph analytics assumes a static setting, and there is a lack of systematic study of the various dynamic scenarios, the complexity they impose on the analysis tasks, and the challenges in building efficient systems that can support such tasks at a large scale.

In this dissertation, I design a unified streaming graph data management framework, and develop prototype systems to support increasingly complex tasks on dynamic graphs. In the first part, I focus on the management and querying of distributed graph data. I develop a hybrid replication policy that monitors the read-write frequencies of the nodes to decide dynamically what data to replicate, and whether to do eager or lazy replication in order to minimize network communication and support low-latency querying. In the second part, I study parallel execution of continuous neighborhood-driven aggregates, where each node aggregates the information generated in its neighborhoods. I build my system around the notion of an aggregation overlay graph, a pre-compiled data structure that enables sharing of partial aggregates across different queries, and also allows partial precomputation of the aggregates to minimize the query latencies and increase throughput. Finally, I extend the framework to support continuous detection and analysis of activity-based subgraphs, where subgraphs could be specified using both graph structure as well as activity conditions on the nodes. The query specification tasks in my system are expressed using a set of active structural primitives, which allows the query evaluator to use a set of novel optimization techniques, thereby achieving high throughput.

Overall, in this dissertation, I define and investigate a set of novel tasks on dynamic graphs, design scalable optimization techniques, build prototype systems, and show the effectiveness of the proposed techniques through extensive evaluation using large-scale real and synthetic datasets.

# REAL-TIME ANALYTICS ON LARGE DYNAMIC GRAPHS

by

Jayanta Mondal

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

# Acknowledgments

I feel greatly humbled in completing this dissertation. I owe my gratitude to all the people who have made this possible through their constant help, guidance, support, and encouragement.

Foremost, I want to convey my heartfelt appreciation to my advisor, Dr. Amol Deshpande. It is only due to his patience and support, I have successfully achieved this landmark. He gave me the freedom to explore on my own, and at the same time the guidance to stay on the right track. His advice on how to approach problems in a structured fashion and express ideas with clarity, has helped me mature as a researcher. I feel very fortunate to have him as my advisor.

Additionally, I would like to acknowledge my committee members V.S. Subrahmanian, Hector Corrada Bravo, Tudor Dumitras, Louiqa Raschid, for their constructive feedback and critiques. Special thanks to Tudor for the wonderful time I had collaborating with him. His guidance has helped me garner a lot of knowledge outside my core area of expertise. I would also like to thank my mentor Dr. Sudipto Das at Microsoft for his guidance during my internships. Working with him has helped me broaden my vision about database research and gain valuable insights about doing research in industry.

I would like to take this opportunity to thank the computer science department staff, Jenny Story, Fatima Bangura, Brenda Chick, Sharron McElroy, Jodie Gray, Adelaide Findlay for making my life much simpler by taking care of the dreaded administrative matters.

Life is nothing without friends. I will cherish the time spent with my grad school friends Rajan, Udayan, Hui, Ashwin, Theodoros, Abdul, Souvik, Amit, Rajesh, Varun, Rajibul, Anirban, Shrin. All the technical and non-technical conversations we had, the meals we shared, the trips to Board and Brew will always be in my memory. This acknowledgement would not be complete without a shout-out to my long-time friends Suman, Sahani, Pravuda, Anushri, Ritwik, Arnab, Kaushani, Moumita, Nagda, Rajarshi, Dibyajyoti, Shaoni. Thank you guys for enriching my life with all the love, memories, conversations, food, and adventures. Special thanks to my girlfriend Jineta for being a great friend, critic, and guide over the years. Above all, I want to thank my parents Sukla and Jaychand and my brother Joydip, for their selfless love and support. Everything I have achieved in my life, I owe every bit to them.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Graph-structured data naturally arises in a variety of application domains, including social networks, communication networks, traffic networks, phone call networks, email networks, financial transaction networks, to name a few. These graphs encode fine-grained relationships among the entities in them. For example, in social networks, users are connected via friendship relationships; in traffic networks, locations are connected via road segments, in financial transaction networks users are connected via transactions or trades, etc. These graphs also exhibit tremendous amount of structural diversity depending on the roles of the entities. For example in social networks, some of the distinct structural patterns appear in the form of celebrities and their followers, close-knit special-purpose groups, densely connected communities, users acting as bridges between communities, and many more. Similarly, in a cell phone network, variations in calling behavior lead to different structural patterns depending on whether a user tends to receive more phone calls than he/she places; whether a user is a spammer who calls a different set of people everyday; whether a set of users communicate revealing the hierarchy of the organization they work for, and so on. Detection and analysis of these interconnection patterns have a lot of value as they can reveal interesting insights like: *Who are the most influential users in a social network? Which users in a social network should be chosen as a mediator between two communities? Which callers show anomalous behavior in phone call activity in a cell phone call network? Which webpages are more authoritative in web graphs? Which group of users in a social network might have ties with or are part of terrorist organizations?* This has lead to a large body of research, over the past decade, commonly categorized as *graph analytics* or *graph mining*. Aggarwal et al. [24] and Chakrabarti et al. [52] are good references for a detailed coverage of techniques and applications of data mining techniques that exploit graph structure.

While most of work in graph mining literature focuses on structural analysis of static graphs, in many of the domains these graphs are dynamic in nature, as the nodes actively produce a lot of information, and they also interact with each other. For example, in a social network, users produce a stream of status updates, news articles, photos, videos, and a variety of other types of content. They also interact with each other via instant messages, through tags, and mentions. Similarly, in a cell phone network, the nodes generate a trace of location coordinates as they travel around; they also interact with other users via phone calls and text messages. Such activities/interactions in these graphs have added a new dimension to the graph analytics, as reasoning about the interconnection and the activity/interaction patterns together can reveal interesting behavioral events in these graphs. Following are few examples of such events.

**Example 1.** *A natural structural pattern in a graph is a* star, *that captures a node along with its connections. Activity behaviors of the participating node in a star often reveal different event semantics. For example, in a communication network, if the central node/device always sends one-way messages to the peripheral nodes/device, it might be an example of command-and-control behavior shown by a botnet master. Considering an example from social media, a Facebook page is connected with its followers in a star pattern, and frequent two-way communications between the page administrator and the followers indicate that the Facebook page is well-maintained.*

**Example 2.** *In social networks, users are often recommended items based on what their friends have liked or posted. However, quality of such recommendations would increase if the users could be categorized based on topic affinity (i.e., whether they like/post about sports, politics, technology, etc.), and they are recommended using items from friends who show similar topic affinities.*

**Example 3.** *Synchronized behavior (i.e., a set of nodes repeatedly performing the same activities over a period of time) in many application domains indicates unusual behavior. For example, such behavior in a user recommendation system like Yelp might indicate a coordinated effort to influence the ratings of some businesses [87].*

**Example 4.** *A clique of users in a social network indicates that they all know each other. However, an active clique, i.e., a clique where all the users are active at the same time, can have a variety of interpretations and utility. An active clique where the nodes rarely communicate with the rest of the network, would indicate suspicious behavior. Moreover,* *by analyzing the number of activities by the nodes within a clique, it is possible to identify the group dynamics (i.e., leader vs subordinate).*

The importance of such analysis is ever increasing as these event-based behavioral insights are often key to the success of a business or an organization. However, performing such analysis tasks in a highly interconnected, dynamic, and high data-rate environment poses a unique set of challenges. Firstly, many analytical tasks on dynamic graphs have real-time or near real-time latency requirements. This is a very natural requirement considering the fact that most interaction patterns come with an expiration time, and the usefulness of detecting/analyzing an interaction pattern would decrease as time progresses. This is especially true for *anomaly detection* [1] where the utility of detecting an event decreases drastically soon after its occurrence time. Secondly, the nature of queries poses different challenges too. The queries can vary along several dimensions like execution frequency, number of instances, latency requirements, and query scope (i.e., what portion of the graph needs to be processed). Such variations necessitate a variety of language constructs to specify the wide range of queries, and different techniques to evaluate them efficiently. For example, newsfeed query [2] in a social network requires support for handling a very large number of parallel queries with low latencies. Such a query has 1-hop query scope (i.e., immediate set of friends) and needs to be executed whenever a user visits the newsfeed webpage. On the other hand, computing social recommendations requires performing more complex tasks (e.g., computing the most popular news articles posted in a user's friend circle) and perhaps has relatively relaxed latency requirements.

[1] `https://en.wikipedia.org/wiki/Anomaly_detection`

[2] Newsfeed of a user is typically described as a list of recent (recency varies across applications) events performed by the friends (often by friends of friends) that user.

Finally, the sizes of these networks and unprecedented growth rate often necessitate the use of parallel and distributed solutions. However, graph partitioning is a notoriously hard problem, and hence even simple queries over a distributed graph may result in a large number of traversals across the network. Moreover, graph operations are not very easy to parallelize. Even though the MapReduce framework has proven successful in executing large-scale analysis tasks in a distributed fashion, it is mainly aimed towards batch processing of largely static data, and cannot support real-time data ingest or real-time querying.

These challenges, however, are not very well addressed by the state-of-the-art graph processing systems (e.g., Pregel [110], GraphLab [106], GraphX [73], Giraph [8]). Most of these frameworks are designed to execute a single large analytical task (e.g., pagerank, all pairs shortest paths, centrality computation etc.) on the entire graph. Moreover, these systems are optimized for throughput, operate in a batch-processing mode and can't produce results in an interactive manner. There is also much work on dedicated stream processing systems (e.g., Storm [142], Spark Streaming [151]) that are excellent at performing stateless aggregation in a distributed fashion, but are designed for largely relational data and cannot handle graph analysis tasks. There has also been much work in the graph mining literature. Many recent works [87, 98] in graph mining have shown the benefit of analyzing the structure and interaction patterns together. Even though they serve as a major source of motivation, most of these works do not focus on building a unified abstraction that could be used to specify multiple different types of tasks or to build general-purpose systems to execute these tasks in an efficient manner. Moreover, they specialize in deriving important insights from specific datasets.

My dissertation research is motivated by this lack of a general-purpose system that exposes an easy-to-use language to specify analytical tasks on dynamic graphs, and executes those tasks in an efficient manner. I systematically study different types of queries on dynamic graphs, and address the aforementioned challenges by developing a *streaming graph data management system*.

In the rest of this chapter, I provide a background of how dynamic graphs are modeled, what are the different classes of queries on dynamic graphs, how and when the answers need to be produced for the queries, and other related basic concepts.

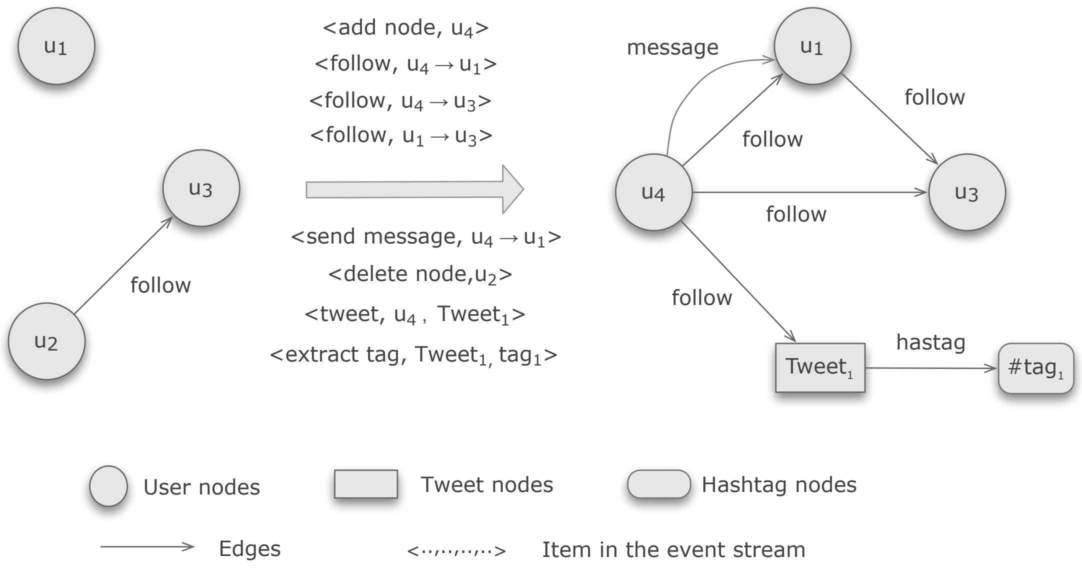


Figure 1.1: Example of the data components of a dynamic graph

## 1.1 Data Components of Dynamic Graphs

The data in a dynamic graph is comprised of two major components: (a) network component, and (b) stream component. However, they are not independent. I first describe the network and stream components independently, and then discuss the dependencies between them. Figure 1.1 shows an example a dynamic graph from the domain of social networks.

| Notation | Description |
|---|---|
| $\mathcal{G}(V, E, V_p, E_p)$ | Property Graph |
| $V$ | Set of nodes in $\mathcal{G}$ |
| $E$ | Set of edges in $\mathcal{G}$ |
| $V_p$ | Set of node properties in $\mathcal{G}$ |
| $E_p$ | Set of edge properties in $\mathcal{G}$ |
| $\mathcal{A}_S$ | Activity Schema |
| $A_{\mathcal{G}}$ | Activity Stream |
| $f_1(), f_2(), ...$ | Stream querying or reasoning tasks |
| $\mathcal{N}^k(v)$ | $k$-hop ego network of node $v$ |

Table 1.1: Notation used in the chapter

### 1.1.1 Network Component

The *network* component captures the underlying interconnection structure among the *nodes* in the graph. The network component in our abstraction is represented using the property graph model [21] which is a popular model to represent graph data. Let us assume that $\mathcal{G}(V, E, V_p, E_p)$ denotes the property graph, where $V$ and $E$ represent the sets of nodes and edges respectively [3]. $V_p$ and $E_p$ denote the set of properties defined on the nodes and the edges, where each property could be seen as a key-value pair. In general, $\mathcal{G}$ is a heterogeneous, multi-relational graph that may contain many different types of nodes and may contain both directed and undirected edges. For example, in the context of social graphs, nodes may represent users, communities or groups, user tags, webpages, and so on. Similarly, $E$ may include not only symmetric (e.g., *friendship*) edges, but

[3]Table 1.1 summarizes the notations that we use in this Chapter.

may include asymmetric edges (e.g., *follows* edges, *membership* edges), and other types of semi-permanent edges that are usually in existence from the time they are formed till the time they are deleted (or till the current time). Note that, such heterogeneity can be seamlessly captured using the property graph model without any loss of generality. For example, in case of the social graph example, nodes of type users and pages could be accommodated by simply adding a *node_type* property in $V_p$.

### 1.1.2 Stream Component

The *stream* component is comprised of a continuous stream of events originating in the network. The events could be categorized under the following high-level categories: (a) *structural events* that represent deletion/addition of nodes and edges, (b) *content events* that represent an individual entity producing some data, (c) *interaction events* that represent some form of interaction or communication among entities.

In this dissertation, I introduce the notion of *activity schema ($\mathcal{A}_S$)* as a unified abstraction for all different types of events in the stream component. Activity schema is a collection of event schemas. An event schema is represented by a two-tuple $\langle type, Attr_{KV}\rangle$, where *type* represents a unique activity name and $Attr_{KV}$ contains a set of attribute (key_name, value_datatype) pairs. Example 5 shows an example of a possible activity schema for *phone call activity*.

**Example 5.** *In a cellphone call graph, "phone call activity" could be represented using the schema* $\langle$**call**, $\{$caller:String, callee:String, duration:Float$\}\rangle$.

Given the abstraction of the activity schema, the *activity stream ($A_G$)* is modeled as an unbounded stream of time-stamped events where each event follows an event schema from $\mathcal{A}_S$[4]. Example 6 shows example of an event that conforms to the event schema definition in Example 5.

**Example 6.** *Example of an event that conforms to the activity schema above is:* $\langle$**call**, $\{$caller:“Alice”, callee:“Bob”, duration:“10s”$\}$, time:“10:00am”$\rangle$.

We assume that the event timestamps represent the *stream times* of the events. Borrowing from stream processing literature, stream time is the time at which an event is observed by the stream processing system, i.e., the current time according to the system clock. This is different from *event time* which represents the record of system clock time (for whatever system generated the event) at the time of occurrence.

In an ideal world, we would expect the event time and stream time to be the same, however, this is not the case for virtually all real-world applications. In our work, we assume that our system sees all the events in *stream time*. Moreover, we assume that there are no out of order events, and the delivery platform that feeds the data to our system takes care of that issue[5].

The notion of *activity schema* has a close analogy in the relational data model. The relations (i.e., tables) are the central entities in a relational model, and each relation has a schema definition. A data manipulation language uses the relation schemas to specify operations on the data. Similarly, in our case we have schema definitions for different types

[4]An event that doesn't conform to any of the event schema from $\mathcal{A}_S$ can be configured to be ignored by the system.

[5]There exists a rich literature on the *out of order* data issue, and we don't do further exploration of this issue in this dissertation.

of events in the stream. This allows for declarative specification of *triggers (or adapters)* on the incoming event stream that can be used to construct different types of graphs. For example, if the goal is to build a user-to-hashtag bipartite graph using tweet stream, in our system, a user can write a declarative trigger that converts each $\langle user, hashtag\rangle$-pair into two nodes (one for the user and the other for the hashtag) and an edge in the data graph. I defer a detailed discussion of this topic to Chapter 5.

### 1.1.3 Dependancy between Network and Stream Components

In a dynamic graph, the network and stream component are not independent as the activity stream constantly changes the state of the graph. The simplest forms of changes are the structural changes caused by the addition and deletion of nodes and edges. However, there could be more complex cases depending on the application domains and design choices of the attributes (of the nodes/edges). An example could be the *timeline* attribute of a Twitter or Facebook user. The timeline could be modeled as an append-only attribute, which gets appended whenever a user posts a tweet/status (i.e., a content event). The attributes could also be dynamic when we allow them to be derived. For example, if we consider the number of status messages posted by a user in last 5 hours to be an attribute of the user, the attribute would continuously change as time progresses. In our system, we let the users define such attributes using the activity schema. I discuss the formal definition of such dynamic, active attributes in Chapter 5.

## 1.2 Task Categories

With the intuitive description of the data model, now we are ready to describe how queries could be specified on that data. However, before that, I provide a brief survey of different types of tasks that are of interest on dynamic graphs.

### 1.2.1 Large Number of Independent Local Queries

In many cases, there is an interest in executing a large number of independent queries, each corresponding to a node in the graph, where the answer to a query is computed by traversing the local neighborhood of the query node, and simultaneously processing the structural and content information of the visited neighborhood. The actual nature of the computation could vary across applications. The simplest form of computation involves fetching and organizing the content stream generated by the neighbors of the query node. This is also known as *feed following*, and is one of the most common queries on social networks. Millions of such queries often need to be executed in parallel. A more complex version of the feed following query involves selection of neighbors based on filtering conditions. For example, an application might be interested in serving newsfeed from friends who live in the same town as the queried user. Other examples of local queries might require search [63] or aggregation over the attribute values or the content stream.

Depending on the application domain, queries in this category might have different requirements. For example, the feed following and the graph search queries are point queries, and have real-time latency requirements. On the other hand, computation of local trends could be done in batches over a sliding window, and is relatively less latency-critical. However, such batch aggregation queries would typically need to achieve high throughput.

### 1.2.2 Global Analytics over Content-dynamic Graphs

While local queries described above are extremely important for personalization and recommendation, analyzing data from the entire network can generate the following global insights: What are the most popular topics of international interest? Which users are the most influential? What is the most common/shortest path to propagate information from a node A to node B? Some recent works like Kineograph [57] and GraphInc [49] addressed this type of continuous analytics over dynamic graphs. Unlike local queries, the number of global queries run on a graph is typically much smaller. Moreover, they are often not interactive and don't have a strict real-time requirement.

### 1.2.3 Event/pattern Detection on Dynamic Graphs

This category represents queries that aim to detect events or anomalies on a graph whose structure evolves very rapidly. The events are typically pre-specified using the structural as well as attribute-specific properties of the network. For example, in a recent work, Srivastava et al. [30] developed a solution to detect new events of interest, by finding and incrementally maintaining dense subgraphs of a large hashtag graph, which was constructed by treating hashtags as nodes, and by adding an edge between every pair of co-occurring hashtags. Other examples include detection of anomalies and patterns in communication networks or financial transaction networks. These queries are very latency-critical, and need to be evaluated in a continuous manner.

### 1.2.4 Managing a Large Collection of Small Graphs.

Applications in this category need to manage a large collection of small graphs, in contrast to dealing with a single huge graph. Security applications often model activities (software downloads or file accesses) on host machines as graphs, leading to a big collection of graphs. Structural properties (e.g., density, clustering coefficient, and others) computed on these graphs are then used by machine learning algorithms (e.g., for classifying malware from benign software, detecting *advanced persistent threat*[6] by analyzing the file access patterns). Such structural features, being relatively domain-agnostic, are thought to be more robust. Note that, these graphs are typically constructed at a centralized server from a stream of activity events relayed from the host machines. The need to manage such a large collection of relatively small graphs, also arises in *influence propagation*, where one graph is maintained per entity of interest (e.g., hashtag influence graphs).

## 1.3 Task/Query Model

Given the data model in the previous section and different flavors of tasks, now I move to the discussion of the query model. For clarity, I divide the discussion along three dimensions: (a) query issuers, (b) query scope, and (c) answering model. Query

[6]`http://en.wikipedia.org/wiki/Advanced_persistent_threat`

issuers implicitly or explicitly pose queries to our system (e.g., a user could be system administrator or a consumer of the service provided by the system). The query scope defines how much of the network data (i.e., network scope) and what portion of the stream data (i.e., temporal scope) needs to be considered for a query. The answering model captures how the answers are expected to be evaluated and reported (i.e., whether the answer is one-time vs needs to be computed continuously at a fixed interval, or an answer needs to be produced whenever an event of interest happens on the network). Figure 1.2 summarizes these dimensions visually, using several example tasks.

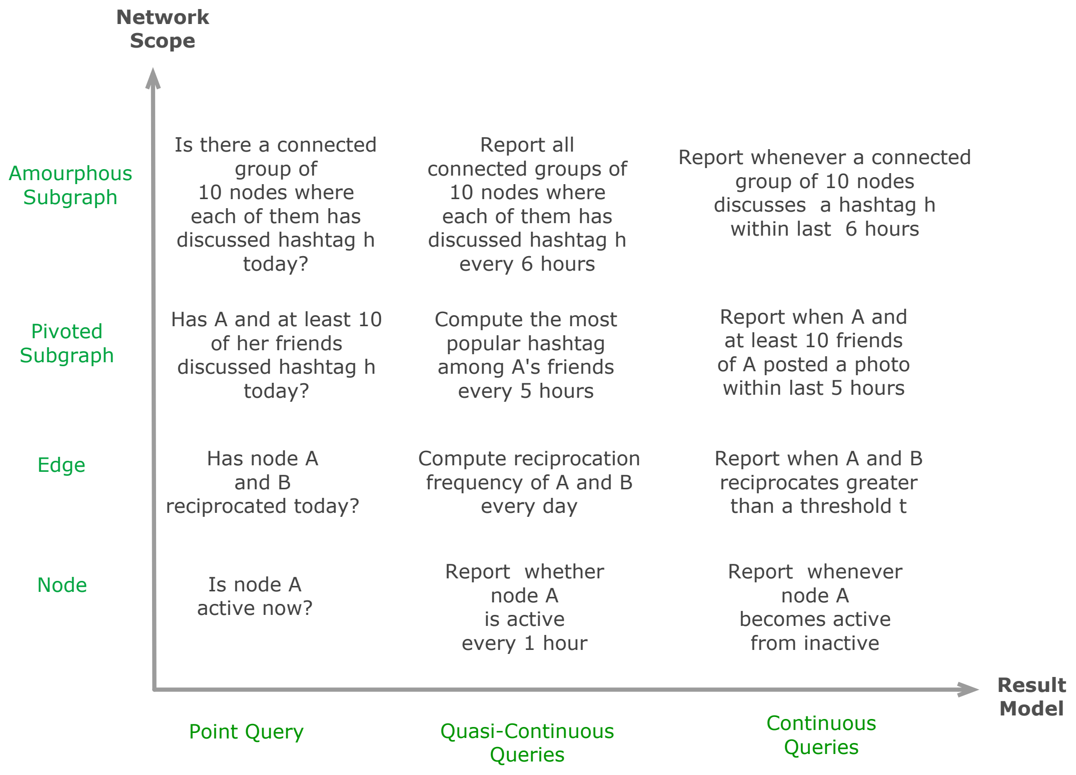

Network Scope
Amourphous Subgraph
Pivoted Subgraph
Edge
Node
Is there a connected group of 10 nodes where each of them has discussed hashtag h today?
Report all connected groups of 10 nodes where each of them has discussed hashtag h every 6 hours
Report whenever a connected group of 10 nodes discusses a hashtag h within last 6 hours
Has A and at least 10 of her friends discussed hashtag h today?
Compute the most popular hashtag among A's friends every 5 hours
Report when A and at least 10 friends of A posted a photo within last 5 hours
Has node A and B reciprocated today?
Compute reciprocation frequency of A and B every day
Report when A and B reciprocates greater than a threshold t
Is node A active now?
Report whether node A is active every 1 hour
Report whenever node A becomes active from inactive
Result Model
Point Query
Quasi-Continuous Queries
Continuous Queries

Figure 1.2: Examples of queries showing various possible *network traversal scope* and *result reporting model*. For all this examples we have assumed a *sliding window-based temporal scope*, as it is the most common temporal scope used in a streaming setting.

### 1.3.1 Query Issuer

In our system, the query issuers could be categorized under three high-level categories.

- **Consumer:** Consumers use the services of a service provider by implicitly querying an underlying system in a specific manner. For example, in the case of Facebook the consumers are the individuals with Facebook accounts. Consumers typically issue small *read* and *write* queries to the system (i.e., queries that touch limited amounts of data and take very little time to execute). For the evaluation of the system described in Chapter 3, we focus on consumer-issued writes (i.e., content generation by the nodes in a social network) and reads (generating newsfeed by fetching updates from the friends). In Chapters 4 and 5, we consider more complex queries (i.e., aggregation queries and event detection queries), those queries can be thought of as meta-queries issued by the service provider that group together simple queries issued by the consumers.

- **Service Provider:** On the other hand, a *fraud detection task* is an example of a task issued by a service provider. Such queries are typically either *long standing complex event detection queries* or *analytical queries* on a large amount of data. The queries that I discuss in Chapter 4 and 5 would be typically issued by the service providers or system administrators.

- **Third Party Client:** Advertisers and application developers are examples of third party clients on a platform like Facebook. They typically issue queries through

Developer APIs supported by the system. In case of Facebook, advertisers can issue *advertisement targeting queries* to the Facebook system by specifying which set of users to target their advertisements to. The queries that I address in Chapter 5 could be issued by a third party client.

### 1.3.2 Query Scope

Now I look at what forms the *input* to a general query defined on a dynamic graph. Following is a classification of the queries/tasks by their *input scope*. Broadly speaking, there are two crucial dimensions along which the queries/tasks may differ.

#### 1.3.2.1 Network Traversal Scope

The first key dimension is what we call *network traversal scope* of a query, which refers to the portion of the network that provides the input to a query or task.

- **Node Scope:** The simplest form of network scope is a node in the network. For *node scope*, the user of the system might be interested in knowing the behavior of a node in isolation, i.e., independent of the influence of any other graph entities. For example, we might be interested in knowing whether a node A has been active on Twitter. Here the *active* attribute could be based on whether the node has posted more than 10 tweets in a day or not.

- **Edge Scope:** The next important network scope would be an edge in the network. An *edge scope* captures the interaction between two nodes in the system. For example, in a social network, *message reciprocity* could be an interaction of interest.

Message reciprocity could be defined as the proportion of two-way messages between two nodes. Ignoring the quantitative definition, we could label two nodes as reciprocative.

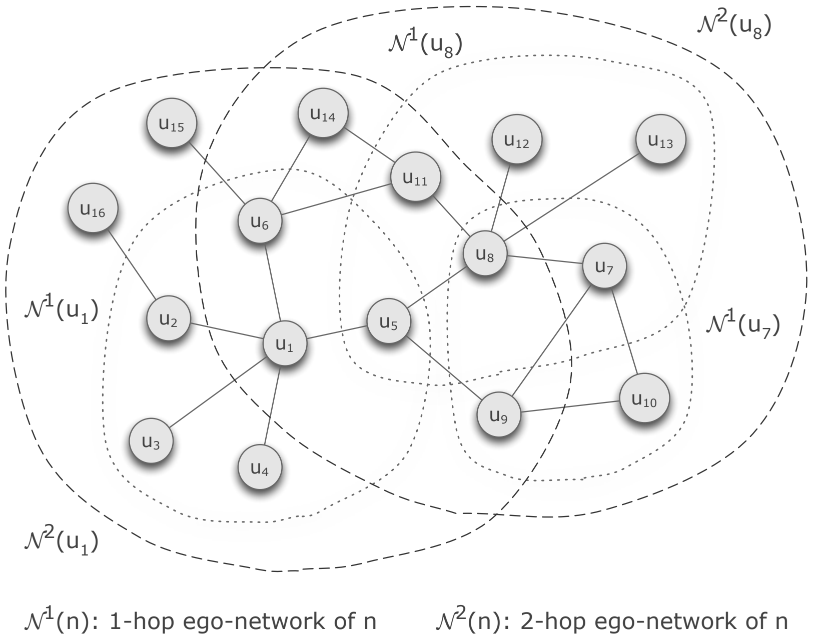


Figure 1.3: Stream queries often have ego-centric scope: figure shows 1-hop ego-networks of $u_1, u_7$, and $u_8$, and 2-hop ego-networks of $u_1$ and $u_8$.

- **Pivoted-Subgraph Scope:** In many cases, a task or a query may only focus on a local neighborhood in the network, often termed *ego networks*. For example, if the goal is to identify *social circles* for a user [111], then only a 1- or 2-hop neighborhood around the user may be of interest (Figure 1.3). Personalized trend detection task is another example of such a task. Note that, in many cases, we may want to execute the same task for every node in the network (e.g., we may wish to do continuous trend detection for every user of the social network), and in total, updates in the entire network may need to examined. However, those should be treated as separate tasks each of which is ego-centric in scope. The most common example of an ego network is the network over the immediate set of neighbors of a node. However, in general, an ego network of a node could be defined as *k-hop neighborhood* containing all nodes reachable within $k$ hops from the node (and all the incident edges among those). We use the notation $\mathcal{N}^k(v)$ to represent *k-hop* neighborhood of a node $v$.

- **Amorphous Subgraph Scope:** In many cases, a task may be defined over an arbitrary subgraph of interest. For example, an advertiser might be interested in performing some analytics over a subgraph induced by people from a specific geographic location and belonging to a specific age-group. Note that, ego-centric scope also effectively defines a subgraph of interest, however, due to special structure of the ego-centric scope (i.e., the subgraph is centered around a central node or ego), it is treated separately.

- **Global scope:** Many tasks require computation over the entire network. An example of such a task is computation of *PageRank* (or other centrality measures like *betweenness centrality*, *eigenvector centrality*, etc.). Dense subgraph maintenance task is also an example of a task with global scope.

#### 1.3.2.2 Temporal Scope

The second key dimension captures the temporal scope of the task, and has a direct impact on the amount of state that must be stored, accessed, and analyzed.

- **Entire stream:** At one extreme, the temporal scope of a task may stretch from the beginning of the stream to the current time. Note that, not all the data generated so far may be of interest – e.g., the task may only see a subset of the data by choosing to focus only on certain attributes of the nodes or edges. However, the data of interest may have arrived into the system at any point in the past. For example, in a social network with location data, a task may wish to process all the location updates ever produced by a user for predicting future user movements. We expect such types of tasks to be somewhat uncommon given the large volumes of data generated in most dynamic graphs.

- **Current state of the network:** Many tasks will take the current state of the network as the input. An example of this task is *online dense subgraph maintenance* [30] where the goal is to maintain the dense subgraphs of the current social network at all times.

- **Sliding window:** The third alternative, that falls in between the two extremes above, is that the task defines a sliding window on the data stream, and the input consists of all updates that arrive during that window. For instance, one may be interested in analyzing all messages that were exchanged during the last 24 hours among the users of a network to identify anomalous behavior in real-time. Another example of such a task is detection of personalized trends where the goal is to find the most commonly seen words or phrases in the recent status updates or blog posts by the friends of a user. As time progresses, the window slides and new message edges will be added to the graph and old message edges (that fall out of the window) will be deleted.

### 1.3.3 Answering Model

Based on the specific nature of the query issuer, the expected return time of the query-output would differ. Following is a categorization of the queries based on their expected return time:

- **Point:** Queries issued by the consumers are typically expected to return immediately (however, this might change depending on the nature of the service provided to the consumers). In case of Facebook, queries like *read notifications* and *fetch newsfeed* are expected to return immediately.

- **Continuous:** In many cases, continuous execution of the query may be desired, i.e., the output is needed to be kept updated for every relevant change in the input. Anomaly detection queries typically need to be executed in this fashion since anomalies must be detected as soon as they are formed.

- **Quasi-continuous:** On the other hand, for quasi-continuous queries, a user may specify a frequency with which a query or a task needs to be executed (e.g., every hour or every day). These queries are also typically issued by the service providers and read (or consumed) by the consumers of the system. Computing Twitter trends is an example of such queries.

## 1.4 System Architecture and Execution Model

Graph data management systems often use distributed and parallel architectures to keep up with large volumes of data and the requirement to process complex queries quickly. The main trends here are:

- **Scale-out Architectures:** Scale-out (or horizontal scaling) architectures distribute the data across multiple servers (relatively cheap machines with limited RAM and processing power) and run queries in a distributed fashion. Even though scale-out architectures are cost-effective and more resilient to faults, they often suffer from high query execution costs due to the distributed nature of the queries. Moreover it is often hard to implement algorithms correctly for a distributed setting.

- **Scale-up Architecture:** Scale-up (or vertical scaling) architectures generally refer to use of more expensive and robust servers with high amount of RAM and processing power. While vertical scaling provides superior performance and ease of implementation, they are costly and often have a single point of failure.

The decision to choose between *scale-up* and *scale-out* is mostly application specific. Here are some of the key metrics that need to be consulted before making such a decision, or need optimization once such a decision is made.

- **Computation Cost:** One of the important requirements is to minimize overall computation cost as much as possible. Reduction of computation cost is the key to low CPU load, and hence increased throughput and less power consumption. The main three parameters that could significantly affect the number of computations are query frequency, average degree of nodes, and type of analysis task under consideration. In case of dynamic graphs, any change in the graph structure might require actions depending on the algorithms one is using. However, it is important to tackle such changes with care. Techniques that are reactive to dynamic changes might harm the system efficiently.

- **Network Bandwidth:** It is desirable from a distributed implementation that the communication overhead be minimized. In most of the real-time applications today, read/write operations are latency-critical and failing to keep those under acceptable limits may lead to demise of such applications. To ensure low-latency query execution, one needs to minimize the number of cross-partition traversals, and if there is no natural partitioning of the data, then we must use *replication* for that purpose. Depending on the skew in the number of *read* and *write* queries there are two common replication policies:

  - **All Push:** An *all push* approach requires replicas to be updated to all relevant partitions immediately following a write in the system. This is a common strategy for systems with low write-to-read ratio. The benefit of all push approach is that the query latencies are minimized, however, it might incur a lot of unnecessary communication. This model is also known as *push-on-change* or *eager replication*.

  - **All Pull:** An *all pull* approach requires data to be pulled from all relevant partitions during query time. This is a common strategy for systems with high write-to-read query ratio. While an all-pull model might save unnecessary communication, the query latencies are typically higher. This model is also known as *pull-on-demand* or *lazy replication*.

In a dynamically evolving graph, the cost of keeping the replicas up-to-date may exceed the benefits of replication. Furthermore, both the write and read access patterns may change dynamically, and different policies may be best at different times.

- **Latency of execution:** Ideally it is expected that all the queries would be executed with low latency, especially in case of an *online* system. Typically the consumer facing queries are the most latency-critical queries. On the other hand, anomaly detection tasks, issued by the service providers, also require anomalies to be detected as soon as they occur.

- **Data Movement:** Minimizing data movement within the memory module is analogous to communication minimization in a distributed set up. In an in-memory system this could mean data objects being copied from one memory location to another. Too many of these data movements could hurt performance. It is important to keep the data movement during the execution of queries/tasks, within a user-specified budget.

- **Throughput:** Throughput is another key metric for measuring the overall performance of such systems, especially when the real-time requirements are important. Throughput indicates total number of queries the system can execute per second. Apart from the use of scale-up and scale-out architectures to increase system throughput, researchers have also looked into complex techniques like multi-query optimization, sampling based approximate query execution, and so on.

Given the background of different system architectures and the popular optimization metrics, now I discuss the most popular execution models for executing graph algorithms.

- **Bulk Synchronous Parallel (BSP):** Under the BSP model a graph algorithm is defined as a series of global *supersteps*. Each superstep consists of the following:

  - **Concurrent Computation:** In the first step, every participating entity (e.g., nodes in the graph or subgraphs of interest) of the graph performs local computation in isolation. These computations are done in a concurrent fashion.
  - **Communication:** In the next step, output of the computation done in the previous step is communicated to the relevant neighboring entities by each participating entity.
  - **Barrier Synchronization:** Once an entity is done with these two steps it waits for all other participating entities to finish, before proceeding to the next superstep.

  Although global synchronization simplifies algorithm design, analysis tasks writing using BSP may take a long time to converge and suffer from straggler delays.

- **Asynchronous:** On the other hand, an *asynchronous* model achieves better convergence rate by eliminating the barrier and allowing an entity to proceed to the next iteration without waiting for others. However, scheduling overhead is typically high for such models due to the requirement to guarantee consistency.

Another important aspect that can affect the scalability of the execution model is the choice of the *participating entities* (or *central entities*) on which such supersteps are defined. As far as the *participating entity* is concerned the popular models are:

- **Vertex-centric model:** In a vertex centric programming model a graph algorithm is structured as a series of (i) local computations on participating nodes of the graph, (ii) followed by communicating the output of computations to the relevant neighboring nodes. Pregel [110] is an example of a system that uses a *vertex-centric BSP* model. On the other hand, GraphLab [106] is an example of a system that uses a *vertex-centric asynchronous* model.

- **Subgraph-centric model:** In a subgraph-centric programming model, a graph algorithm is structured as a series of (i) local computations on a set of subgraphs (the exact definition of the subgraphs of interest depends on the application at hand), (ii) followed by communication of the outputs to the relevant neighboring subgraphs. A subgraph-centric model has been shown to reduce [141] the communication overhead compared to a vertex-centric programming model for various application scenarios.

Note that, the *network traversal scope* that we have discussed before is orthogonal to the *programming model*. Network traversal scope defines the substrate on which a query/algorithm needs to execute, on the other hand the programming model defines how an algorithm proceeds on a given substrate.

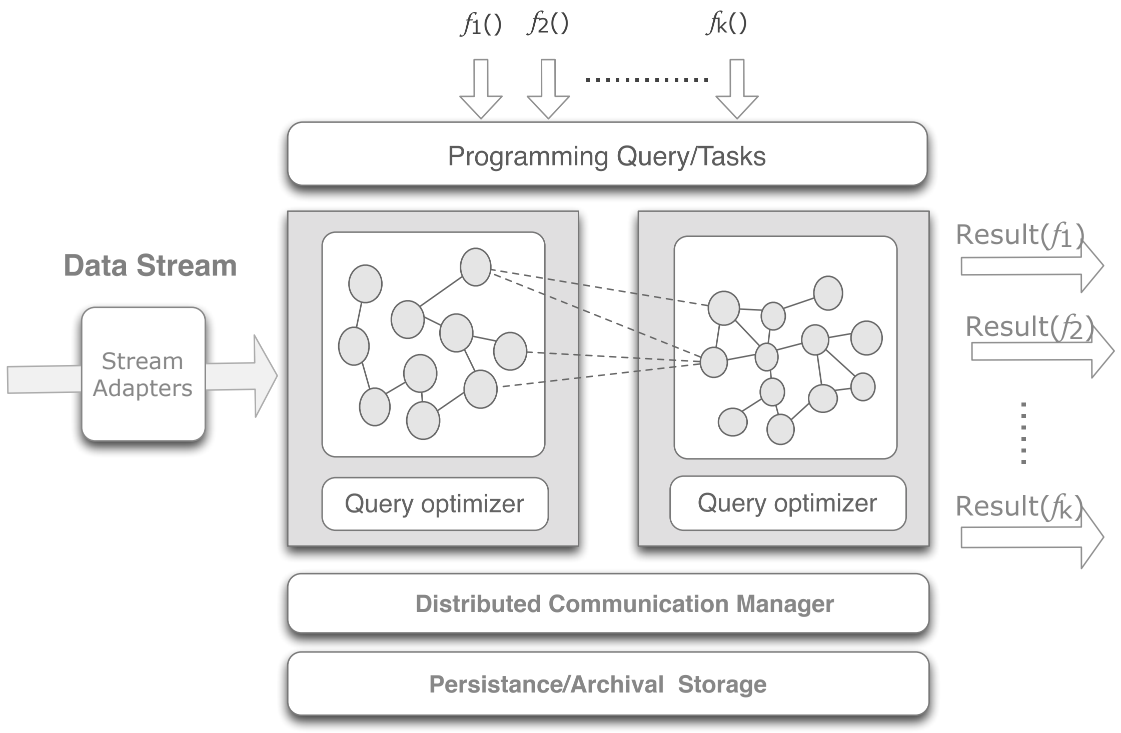


Figure 1.4: High-level overview of the graph stream querying system

## 1.5 Overview of the Dissertation Research

In this section, I provide an overview of the work that I have completed in order to tackle the challenges in developing a graph database management system than can handle real-time queries/tasks on large dynamic graphs. Specifically, I built a *distributed in-memory framework* where,

- the data graph is *distributed* across multiple machines in order to deal with the large volume of data,
- there is a set of stream adapters, defined on the activity stream, that makes the necessary changes in the underlying data graph in response to new events or updates,
- within a machine, the data (i.e., a partition of the initial graph and the data produced

by the nodes in the partition) is stored and analyzed fully *in memory* with the disk being used as backing store for historical and archival data,

- there is also a archival/persistence layer that provides an API to access historical data about the nodes which could be consulted during the analysis tasks.

Figure 1.4 depicts the high level architecture of our system, where the *graph engines* are responsible for executing the graph algorithms/queries and the *replication managers* deal with the data transfer required for distributed execution. I first discuss our prototype distributed graph database (with a simplistic graph engine) and then focus on enriching the graph engine through extending support for different types of queries/tasks.

### 1.5.1 Distributed Graph Database with Adaptive Replication

In this section I discuss the distributed graph database framework that I built. For the evaluation purpose I focus on the prevalent *"fetch neighbors' updates"* query (a read query defined over an ego-centric scope). The key challenge in building such a system is that effectively partitioning graphs is notoriously hard, especially in a dynamic environment. Standard hash-based partitioning schemes limit scalability because they end up *cutting* too many edges, i.e., placing the endpoints in different partitions. This not only increases query latencies but also increases the total network communication, thus limiting the scalability of the system. Although the problem of optimally partitioning a graph into equal-sized partitions while minimizing the edges cut is NP-Hard, there is much work on practical solutions to this problem, and several software packages are available that can generate very good graph partitionings [91, 51, 83]. These techniques however cannot handle highly dynamic graphs where the node access patterns and the graph structure itself may change very rapidly [101]. More importantly, in most practical applications, the highly interconnected nature of graph data means that clean disjoint partitions that minimize the edge-cut do not typically exist [124]. Social networks in particular are very hard to partition because of overlapping community structure, and existence of highly-connected dense components (cores) [102, 119, 120, 42] .

I investigate an aggressive replication-based approach where the key idea is to replicate the nodes in the graph to minimize the number of distributed traversals. This approach has been extensively studied in distributed systems and distributed databases (see, e.g., [147, 89]), however, to our knowledge, there is little work on understanding how to use it for distributed graph data management. In a recent work, Pujol et al. [124, 125] considered one extreme version of it for scaling online social networks: they aim to replicate the graph sufficiently so that, for every node in the graph, all of its neighbors are present locally (called *local semantics*). They also use *active replication* (i.e., a *push-on-change* model) where all the replicated data is kept up-to-date. Such an approach however suffers from very high, unnecessary communication to keep the replicas up-to-date; Facebook reportedly uses *pull-on-demand* model instead. Further, the replication overhead to guarantee local semantics, i.e., the number of average copies of each graph node, may be too high in most cases (for a sample Facebook dataset, Pujol et al. needed approximately 2 copies of each node with just 8 partitions [125]).

I design a hybrid, adaptive replication policy that uses a novel *fairness requirement* to guide the replication decisions, and utilizes predictive models about node-level read/write access patterns to choose whether to maintain the replicas actively or passively.

The fairness requirement is characterized by a threshold $\tau \leq 1$, and can be stated simply: for each graph node, the requirement is that at least a $\tau$ fraction of its neighbors be present locally at the same site. The local semantics [125] becomes a special case of this with $\tau = 1$. A key concern with a policy that makes fine-grained push-pull decisions is that, the overhead of maintaining these decisions (i.e., for a node, deciding which of the neighbor replicas are up-to-date) is very high. I design and evaluate novel clustering-based schemes for this purpose, where nodes with similar access patterns are grouped together to reduce the overhead without compromising quality. I analyze the problems of deciding what to replicate, and choosing when to push vs pull, and provide both theoretical analysis and efficient practical algorithms (optimal for the latter problem). The algorithms I designed are *decentralized* by nature, and enable us to make the decisions locally at each node. This also naturally enables us to change the decisions during periods of low load, and/or stagger the times when they are made to avoid significant slowdowns.

I have implemented a prototype distributed graph data management system using Apache CouchDB open-source key-value store. I present a comprehensive experimental evaluation which shows that my algorithms are practical, support low-latency operations, and decrease the total amount of communication by a significant fraction over other policies.

**Summary of Contributions:** My main contributions in this work are the following:

- I build a scalable and decentralized replication middleware to minimize distributed transactions in a distributed graph database. I believe that the replication middleware I built could be of independent interest to any distributed graph database where bandwidth minimization and load balancing are key concerns.

- I show that adaptive replication techniques, through monitoring of read/write frequencies of the nodes, perform much better than all push and all pull approaches and save network bandwidth by almost 30% on average.

- I show that monitoring user activity patterns at a fine granularity to make replication decisions, achieves better bandwidth savings compared to monitoring user activity patterns at a coarse granularity.

- I design a clustering-based decentralized approach to scalably handle the large number of replication decisions that need to be made in such a dynamic environment.

- I also introduce a novel fairness guarantee to provide a trade-off between network bandwidth and query execution latency. I also show *local semantics* to be a special case of *fairness factor = 1.0*.

- I perform extensive evaluation with a semi-synthetic dataset on Amazon EC2 to establish the efficacy of the techniques that I have developed.

### 1.5.2 Ego-centric Aggregation Framework for Large Dynamic Graphs

In the second component of my work, I built EAGr, a framework for evaluating Ego-centric Aggregate queries on Large Dynamic Graphs . EAGr was developed to extend the range of queries our system can support. An ego-centric aggregate query has an *ego-centric scope* and is defined as an aggregate over the current state or the recent history of the local neighborhood of a node. Typically, we need to compute such ego-centric aggregates for each node (i.e., ego) in the data graph. These queries are executed in a quasi-continuous fashion since there is no need to produce the result unless the user asks for it (for reducing latency, full or partial pre-computation may be performed). The main challenge here is to execute a large number of ego-centric queries with sufficiently low latencies under a high data-rate environment. A naive *on-demand* approach, where the neighborhood is traversed in response to a read, is unlikely to scale to the large graph sizes, and further, would have unacceptably high query latencies. On the other hand, a *pre-computation-based approach*, where the required query answers are always pre-computed and kept up-to-date will likely lead to much wasted computation effort for most queries. Furthermore, both these approaches ignore many potential optimization opportunities, in particular, the possibility of *sharing* the aggregate computation across different queries (corresponding to different ego networks).

I design an approach that maintains a special directed graph (called an *aggregation overlay graph* or simply an *overlay*) that is constructed given an ego-centric aggregate query and a subset of nodes in the data graph for which it needs to be evaluated. The overlay graph exposes sharing opportunities by explicitly utilizing *partial aggregation* nodes, whose outputs can be shared across queries. The nodes in the overlay are labeled with *dataflow decisions* that encode whether data should be *pushed* to that node in response to an update, or it should be *pulled* when a query result needs to be computed. During execution, the overlay simply reacts to the events (i.e., reads and writes) based on the encoded decisions, and is thus able to avoid unnecessary computation, leading to very high throughputs across a spectrum of workloads. Constructing the optimal overlay graph is NP-Hard for arbitrary graph topologies. Further, given the large network sizes that are typically seen in practice, it is infeasible to use some of the natural heuristics for solving this problem. I present a series of highly efficient overlay construction algorithms and show how they can be scaled to very large graphs. Surprisingly, the problem of making the dataflow decisions for a given overlay is solvable in polynomial time, and I present a *max-flow*-based algorithm for that purpose. The framework can support different neighborhood functions (i.e., 1-hop, 2-hop neighborhoods), and also allows filtering neighborhoods (i.e., only aggregating over subsets of neighborhoods). The framework also supports a variety of aggregation functions (e.g., *sum, count, min, max, top-k*, etc.), and exposes an aggregation API for specifying and executing arbitrary user-defined aggregates. I conduct a comprehensive experimental evaluation over a collection of real-world networks, and the results show that overlay-based execution of aggregation queries saves redundant computation and significantly boosts the end-to-end throughput of the system.

**Summary of Contributions:** My main contributions in this work are the following:

- I build an in-memory, general purpose, user-extensible ego-centric aggregation framework that can achieve high query throughputs for large dynamic graphs. My work is the first systematic approach that looks into ego-centric aggregation queries.

- I show that optimizing for throughput by effective sharing of partial aggregates and selective pre-computation of the aggregates is a hard problem to solve. To make the problem more tractable I decouple the problem into *overlay building* and *dataflow decision making*, instead of performing a joint optimization which is a much harder problem to solve.

- I develop a set of novel algorithms to build overlays and maintain them in an incremental way. I believe my work is of independent interest to the graph-compression community.

- I also show that the dataflow decision making problem is optimally solvable and design efficient algorithm for the same.
- I perform extensive evaluation using both real and synthetic data to show the efficacy of our techniques. Using a single powerful machine I was able to achieve a throughput of 500K queries per second.

### 1.5.3 A System for Continuous Analytics of Activity-based Subgraphs

Finally, in the third part, I focus on extending our query model to support analytical tasks on dynamically changing active subgraphs of interest. Active subgraph patterns need to reason about how the nodes behave (i.e., the activity component) in the network as well as how they are connected to each other (i.e., the structural component). An example of such an activity-based subgraph pattern is *a* clique *of users in a social network (the structural predicate), who each have posted more than 10 messages in last 2 hours (the activity-based predicate).* Detecting such active subgraphs and analyzing them in real time can help with anomaly detection in phone call networks, advertisement targeting in social networks, malware detection in file download graphs, and so on. In Chapter 5, I present CASQD, a system for *continuous* detection and analysis of such *active subgraph pattern queries* over large dynamic graphs. Some of key challenges in executing such queries include: handling a wide variety of user-specified activities of interest, low selectivities of activity-based predicates and the resultant exponential search space, and high data rates often seen in these application domains. A key abstraction in CASQD is a notion called *graph-view*, which acts as an independence layer between the query language and the underlying physical representation of the graph and the active attributes. This abstraction is aimed at simplifying the query language, while empowering the query optimizer. Considering the balance between expressibility (i.e., patterns that cover many real-world use cases) and optimizability of such patterns, I primarily focus on efficient continuous detection of the *active regular structures* (specifically, active cliques, active stars, and active bi-cliques). I develop a series of optimization techniques including model-based neighborhood explorations, lazy evaluation of the activity predicates, neighborhood-based search space pruning, and others, for efficient query evaluation. I perform a thorough comparative study of the execution strategies under various settings, and show that our system is capable of achieving end-to-end throughputs over 800k/s using a single, powerful machine.

**Summary of Contributions:** My main contributions in this work are are the following:

- I build an in-memory, general purpose, user-extensible active subgraph detection and analysis framework that can achieve high event ingestion throughputs for large dynamic graphs. My work is the first systematic approach that looks into active subgraph pattern matching queries.

- I introduce an abstraction called *graph-view* which act as an independence layer between the query language and the underlying representation of the data graph and activity-based attributes. We also introduce the notion of *activity schema* akin to the relational data model. Activity schema encodes the signatures of different types of events that comprise the input event stream.

- I design a powerful task model. A task in our system is specified using an active subgraph pattern as well as a user-defined analytical task. The query optimizer enables efficient identification of active subgraphs, while the user-defined task can perform arbitrarily complex analysis tasks.

- I develop a set of novel optimization techniques to detect the active primitives (i.e., active stars, active cliques, and active bipartite cliques efficiently).

- I perform extensive evaluation using both real and synthetic data to show the efficacy of our techniques. Using a single, powerful machine, I could achieve a throughput of 800K events per second.

## 1.6 Organization of the Dissertation

In Chapter 2, I review the related work in connection to my dissertation research. In Chapter 3, I present more details about my work on building distributed graph databases with adaptive replication. In Chapter 4, I focus on EAGr, which is an extension of our system to support ego-centric aggregation. In Chapter 5, I discuss CASQD, our system for continuous identification and analysis of active subgraph patterns on dynamic graphs. Finally, I conclude and outline possible future directions for my work in Chapter 6.

# Chapter 2

# Related Work

In this chapter, I review the work related to my dissertation research. I divide the discussion under four different categories: (a) systems and techniques for managing large graphs, (b) data stream management, (c) query languages for graphs, and (d) subgraph pattern matching. As far as my dissertation is concerned all these aspects are tightly coupled, however, traditionally, they have largely been studied independently.

## 2.1 Systems and Techniques for Managing Large Graphs

There is an increasing interest in large-scale, graph data management, with several new commercial and open-source systems being developed for that purpose. The examples include many specialized graph databases (e.g., Titan [19], Neo4j [14], OrientDB [16], HyperGraphDB [11], InfiniteGraph [12], FlockDB [6], GraphBase [9] etc.) as well as graph execution engines (e.g., Apache Giraph [8], GraphLab [106], GraphX [73], Mizan [92], GPS [128], Pegasus [90], Ligra [132], Galois [121], etc.). Recently several researchers have also investigated the possibility of executing graph analysis tasks using a relational database system (e.g., Vertexica [88], GRAIL [64], Aster Graph Analytics [136]).

All these systems show a lot of variety in how they store the data, what kinds of tasks they support, what languages they use, etc. In a recent survey, Angles [31] has performed a systematic study of many of these aspects with a focus on comparing the specialized graph databases. On the other hand, in another recent study, Han et al. [79] have focused on comparing the various graph execution engines.

Although some of these systems support updating of the graph data, most of the systems primarily support query processing over static snapshots of the graphs. Among these different classes of systems, the specialized graph databases are primarily optimized for point queries like reachability, keyword search, subgraph pattern matching, shortest path queries, etc. On the other hand, most of the graph execution engines operate in a batch processing mode, and focus on efficient execution of a single large task (e.g., PageRank, All-pairs shortest path, belief propagation, motif finding, etc.) in a distributed fashion.

There has also been some recent efforts on extending the static data and query model to support interactive and continuous analytics. Kineogrph [57] uses a graph snapshot-based mechanism for analyzing content and structure (example tasks include user ranking, controversial topic detection in Twitter, approximate shortest paths etc.). GraphInc [49] focuses on facilitating real-time graph mining operations, while Trinity [40] is designed for executing subgraph pattern matching queries on large dynamic graphs.

**Distributed Graph Data Management:** Now we look at some of the work that focuses on distributed management of graph data, as they are closely related to the work presented in this dissertation.

The standard way to deal with large graphs is to partition the data across multiple machines, and execute queries in a distributed fashion. The most well-known algorithms to distribute a graph are *min-cut partitioning* [1] and its variations like *weighted min-cut partitioning*, and *k-way balanced min-cut partitioning*. The general goal of these algorithms is to divide the graph in such a way that leads to the minimum number of cross-partition edges, thereby reducing the latency and the bandwidth consumption for distributed querying. These techniques however, mainly focus on the structure of the graph, and can't naturally model the content generated in the network.

Recently, there has been work in the area of graph partitioning that looks beyond the graph structure, and makes use of additional information available like node co-access patterns, identified partition overlaps, content update patterns of the nodes, and so on. COSI [45] is a framework that can partition very large social networks according to query history. Pujol et al. [123] developed a partitioning and replication middleware, called SPAR, where the replication is used to achieve data locality as well as fault tolerance. Specifically, they used *local semantics* where the distributed neighbors of a node are guaranteed to be locally replicated, in order to bring down the execution latencies. They also point out a seemingly surprising fact that, minimizing the number of cross-partition edges does not minimize the number of replicas required. In another work, Yang et al. [149] introduced a two-level partition management architecture for executing queries on dy-

[1] `https://en.wikipedia.org/wiki/Minimum_cut`

namic graphs. Their system maintains complementary primary partitions and dynamic secondary partitions, with the invariant that a cross-partition edge in the primary partition would be maintained as an internal edge in the complementary partition, thereby bringing down the average latency.

Graph partitioning could be especially challenging for real-world graphs due to the presence of many high-degree nodes (also known as power-law graphs). Using edge-cut-based metrics for the min-cut partitioning algorithms in such situations might lead to too many cut edges. PowerGraph [72] proposes an alternative way of handling this by partitioning the graphs using the notion of vertex-cut, where the high degree nodes are logically cut and placed on multiple partitions. These vertices are then kept consistent via replication thereby simulating local visibility for the neighbors in each of those partitions, reducing the number of cut edges due to high-degree nodes drastically.

Once the graph has been partitioned, an important design concern in distributed graph management is how the data is kept consistent and available across machines, especially when the queries have real-time latency requirements and need to access data spanning multiple partitions. Two important notions in this context are: (a) *active/eager/push-on-change* replication and (b) *passive/lazy/pull-on-demand* replication. In the push-on-change model, all the remote replicas (i.e., all remote locations that might access the data from the concerned location) are kept up-to-date. Such an approach however suffers from very high, unnecessary communication to keep the replicas up-to-date. In contrast, in the pull-on-demand model, required data is fetched from remote locations at query time (Facebook reportedly uses this model). This reduces communication overhead, however, may lead to higher execution latencies.

There is also work on monitoring read/write frequencies and making replication decisions based on them. For instance, Wolfson et al. propose the adaptive data replication approach [146], where they adapt the replica placement based on the read/write patterns. Their algorithm is primarily designed for a tree communication network, but can also handle general graph topologies. In a subsequent work on this topic Kadambi et al. [89] used a similar approach for geographically distributed data management system. However, the problems encountered in dynamic graph data management are significantly different and require us to develop new approaches. The primary reason for this is that the data items are not read individually, but are always accessed together in some structured manner, and exploiting that structure is essential in achieving good performance.

Similar decisions about pushing vs pulling also arise in content distribution networks (CDNs), or publish-subscribe systems. In a related work, Silberstein et al. [133] modeled each node in the graph as both a producer of information and a consumer of information, and developed techniques to decide between push or pull. However, that work has primarily considered a situation where the producers and consumers are distinct from each other, and usually far apart in the communication network. The reciprocal relationships observed in graph data change the optimization problems quite significantly. Secondly, that work has focused purely on the information delivery problem, and the techniques cannot be directly used for executing other types of queries.

## 2.2 Data Stream Management

Unlike one-time queries, for a continuous queries (also sometimes called a *standing* queries), the systems are expected to keep the answers up-to-date as new data arrives. Such queries are often observed in *publish-subscribe systems* or *complex event processing systems* where a centralized system is typically in charge of ingesting the published data from the data sources, and deciding if any updates need to be sent to the subscribers (whose subscriptions can be viewed as continuous queries) or if any events need to be generated. Efficiently supporting continuous queries over rapidly changing data streams has seen much work over the last decade. Some of the most well-know works include NiagaraCQ [118], TelegraphCQ [108], STREAM [114], Aurora [22], and HiFi [61]. Several SQL extensions have also been proposed to express continuous queries over data streams. Similarly, languages have also been designed for specifying event patterns to be matched against data streams (e.g., SASE [77]).

Continuous query processing also bears strong resemblance to materialized view maintenance, an area that has also seen much work [76, 94, 131, 41, 75]. The key difference between the two research areas has been that: continuous query processing systems are designed to simultaneously support large numbers of relatively simple queries over highly dynamic data, whereas view maintenance techniques usually focus on a small number (usually just one) of more complex queries. The former also tend to build intermediate data structures like *predicate indexes* to efficiently identify the queries whose results are affected by new updates. Another line of work has focused on development of *one-pass* algorithms that can incrementally compute some quantities of interest over very large volumes of data (e.g., statistics or aggregates) while using very small amounts of memory. The books by Aggarwal [23], Muthukrishnan [117], and Garofalakis et al. [69] provide an extensive survey of the research done in this area.

While stream data management has seen much work in the past two decades, more recently, we have seen a renewed interest in building large-scale distributed stream processing systems, with many of the contributions coming from the industry. The primary reason being the proliferation of interconnected smart devices producing a large amount of data, and the requirement to build products and services to keep the users engaged. Some of the most prominent systems includes Spark Streaming [152], Heron [97], Dataflow [26], Kinesis [13], MillWheel [25], Apache S4 [2], Apache Flink [1], Trill [53], and Pulsar [17].

**Aggregation Queries on Streams:** As far as our work on ego-centric aggregation queries is concerned, the work on evaluating continuous aggregate queries over data streams is most closely related [33, 28, 95, 145, 36]. However, much of the work on computing streaming aggregates is done in a relational setting. Arasu et al. presented an SQL-based continuous query language that supports user-defined aggregates and rely on mapping between streams and relations for efficient implementation [33]. Krishnamurthy et al. [95] and Wang et al. [145] presented techniques for sharing work across different queries with different sliding windows. Al Moakar et al. [28] generalized that and proposed a 3-level aggregation overlay.

However, computation sharing techniques proposed in a relational setting is fundamentally different than the sharing opportunities in ego-centric aggregate computation over graphs.Further, most of the prior work on evaluating continuous aggregates has only considered the *all-push* model of query evaluation.

There has also been much work on aggregate computation in sensor networks and distributed databases, some of which has considered sharing of partial aggregates [143, 135, 107, 85, 122]. However the primary optimization goal in that work has been minimizing communication cost during distributed execution, and hence the techniques developed are quite different.

There is another line of work that deals with aggregation over multiple streams mainly to support monitoring style queries [85, 122, 153]. Most of those systems either support distributed monitoring queries by aggregating over multiple streams, or provide efficient techniques to evaluate multiple aggregate queries on a single stream [96, 145, 109, 36].

Almost all of the works discussed so far either focus on static analysis of graphs or aim to do efficient distribution of data items in order to perform distributed monitoring, and hence are not really suitable for answering on-demand neighborhood-based aggregation queries for large dynamic graphs.

## 2.3 Graph Query Languages

One of the major challenges in building general-purpose graph data management systems is developing an expressive and succinct query/task specification language. Developing such a language for graph data is especially challenging due to the unstructured nature of the graph data which allows for various different types of queries like traversal queries, reachability queries, centrality measure queries, etc. to be specified. For this reason, popular relational query languages were deemed unsuitable for graph-structured data. However, more recently, there have been several proposals for graph query languages (recently, Peter Wood presented a survey of many such languages [148], none has gained wide acceptance; perhaps the only exception is the SPARQL query language, but the use of that query language has been largely limited to RDF datasets. Many extensions of SPARQL has also been proposed in recent times, e.g., Streaming SPARQL [43], Continuous SPARQL(C-SPARQL) [38], EP-SPARQL [32]. Although these languages focus on RDF data streams, they could be adapted to use in social networks by treating social network data as RDF data. Some of the key extensions to SPARQL include the use of "REGISTER QUERY" keyword to specify a continuous query that should be evaluated continuously, and a way to specify a window over the stream (using keyword "RANGE"),

Another line of work attempted to generalize XPath. For example, Mozafari et al. [116] propose XSeq, an extension to XPath to express both sequential and Kleene-closure expressions for XML streams. A key challenge here is that XPath is designed to operate on tree-structured data, not graph-structured data. However, recent theoretical work suggests that it may be possible to use XPath for specifying graph queries [103].

In recent years, Datalog [126] has been shown to be an effective centerpiece in enabling declarative specification in a range of domains including networking [105], data cleaning [34], machine learning [47], and social network analysis [115, 130, 113]. Compared to the above two languages (i.e., SPARQL and XSeq), Datalog is more amenable to be extended to support a large class of complex aggregate queries (e.g., global queries like *PageRank* computation, *shortest paths*, etc., can be specified using *recursion*). Apart from Datalog, there have also been several other proposals for declarative (e.g., Green-Marl [84], HelP [129]) frameworks. Singh et al. developed a query language [80] for finding sub-graph patterns using a graph as a query primitive. Two other query languages that have been made popular by recent open-source graph databases are Cypher [5] (used in Neo4j [14] graph database) and Gremlin [10]. Gremlin is a graph traversal language over the property graph data model , and has been adopted by several open-source systems.

While most of the specialized graph databases support some form of a declarative query language, many graphs processing engines provide a programmatic access to the data graphs in languages such as Java, Python, Scala, C++, etc. In a majority of such graph engines, the most commonly adopted programming model is vertex-centric programming framework where the computation is defined as a program that is to be run for each vertex in parallel. Giraph, GraphLab, GraphX, and several other systems have adopted this model. Apart from these large-scale systems, there also exists graph libraries such as NetworkX [15], Blueprints [3] and SNAP [18] that provide a rich set of implementations of graph-based measures and graph theoretic algorithms.

## 2.4 Subgraph Pattern Matching on Dynamic Graphs

Detecting occurrences of interesting events on dynamic graph-structured data is of much interest. The most common way of expressing such events is through specifying subgraph patterns. Subgraph pattern matching has been a very well researched problem, with a large number of prior techniques and approaches. A comprehensive comparison of different well-known methods can be found in [99]. Apart from the algorithm development, there has also been effort to build large-scale systems (e.g., Arabesque [140], TrinityCloud [138]) for performing subgraph pattern matching. However, these algorithms and systems primarily target a static setting and unsuitable for continuous pattern matching on dynamic graphs.

There are several researchers who have looked at the problem of continuous detection of subgraph pattern matching queries over streaming graph data. However, most of them mainly focus on changes in graph structure and on categorical attributes, and can't handle the window-based active attributes that we are interested in. Moreover, most of the existing works focus on tree-structured patterns and would be too expensive for detecting structures like large stars, cliques or bipartite cliques. Song et al. have looked at the problem of event pattern matching over graph streams [137]. The fact that their queries have additional timing order constraints (i.e., *happened before relationships* in events) along with the graph structure, makes them quite different from ours. Choudhury et al. [59] have investigated a selectivity-driven approach for continuous pattern detection on streaming graphs. Their approach is to perform continuous pattern mining by decomposing the main query based on the selectivity of the node attributes, matching the individual components, and finally performing a multi-way join. Because they only look at path-based patterns and static node attributes (i.e., they only consider structural dynamicity), the problems they face are much different than ours. In a recent work [68], Gao et al. have proposed a vertex-centric approach for continuous pattern matching for dynamic graphs using Apache Giraph. Their approach focuses on decomposing the query graph into a DAG and then using the DAG to define message transition rules for each of the nodes in the Giraph framework. The DAGs could be seen as exploration plans, to be traversed by Giraph, one edge at a time. While their approach is a nice fit for Giraph's programming model, such a framework might not be usable when there exist strict latency requirements. Moreover, their approach is more suitable for tree patterns, and may require a very large number of steps to detect structures like cliques and bicliques. Additionally, using a framework like Giraph would be very restrictive to the types of optimization we need to perform in our case. Another work by Chen et al. [144] used an index-based technique for continuous subgraph pattern matching. For each vertex in the graph, the index, named node-neighbor tree, encodes all the simple paths of length $l$ rooted at the vertex. Such an index-based technique, however, could be too expensive to maintain as the graph structure changes.

Zhao et al. [65] have also looked at a similar problem where they use an incremental algorithm to compute the changes to the existing pattern match rather than expensive re-computation. Gao et al. in their work on LBSN [67] define two behaviour models: (i) a historical model (HM) and (ii) a social-historical model (SHM) and compare their performance in order to provide meaningful location related services. Chandramouli et al. [55] have looked into the problem of generating on-demand recommendations that require real-time incremental processing.

Chapter 3

# Distributed Graph Data Management with Adaptive Replication

In this chapter, we discuss our work on scalably managing large, distributed graphs for supporting real-time querying. We begin with a brief overview of our proposed system and discuss various design decisions that we have made (Section 3.1). We then discuss a key component of our system, the *replication manager* (Section 3.2), and present algorithms for making the replication decisions (Section 3.3). We then present a comprehensive experimental evaluation (Section 3.4).

## 3.1 Overview

We start with a recap of the underlying data model and discuss the high-level system architecture, and then briefly mention some of the key trade-offs relevant to our system and define a fairness criterion.

### 3.1.1 Data and Query Model

Recall that the data is represented as a graph $G(V, E, V_p, E_p)$ where $V$ is the set of all nodes and $E$ represents the set of all edges. To avoid confusion, we refer to the vertices of the graph as nodes, whereas we refer to the sites (machines) across which the graph is partitioned as either sites or *partitions*. The graph is distributed across multiple partitions, and each node has information about its cross-partition neighbors. A $write$ on a node is simply updating or appending node information, whereas a $read$ on a node is reading the information stored in that node. In a traversal we are performing reads on all the nodes that are part of the traversal.

A typical query in a social network context could be: *For a person $x$, find all of his friends who have attended Stanford Business School and who have a friend from South Africa*. For a query like this, we have to start from node $x$, visit all its neighbors to check which of them had attended Stanford business school, and then for all those friends, visit their neighbors till we find a neighbor matching the predicate.

Another type of query we may want to support is a subgraph pattern matching query. An example of such a query could be: *Given a citation network, find all the papers that discuss graph pattern matching, and are a result of collaboration between researchers from Stanford and MIT*. To execute such a complex query, we would typically need to build an index on the relevant attributes (e.g., paper abstracts) to quickly find the candidate nodes that may be of interest and then traverse the neighborhoods of those candidate nodes to find the matches.

Although our system is aimed at supporting different types of queries flexibly, for ease of illustration, we primarily focus on a special type of query prevalent in the social network domain, namely, the *"fetch updates from all my neighbors"* query. Given a specific node $x$, this query requires us to traverse the neighborhood of $x$ and find the latest writes that have been made in that neighborhood. In today's social networks with a large fraction of nodes having a non-trivial number of neighbors, these queries are very hard to scale. This problem is also called the "feed delivery" problem, and also arises commonly in publish-subscribe networks [134]. We revisit the issue of generality below when we describe the system architecture and formulate the optimization problems.

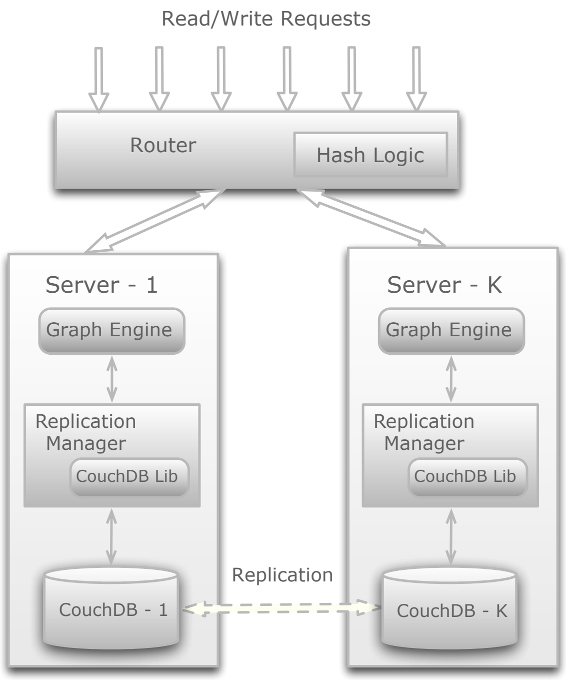


Figure 3.1: System Architecture

## 3.1.2 Architecture

Figure 3.1 shows the high-level architecture of our system comprising of *replication manager* and other supporting modules. The key components of the system are as follows:

**Router:** The router is responsible for routing the incoming read and write requests to the appropriate sites. We assume that for all the requests, we are given a *start* node *id*, i.e., the identifier of the node from which the traversal should begin. For complex queries (e.g., subgraph pattern match queries), we assume that an external query processor is responsible for generating the set of start node candidates, perhaps through use of an index. We do not discuss that component further in the chapter.

The router uses a *hash partitioning-based* scheme to partition the nodes across the sites. this may result in very large edge-cuts compared to using min-cut-based partitioning, but is still the preferred method of partitioning in practice for several reasons: (1) hash-based partitioning typically results in balanced workload across the sites and is much more robust to *flash traffic*, (2) the routing process is not only highly efficient, but it can be infinitely parallelized by adding more routers, (3) there is no complicated logic involved in assigning new nodes to partitions, and (4) for a given node, we only need to list its neighbors' ids and not their locations. A scheme that tries to optimize the edge-cuts or other metrics requires large routing tables to be maintained [62, 139], which increase the routing cost and are hard to keep consistent if the router is replicated. Further, in dynamic networks like social networks, the partitioning may become suboptimal very quickly.

However, we note that the algorithms and the techniques that we develop in the rest of the chapter are completely independent of the choice of the partitioning logic.

**Storage and Replication:** We use the Apache CouchDB [4] key-value store as our backend storage to store all the information related to a node. CouchDB is a schema-free document-oriented data management system, which stores *documents* that can contain any number of fields and attachments. The schema-free nature of CouchDB makes it ideal for storing heterogeneous graph data, where different nodes may have different attributes, and the amount of information stored about a node may have a very wide range (we typically would wish to store historical information as well). There are several reasons we chose CouchDB over other key-value stores. Many of the other key-value stores (e.g., HBase) do not give us sufficient control over where the data is placed, whereas CouchDB is intended primarily as a single-server product.

More importantly, CouchDB has excellent replication support, optimized for minimizing network bandwidth, for selectively replicating data across multiple sites. The documents or databases to be replicated can be specified between any pair of CouchDB servers, and CouchDB will keep the replicas up-to-date by sending only the changes that have occurred. Further, for each database that is replicated, we can specify whether the replication should be "continuous" (i.e., push-based) or not (i.e., pull-based), and these decisions can be changed easily.

As above, the techniques we develop in the rest of the chapter are largely independent of this choice, and we can replace CouchDB with another key-value store. However, in that case, depending on the features supported by the key-value store, we may have to write a layer on top of the key-value store to support adaptive pull-based or push-based replication.

**Graph Engine:** Graph Engine is the module responsible for parsing, scheduling, and executing the queries. For this work we present a simplistic graph engine with a focus on *fetch all updates from friends* style queries in social networks. During the execution of a query, the graph engine talks to the replication manager when it needs to read or update data for a node in the system. The replication manager (which we discuss next) deals with any distributed communication if required. For example, all the neighbors of a node $n$ (residing in $server$-$K$) might not be present in $server$-$K$, and a read query on $n$ would require us to fetch data from neighbor nodes in remote servers. The replication manager fetches the data for all such distributed neighbors through pre-computed replication decisions (i.e., either by *lazy* or *eager* replication).

**Replication Manager:** The replication manager is the most important component of our system, and is in charge of making the replication decisions to minimize the network bandwidth and query latencies, and enforce the fairness requirement (discussed below). Unlike the fixed replication strategies adopted in many real systems (i.e., push-on-change or pull-on-demand models discussed in Section 2.1), the replication manager in our system adopts a more flexible model. It monitors the node read-write frequencies, which are themselves stored along with the nodes in the CouchDB server, and uses them to make cost-based replication decisions (i.e., what is replicated, and whether it is active/eager/push-on-change or passive/lazy/pull-on-demand). Moreover, the decisions are computed in a decentralized fashion – each replication manager can make the decisions for the graph node in its partition autonomously. It implements those decisions by appropriately instructing the CouchDB servers. We discuss the specifics of the replication manager in more detail in the next section.

### 3.1.3 Trade-offs and Requirements

In this section we briefly reiterate some of the key trade-offs and desired properties of a dynamic graph data management system in the context of the problem we aim to solve. We also define the fairness criterion and discuss its implications.

**Network Bandwidth:** It is desirable from a distributed system that the communication overhead be minimized. As discussed earlier, there are two factors at play here: query latencies and replica maintenance. In most of the real-time applications today, read/write operations are latency-critical and failing to keep those under acceptable limits may lead to demise of such applications To ensure low-latency query execution, we need to minimize the number of cross-partition traversals, and if there is no natural partitioning of the data, then we must use active replication for that purpose. However, in a dynamically evolving graph, the cost of keeping the replicas up-to-date may exceed the benefits of replication. Furthermore, both the write and read access patterns may change dynamically, and different policies may be best at different times. Hence, we must not only choose replicas carefully to ensure low-latency operations, but we should also try to adapt the replication decisions in response to changing access patterns.

**Balanced Load:** Balanced load across the sites is another very important metric. Balanced load ensures that no resource is under or over-utilized, thereby bringing down the overall system cost and increasing the efficiency of the system. Apart from minimizing network bandwidth, it is expected that the network load will also be balanced for maximum utilization of the system bandwidth. Since our data graph is hash partitioned across sites, it is fair to assume that the network load will be evenly balanced. But, even then we have to make sure that the replication algorithm doesn't interfere with the balance. Secondly, balancing system load, i.e., the load of a site is equally important. Key resources that can get hit hard in such scenario are the CPU and the main memory. Once again, hash partitioning naturally helps us with guaranteeing balanced load, however skewed replication decisions may lead to load imbalance.

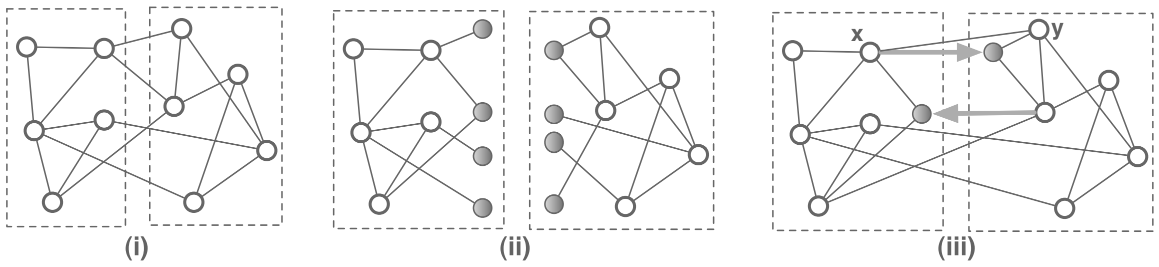


Figure 3.2: (i) An example graph partitioned across two partitions; (ii) Maintaining *local semantics* [125] requires replicating 80% of the nodes; (iii) we can guarantee fairness with $\tau = \frac{2}{3}$ by replicating just two nodes

**Fairness Criterion:** Ideally we would like that all queries are executed with very low latencies, which in our context, translates to minimizing the number of pulls that are needed to gather information needed to answer a query. For "fetch neighbors' updates" queries, this translates into minimizing the number of neighbors that are not present locally. As discussed in Section 2.1, Pujol et al. [125] presented a solution to this problem where they guarantee that all the neighbors of a node are replicated locally, and the replicas are kept up-to-date (they called this *local semantics*). This guarantees that no pulls are required to execute the query. However, the number of replicas needed to do this in a densely connected graph can be very high. Figure 3.2 shows an instance of this where we need to replicate 8 out of 10 nodes to guarantee local semantics for all the partitions. The cost of maintaining such replicas is likely to overwhelm the system. This may be okay in a highly over-provisioned system (we would expect Facebook to be able to do this), but in most cases, the cost of additional resources required may be prohibitive.

Instead, we advocate a more conservative approach here where we attempt to ensure that all queries can make some progress locally, and the query latencies are largely uniform across the nodes of the graph. Such uniformity is especially critical when we are using read/write frequencies to make replication decisions, because the nodes with low read frequencies tend to have their neighbors not replicated, and queries that start at such nodes suffer from high latencies. We encapsulate this desired property using what we call a *fairness criterion*. Given a $\tau \leq 1$, we require that for all nodes in the graph, at least a $\tau$ fraction of its neighbors are present or replicated locally. In case of "fetch neighbors' updates" queries, this allows us to return some answers to the query while waiting for the information from the neighbors that are not present locally. For other queries, the fairness requirement helps in making progress on the queries, but the effect is harder to quantify precisely. As we can see in Figure 3.2(c), we need to replicate 2 nodes to guarantee a fairness of 0.8 for the example graph.

**Provide Cushion for Flash Traffic:** Flash traffic is simply a flood of unexpected read/write requests issued to the system within a small period of time. For example, events like earthquake could cause a deluge of tweets to be posted and consumed on Twitter within seconds. In such situation, any system that does aggressive active replication (e.g., if we were maintaining local semantics) could suffer significantly, as the bandwidth requirement will increase suddenly. we do not optimize for flash traffic directly in this

| Notation | Description |
|---|---|
| $\Pi = \{P_1, \cdots, P_l\}$ | Set of all partitions |
| $R_{ijk}$ | Replication table corresponding to the cluster $C_{ij}$ and partition $P_k$ |
| $C_{ij}$ | $j^{th}$ cluster of $P_i$ |
| $\langle C_{ij}, P_k \rangle$ | a cluster-partition pair, $i \neq k$ |
| $H$ | Cost of a push message |
| $L$ | Cost of a pull message |
| $\omega(n_i, t)$ | Write frequency of $n_i$ at time interval $t$ |
| $\omega(C_{ij}, t)$ | Cumulative write frequency of $C_{ij}$ |
| $\rho(n_i, t)$ | Read frequencies for $n_i$ |
| $\rho(P_k, C_{ij})$ | Cumulative read frequency for $P_k$ w.r.t. $C_{ij}$ |

Table 3.1: Notation used in the chapter

work. However, conservative replication and hash-based partitioning helps in alleviating these problems in our system.

## 3.2 Replication Manager

In this section, we describe the design of our replication manager in detail. We begin with a brief overview and describe the key operating steps. We then discuss each of the steps in detail.

### 3.2.1 Overview

We define some notation that we use in the rest of the chapter. As before, let $G(V, E, V_p, E_p)$ denote the data graph, let $\Pi = \{P_1, \cdots, P_l\}$ denote the disjoint partitions created by hash partitioning, i.e., $\forall i : P_i \subset V$ and $\cap_i P_i = \phi$. Each of the partitions $P_i$ itself is divided into a number of *clusters*, $C_{i1}, \cdots, C_{ik}$ (we assume the same number of clusters across the partitions for clarity). All replication decisions are made at the granularity of a cluster, i.e., the replication decisions for all nodes within a cluster are identical (this does not however mean that the nodes are replicated as a group – if a node has no edges to any node in another partition, we never replicate it to that partition). We discuss both the rationale for the clustering, and our approach to doing it below.

**Implementing the Replication Decisions:** As we have discussed before, we use CouchDB as our backend store and to implement the basic replication logic itself. In CouchDB, we can specify a *table* (called *database* in CouchDB) to be replicated between two CouchDB servers. Our replication logic is implemented on top of this as follows. For every cluster $C_{ij} \in P_i$, for every other partition $P_k$ with which it has at least one edge, we create a table, $R_{ijk}$, and ask it to be replicated to the CouchDB server corresponding to $P_k$. We then copy the relevant contents from $C_{ij}$ to be replicated to that table $R_{ijk}$. Note that, we usually do not copy the entire information associated with a graph node, but only the information that would be of interest in answering the query (e.g., the latest updates, rather than the history of all updates).

If the decision for the cluster-partition pair $\langle C_{ij}, P_k \rangle$ is a "push" decision, then we ask the CouchDB server to keep this table *continuously* replicated (by setting an appropriate flag). Otherwise, the table has to be manually *sync*-ed. We discuss the impact of this design decision on the overall performance of the system in detail in Section 3.4.4. We periodically delete old entries from $R_{ijk}$ to keep its size manageable.

We also need to maintain metadata in partition $P_k$ recording which clusters are pushed, and which clusters are not (consulting $R_{ijk}$ alone is not sufficient since partial contents of a node may exist in $R_{ijk}$ even if it is not actively replicated). There are two pieces of information that we maintain: first, we globally replicate the information about which clusters are replicated to which partitions. Since the number of clusters is typically small, the size of this metadata is not significant. Further, the replication decisions are not changed very frequently, and so keeping this information up-to-date does not impose a significant cost. Secondly, for each node, we maintain the cluster membership for all its cross-partition neighbors. This coupled with the cluster replication information enables us to deduce whether a cross-partition neighbor is actively replicated (pushed) or not. Note that, the cluster membership information is largely static, and is not expected to change frequently. If we were to instead explicitly maintain the information about whether a cross-partition neighbor is replicated with each node, the cost of changing the replication decisions would be prohibitive.

**How and When to Make the Replication Decisions:** We present our algorithms for making the replication decisions in the next section. Here we present a brief overview.

- The key information that we use in making the replication decisions are the read/write access patterns for different nodes. We maintain this information with the nodes at a fine granularity, by maintaining two histograms for each node. As an example, for a social network, we would wish to maintain histograms spanning a day, and we may capture information at 5-minute granularities (giving us a total of 120 entries). We use the histogram as a predictive model for future node access patterns. However, more sophisticated predictive models could be plugged in instead. We discuss this further in Section 3.2.2.

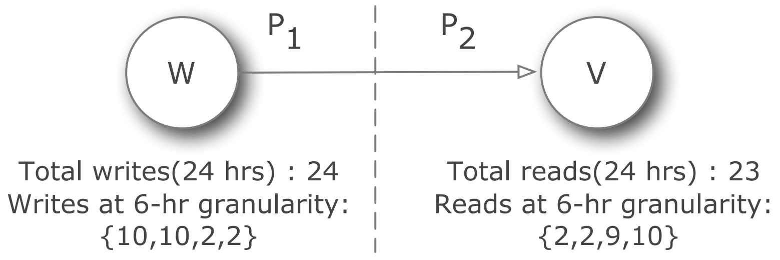
- For every cluster-partition pair $\langle C_{ij}, P_j \rangle$, we analyze the aggregate read/write histograms of $C_{ij}$ and $P_k$ to choose the *switch points*, i.e., the times at which we should change the decision for replicating $C_{ij}$ to $P_k$. As we discuss in the next section, this is actually not optimal since it overestimates the number of pull messages required. However, not only can we do this very efficiently (we present a linear-time optimal algorithm), but we can also make the decisions independently for each cluster-partition pair affording us significant more flexibility.

- When the replication decision for a cluster-partition pair $\langle C_{ij}, P_k \rangle$ is changed from push to pull, we need to ensure that the fairness criterion for the nodes in $P_k$ is not violated. We could attempt to do a joint optimization of all the decisions involving $P_k$ to ensure that it does not happen. However, the cost of doing that would be prohibitive, and further the decisions can no longer be made in a decentralized fashion. Instead we reactively address this problem by heuristically adjusting some of the decisions for $P_k$ to guarantee fairness.

In the rest of the section, we elaborate on the motivation behind monitoring access patterns and our clustering technique.

### 3.2.2 Monitoring Access Patterns

Many approaches have been proposed in the past for making replication decisions based on the node read/write frequencies to minimize the network communication while decreasing query latencies. Here we present an approach to exploit *periodic patterns* in the read/write accesses, often seen in applications like social networks [39, 71], to further reduce the communication costs. We illustrate this through a simple example shown in Figure 3.3. Here for two nodes $w$ and $v$ that are connected to each other but are in different partitions, we have that over the course of the day, $w$ is predicted to be updated 24 times, and whereas $v$ is predicted to be read (causing a read on $w$) 23 times. Assuming the push and pull costs are identical, we would expect the decision of whether to push the updates to $w$ to the partition containing $v$ or not to be largely immaterial. However, when we look at fine granularity access patterns, we can see that the two nodes are active at different times of the day, and we can exploit that to significantly reduce the total communication cost, by having $v$ pull the updates from $w$ during the first half of the day, and having $w$ push the updates to $v$ in the second half of the day. In the context of human-activity centered networks like social networks, we expect such patterns to be ubiquitous in practice.



P1
P2
W
V
Total writes(24 hrs) : 24
Writes at 6-hr granularity:
{10,10,2,2}
Total reads(24 hrs) : 23
Reads at 6-hr granularity:
{2,2,9,10}

Figure 3.3: Illustrating benefits of fine-grained decision making: Making decisions at 6-hr granularity will result in a total cost of 8 instead of 23.

To fully exploit such patterns, we collect fine granularity information about the node access patterns. Specifically, for each node we maintain two equi-width histograms, one that captures the update activity, and one that captures the read activity. Both of these histograms are maintained along with the node information in the CouchDB server. We will assume that the histogram spans 24 hours in our discussion; in general,we can either learn an appropriate period, or set it based on the application.We use these histograms as a predictive model for the node activity in future.

For a node $n_i$,we denote by $\omega(n_i, t)$ the predicted update frequency for that node during the time interval starting at $t$ (recall that the width of the histogram buckets is fixed and hence we omit it from the notation). We denote cumulative write frequency for all nodes in a cluster $C_{ij}$ for that time interval by $\omega(C_{ij}, t)$.We similarly define $\rho(n_i, t)$ to denote the read frequency for $n_i$. Finally, we denote by $\rho(P_k, C_{ij}, t)$ the cumulative read frequency for $P_k$ with respect to the cluster $C_{ij}$ (i.e., the number of reads in $P_k$ that require access to a node in $C_{ij}$).

### 3.2.3 Clustering

As we discussed above, we cluster all the nodes in a partition into multiple clusters, and make replication decisions for the cluster as a unit. However, we note that this does not mean that all the nodes in the cluster are replicated as a unit. For a given node $n$, if it does not have a neighbor in a partition $P_j$, then it will never be replicated at that partition. Clustering is a critical component of our overall framework for several reasons.

First, since we would like to be able to switch the replication decisions frequently to exploit the fine-grained read/write frequencies, the cost of changing these decisions must be sufficiently low. The major part of this cost is changing the appropriate metadata information as discussed above. By having a small number of clusters, we can reduce the number of required entries that need to be updated after a decision is changed. Second, clustering also helps us in reducing the cost of making the replication decisions itself, both because the number of decisions to be made is smaller, and also because the inputs to the optimization algorithm are smaller. Third, clustering helps us avoid *overfitting*. Fourth, clustering makes node addition/deletion easier to handle as we can change node's association to cluster transparently w.r.t. other system operations. By making decisions for clusters of nodes together, we are in essence averaging their frequency histograms, and that can help us in better handling the day-to-day variations in the read/write frequencies.

To ensure that clustering does not reduce the benefits of fine-grained monitoring, we create the clusters by grouping together the nodes that have similar write frequency histograms. More specifically, we treat the write frequency histogram as a vector, and use the standard *k-means* algorithm to the clustering. We discuss the impact of different choices of $k$ in our experimental evaluation.

We note that clustering is done offline, and we could use sampling techniques to do it more efficiently. When a new node is added to the system, we assign it to a random cluster first, and reconsider the decision for it after sufficient information has been collected for it.

## 3.3 Making Replication Decisions

In this section, we present our algorithms for making replication decisions. We assume that the clustering decisions are already made (using the *k-means* algorithm), and design techniques to make the cluster-level replication decisions. We begin with a formal problem definition, and analyze the complexity of the problem. We then present an optimal linear-time algorithm for making the replication decisions for a given cluster-partition pair in isolation ignoring the fairness requirement (as we discuss below, this is not an overall optimal since the decisions for the clusters on a single partition are coupled and cannot be made independently).We then present an algorithm for modifying the resulting solution to guarantee fairness.

### 3.3.1 Problem Definition

As before let $G(V, E, V_p, E_p)$ denote the data graph, $P_1, \cdots, P_l$ denote the hash partitioning of the graph, and let $C_{ij}$ denote the clusters. We assume that fine-grained read/write frequency histograms are provided as input. For the bucket that starts at $t$, we let $\omega(n_i, t), \omega(C_{ij}, t)$ denote write frequencies for $n_i$ and $C_{ij}$; $\rho(n_i, t)$ denote the read frequency for $n_i$; and , $\rho(P_k, C_{ij}, t)$ denote the cumulative read frequency for $P_k$ with respect to the cluster $C_{ij}$.

Next we elaborate on our cost model. We note that the total amount of information that needs to be transmitted across the network is independent of the replication decisions made, and depends only on the partitioning of the graph (which is itself fixed a priori). This is because: (1) the node updates are assumed to be append-only so waiting to send an update does not eliminate the need to send it, and (2) we cache all the information that is transmitted from one partition to the other partition. Further, even if these assumptions were not true, for small messages, the size of the payload usually does not impact the overall cost of sending the message significantly. Hence, our goal reduces to minimizing the number of messages that are needed. Let $H$ denote the cost of one push message sent because of a node update, and let $L$ denote the cost of a single pull message sent from one partition to the other. We allow $H$ and $L$ to be different from each other.

Given this, our optimization problem is to make the replication decisions for each cluster-partition pair for each time interval, so that the total communication cost is minimized and the fairness criterion is not violated for any node.

It is easy to capture the read/write frequencies at very fine granularities (e.g., at 5-minute granularity), however it would not be advisable to reconsider the replication decisions that frequently. We can choose when to make the replication decisions in a cost-based fashion (by somehow quantifying the cost of making the replication decisions into the problem formulation). However, the two costs are not directly comparable. Hence, for now, we assume that we have already chosen a coarser granularity at which to make these decisions (we evaluate the effect of this choice in our experimental evaluation).

## 3.3.2 Analysis

Figure 3.4(i) shows an example data graph partitioned across two partitions that we use to illustrate the challenges with solving this problem. We assume that the cluster size is set to 1 (i.e., each node is a cluster by itself). We omit the intra-partition edges, and also the time interval annotation for clarity. We consider the question of whether to replicate the clusters from $P_1$ to $P_2$, and use the write frequencies for the nodes in $P_1$, and the read frequencies for the nodes in $P_2$. We call a node in $P_1$ a writer node, and a node in $P_2$ a reader node.

Following prior work [134], one option is to make the replication decision for each pair of nodes, one writer and one reader, independently. Clearly that would be significantly suboptimal, since we ignore that there may be multiple readers connected to the

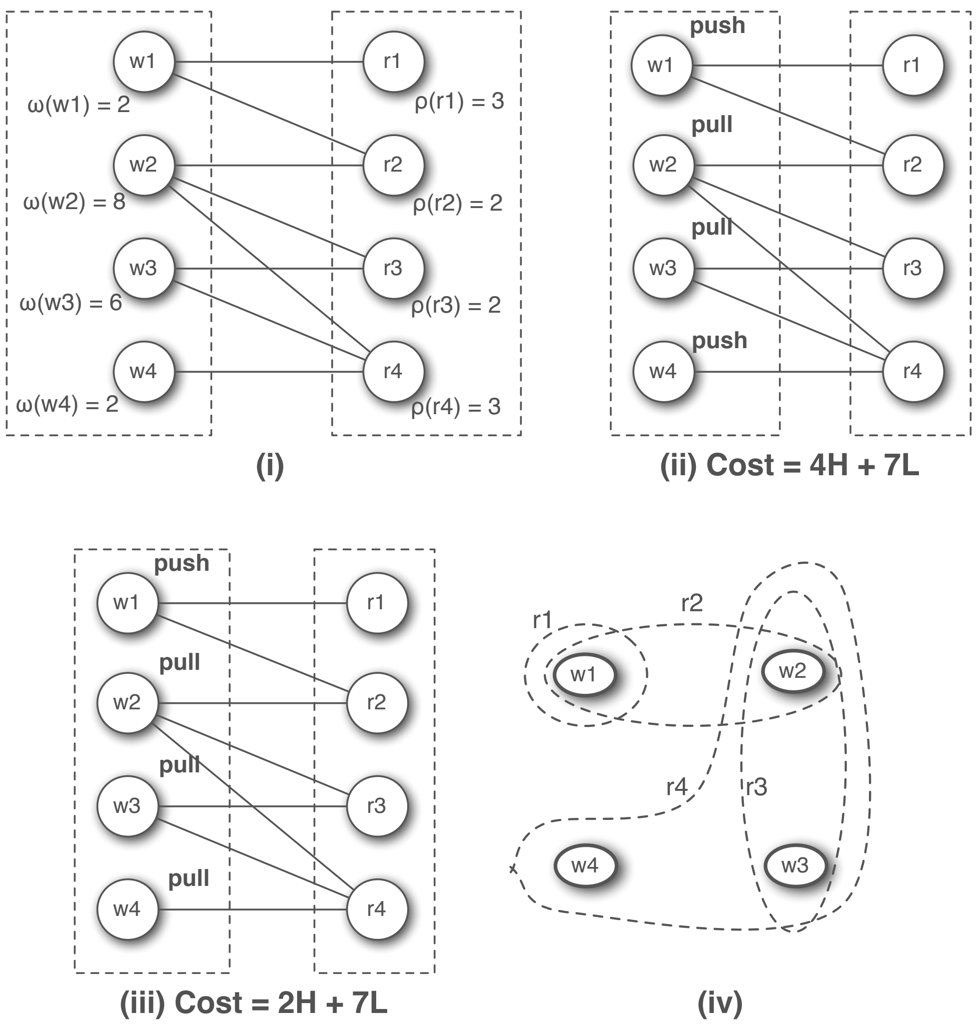


Figure 3.4: (i) An example instance where we consider whether to replicate the single-node clusters from the left partition to the right partition; (ii) Making decisions for each cluster-partition pair independently; (iii) Optimal decisions; (iv) Modeling the problem instance as a weighted hypergraph.

same writer. Instead, we can make the decision for each writer node in $P_1$ independently from the other writer nodes, by considering all reader nodes from $P_2$. In other words, we can make the decisions for each cluster-partition pair. Figure 3.4(ii) shows the resulting decisions. For example, we choose to push $w_1$ since the total read frequency of $r_1$ and $r_2$ exceeds its write frequency (here we assume that $H = L$).

These decisions are however suboptimal. This is because it is useless to replicate $w_4$

in the above instance without replicating $w_2$ and $w_3$, because of the node $r_4$. Since neither of $w_2$ and $w_3$ is replicated, when doing a query at node $r_4$, we will have to pull some information from $P_1$. We can collect the information from $w_4$ at the same time (recall that we only count the number of messages in our cost model – the total amount of data transmitted across the network is constant). Figure 3.4(iii) shows the optimal decisions.

As it turns out, it is possible to make these decisions optimally in polynomial time (note that we ignore the fairness criteria here). Figure 3.4(iv) shows another way to model this problem, where we turn the problem instance into a weighted hypergraph. The nodes of the hypergraph are the nodes in $P_1$, with the write frequencies used as weights. For each reader node, We add a hyperedge to this graph over the nodes that it is connected, and weight of the hyperedge is the read frequency of the node. Now, say a subset $S_1$ of the nodes in $P_1$ are replicated. Let $S_2$ denote the hyperedges that are completely covered by $S_1$, i.e., hyperedges that only contain nodes from $S_1$. Then, the total cost for these two partitions is:

$$\sum_{v \in S_1} \omega(v) + \sum_{u \notin S_2} \rho(u) = C + \sum_{v \in S_1} \omega(v) - \sum_{u \in S_2} \rho(u)$$

where $C = \sum \rho(u)$ is a constant. In other words, we pay the cost of one push message per node in $S_1$ and one pull message per node not in $S_2$. This problem is similar to the well-studied problem of finding the sub-hypergraph of a hypergraph with the maximum density (the standard density metric is $\sum_{u \in S_2} \rho(u) / \sum_{v \in S_1} \omega(v)$). We can use similar max-flow based techniques to solve our problem (in fact the above optimization goal is simpler), however we omit the details because we do not use such an algorithm in our system for several reasons. First, even though the problem can be solved in polynomial time [93, 70], the complexity of the algorithm is still quite high. This coupled with the fact that the size of the input is large (the number of hyperedges is equal to the number of nodes in $P_2$), that approach would be infeasible. We instead use a heuristic that we discuss below that greedily makes a local decision for each cluster-partition pair, significantly reducing both the input size and hence the overall complexity.

So far we have ignored the fairness criterion. For the two partitions $P_1$ and $P_2$ as above, the fairness criterion requires that, for every reader node in $P_2$, at least a $\tau$ fraction of its neighbors be replicated. The problem of finding the optimal replication decisions given a fairness requirement is unfortunately NP-Hard. Note that, when $\tau = 1$, this problem does not reduce to the problem considered by Pujol et al. [125] (who prove their partitioning problem to be NP-Hard). This is because they are trying solve the graph partitioning problem itself, to come up with a good partitioning of the graph. In our case, the solution for $\tau = 1$ is trivial – we must replicate every node into every partition that it is connected to (we call this the *all-push* solution in our experimental evaluation).

**Theorem 3.3.1.** *The problem of optimally replicating nodes to guarantee fairness is NP-Hard.*

*Proof.* We show a reduction from the *set cover* problem. In a set cover instance, we are given a collection of sets $S_1, \cdots, S_n$ over a universe $U = \{e_1, \cdots, e_m\}$ (i.e., $S_i \subseteq U$, and $\cup S_i = U$), and the goal is to find the smallest collection of sets such that every element in $U$ is contained in at least one of those sets. Given a set cover instance, we create an instance of our problem with two partitions as follows.

Following the above terminology, let $P_1$ be the partition that contains the writer nodes, and let $P_2$ be the partition that contains the reader nodes. For each set $S_i$, we add a writer node $w_i$ in $P_1$. For each element in the universe $e_j$, we create a reader node $r_j$ in $P_2$. We connect $w_i$ to $r_j$ if $e_j \in S_i$. Let $\tau$ be the fairness threshold. We connect each of $r_j$ to sufficient nodes in $P_2$ such that we are exactly one neighbor short of achieving fairness for $r_j$. For instance, if $\tau = 0.5$ and if $r_j$ is connected to 5 nodes in $P_1$, then we connect $r_j$ to 4 nodes in $P_2$ (adding dummy nodes if needed). In other words, for every node $r_j$, we need to replicate exactly one of its neighbors from $P_1$ to guarantee fairness.

Finally, we set the read frequencies for the nodes in $P_2$ to be very low, and write frequencies for the nodes in $P_1$ to be sufficiently high so that by default none of the nodes in $P_1$ will be replicated to $P_2$.

Given this setup, it is easy to see that choosing the minimum number of nodes from $P_1$ to push to guarantee fairness for all nodes in $P_2$ is identical to the set cover problem. □

### 3.3.3 Proposed Algorithm

In this section, we present our overall algorithm for making and changing replication decisions. The algorithm is decentralized by nature, and does not require global coordination (however, replication managers do need to communicate statistics and the replication decisions to other replication managers). The algorithm operates in two phases. In the first phase, at each partition $P_i$ and for each cluster $C_{ij}$ in it, we decide whether to replicate the cluster $C_{ij}$ at each of the other partitions, based purely on the read/write frequency histograms, and ignoring the fairness criterion. For efficiency, we do not make global decisions even within a site, and instead we make independent decisions for each cluster-partition pair $\langle C_{ij}, P_k \rangle, i \neq k$. Given the cumulative read/write frequency histograms for the cluster and the partition, we present a linear-time optimal algorithm to decide the switch points, i.e., the points at which the replication decisions should be switched.

In the second phase, run at each partition independently, we enforce fairness criterion for all the nodes at that partition by switching some replication decisions for clusters at other partitions from push to pull. As discussed above, this problem is NP-Hard in general, and we use a greedy heuristic based on the standard greedy heuristic for solving the set cover problem.

**Optimal Decisions for a Cluster-Partition Pair:** Next we present an optimal linear-time algorithm for making decisions of when to switch replication decisions for a given cluster-partition pair $\langle C_{ij}, P_k \rangle, i \neq k$. Let $\omega(C_{ij}, t)$ denote the write frequencies for $C_{ij}$ and $\rho(P_k, C_{ij}, t)$ denote the read frequencies for $P_k$ w.r.t. $C_{ij}$. We assume that we are given a constraint on the maximum number of times we are allowed to switch the replication decision, $C$ (without any such constraint, we would make a different replication decision for each time interval). We can instead assign a cost to making a replication decision, and optimize for the lower total cost – the algorithm below can be easily adapted to that effect.

Let there be $n$ buckets in the frequency histogram. For each bucket (i.e., each time interval), we compute the benefit of replicating $C_{ij}$ over doing a pull from $P_k$. For time interval $t$, this is computed as:

$$b_t = \rho(P_k, C_{ij}, t) \times L - \omega(C_{ij}, t) \times H$$

Thus we have $n$ numbers, denoted $b_1, \cdots, b_n$, that represent the benefit of a push over a pull for the corresponding intervals. Note that some of these numbers may be negative – if all of the numbers are positive, then we would always push $C_{ij}$ to $P_k$.

We first compress this sequence of numbers by coalescing the entries with the same sign together. In other words, if we have a contiguous sequence of positive numbers, we will replace it with a single number that is the sum of those numbers. Similarly, we would coalesce any sequences of negative numbers. The rationale behind this is that, we would never want to switch replication decisions in the middle of such a sequence.

Let $s_1, \cdots, s_m$ denote the resulting sequence of alternating positive and negative numbers. Let $opt(C', push, i)$ denote the optimal cost for the subproblem $s_i, \cdots, s_m$ using at most $C'$ switches and assuming that the decision for the time period corresponding to $s_i$ is a PUSH. We similarly define $opt(C', pull, i)$. Then we can see that:

$$opt(C', push, i) = s_i + max\{opt(C', push, i+1), opt(C'-1, pull, i+1)\}$$

In essence, we check both possibilities for $s_{i+1}$, PUSH or PULL, and choose the best of the two. Similarly,

$$opt(C', pull, i) = -s_i + max\{opt(C', pull, i+1), opt(C'-1, push, i+1)\}$$

Here we have to use $-s_i$ since $s_i$ is benefit of doing push and we are doing a pull in the time period corresponding to $s_i$. The base case of the recursion is when $C = 0$ at which point we simply return the sum of the remaining items, possibly negated. The computational complexity of the algorithm can be seen to $O(nC)$.

**Guaranteeing Fairness:** Finally, we discuss how we ensure that the fairness requirement is satisfied for all nodes. The replication manager at each partition runs this algorithm independently of the other partitions, and may change some of the replication decisions for clusters at other partitions with respect to that partition.

Since the problem is NP-Hard, we develop a heuristic based on the standard greedy heuristic for set cover. For a partition $P_k$, let $\Gamma_k$ denote the nodes for which fairness guarantee is not satisfied. Let $C_{ij}$ be a cluster at another partition $P_i$ which is *not* replicated at $P_k$, i.e., the decision for $\langle C_{ij}, P_k \rangle$ is a pull. Then let $benefit_{ijk}$ denote the total benefit of changing the decision for that cluster-partition pair. This is computed as:

$$benefit_{ijk} = \sum_{v \in \Gamma_k} |nei(v) \cap C_{ij}| - remaining(v)$$

where $nei(v)$ denote the set of neighbors of $v$, and $remaining(v)$ denote the number of neighbors after replicating which the fairness criterion would be satisfied for $v$. Further, let $cost_{ijk}$ be the cost of switching the decision for $C_{ij}$ from a pull to a push. We greedily choose the cluster to be replicated that has the highest $benefit_{ijk}/cost_{ijk}$ ratio, and continue until the fairness criterion is met for all nodes.

## 3.4 Evaluation

In this section, we present a comprehensive experimental evaluation using our prototype system. Lacking real datasets with sufficient detail, we constructed a social network graph based on a commonly used social network model, and also constructed a trace of user activity on a social network by gathering user activity data on Twitter and extrapolating. We focus on the "fetch updates from all my neighbors" queries which are the most common class of queries in such networks. As discussed in Section 2, our system is built on top of CouchDB, and we used Amazon EC2 to run our experiments. Our key findings can be summarized as follows:

- Our hybrid replication approach results in significant savings in the network communication cost over the baseline approaches.

- The granularity at which we make push/pull decisions plays an important role in determining how much savings we can obtain.

- The hash-based partitioning scheme results in balanced network and CPU load in our system.

- Our fairness guarantee reduces the average number of pulls required to answer read queries.

We begin with describing the dataset, and the experimental setup.

### 3.4.1 Dataset

We constructed our data set to match the workload of a social network. We have used a *preferential attachment* model to generate the data graph which has been shown to model a social network very well [50, 37, 29, 100]. The network is generated by adding one node at a time, and the new nodes are preferentially connected to the existing nodes with higher degrees (i.e., the probability of connecting to an existing node depends on the degree of that node). Most of our experiments were run on a social network containing 1.8 million nodes, and approximately 18 million edges (generated with the preferential attachment factor of 10).

The second key component of our simulated dataset is the user activity patterns. We chose 100 Twitter users with sufficient number of tweets and downloaded their tweets to get their access trace. This trace only gives us the write frequencies of the nodes. In our experiment, we have assumed that the read frequency of a node is linearly related to its write frequency. In reality this linear factor might be different for different users; however, we assume a constant read to write ratio for all nodes. From the access traces, we created write frequency histograms and linearly scaled those to get the read frequency histograms. Once we had the pool of histograms, we assigned them to the nodes in the network. Motivated by recent work on modeling user activity on Twitter [74], we used the following assignment process. We assigned histograms to the graph nodes one at a time. When considering which histogram to assign to a node, we check the histograms already assigned to the other nodes in the same partition, and find the histogram that has been assigned to the largest number of nodes in that partition. We assign the same histogram to the node under consideration with 50% probability, otherwise we choose any one of the remaining histograms with equal probability.

However, we do not use these assigned histograms directly. For each user, we instead randomized the assigned histogram by generating a trace by treating the histogram as a probability distribution, and then building a histogram on the generated trace. This ensures sufficient diversity in the user histograms across the network.

### 3.4.2 Experimental Setup

We ran our experiments on Amazon EC2 infrastructure using 7 EC2 instances (1 instance is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, 1 core, and 1.7G of memory). We used 1 instance to play back the write and read traces (i.e., to send updates and queries to the system), and rest of the instances to host the graph. Each server had a CouchDB server and a copy of replication manager running. As we noted earlier, our network has 1.8 million nodes, which translated into about 300,000 nodes per partition. We used a trace containing a total of approximately 25 million events (reads and writes), corresponding to a single day. The default write to read ratio in the query workload was set to 0.2 (i.e., there are 5 read queries for every update). For each of the experiments, we ran the trace against the system (after selecting the appropriate replication approach), and computed the total network messages. For most of our plots, we plot the *average number of network messages per site*.

We compared three approaches: (1) *all-pull*, where we do not do any replication, (2) *all-push*, where the nodes are replicated sufficient to guarantee no pulls would be needed (i.e., local semantics), and (3) *hybrid*, our hybrid approach. Unless otherwise specified, the number of clusters at each partition was set to 6.

### 3.4.3 Evaluation Metric

Our main evaluation metric is the amount of network communication in terms of messages [146] exchanged across all the servers as logged by our replication middleware. As we discussed earlier, once a data item is replicated to a partition, it is cached and will not be transferred again. Because of this, the amount of data transfer across the servers is independent of the replication decisions that are made (we can easily modify the cost functions in our algorithms to account for this if desired). Hence for most of the results, we report the total number of push and pull messages (i.e., we assume $H = L = 1$).

For a *push* decision, we use continuous replication of CouchDB and there is a message involved every time the corresponding graph node is updated. However, the way we count the number of *pull* messages is slightly different, and reflects the constraints imposed by CouchDB and our setup. In fact, this results in a significant underestimation in the number of pull messages as some of our experiments also illustrate.

The way a *pull* works in our system is that, the replication manager asks CouchDB to *sync* the appropriate replication table (see Section 3.1). However since the replication tables correspond to clusters, all updates to that cluster are pulled from the cluster's home partition. To amortize the cost of this, we enforce a minimum gap between two pulls corresponding to the same cluster by using a *timeout*. In other words, if a cluster has been recently pulled, we do not pull it again until the timeout expires. In our experimental evaluation, the timeout is set to 800ms, so the data can be at most 800ms stale (which is reasonable in a social network application). We further discuss the rationale in Section 3.4.4.

## 3.4.4 Results

**Impact of Histogram Granularity:** We start with a set of experiments to verify our hypothesis that by making decisions in a fine-grained manner can result in significant savings. Figure 3.5 shows the results for this experiment. Here we varied the histogram granularity from 1/2 hour to 12 hours, and counted the total number of messages that were needed. The *all-pull* and *all-push* approaches are unaffected by this, however, we can see that by making decisions at the finest granularity, i.e., every 1/2 hour, resulted in almost 33% savings over coarse-grained decisions. This validates our hypothesis that we can exploit the user activity patterns to reduce the network communication costs.

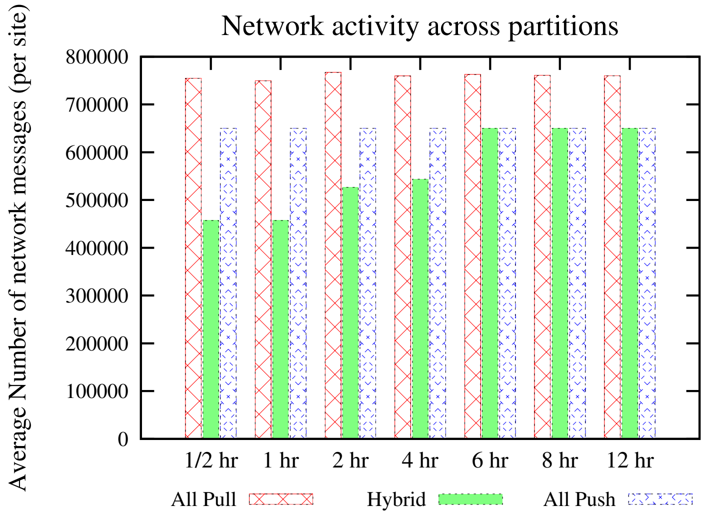


Figure 3.5: Making fine-grained decisions can result in almost 33% savings over coarse-grained decisions

We also note that overall our default workload is read-heavy, and hence all-push solution is usually better than all-pull solution (although it results in higher memory consumption). As we move toward coarse-grained histograms, we observed that most of the replication decisions became push. But our algorithm is able to exploit the diversity in the access patterns when making decisions at finer granularities to achieve significant savings.

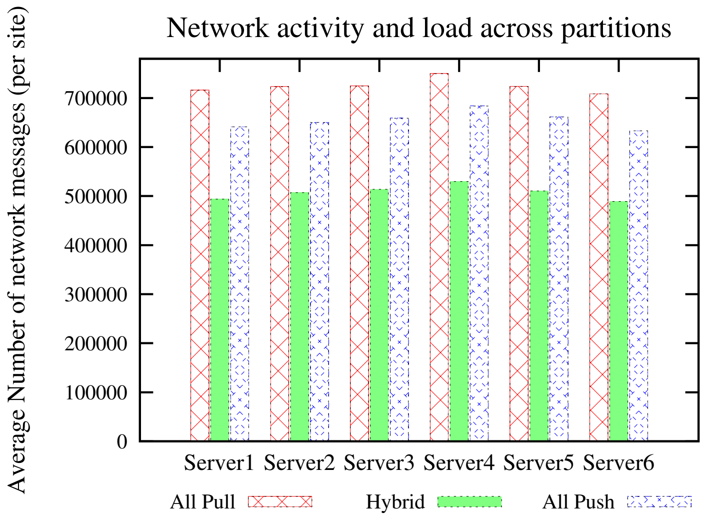


Figure 3.6: Hash partitioning results in almost uniform load across the partitions

**Bandwidth Consumption and Network Load:** Figure 3.6 shows the total network communication across the servers. For each server, we aggregated the network communication resulting from writes happening to the corresponding partition, and the reads directed to the partition. As we can see, all the approaches resulted in fairly balanced load across the partitions, with *hybrid* achieving almost 20% savings over *all-push* in all cases. This can be attributed to the hash partitioning scheme that we use, which guarantees that the overall read and write distributions across the partitions are largely uniform.

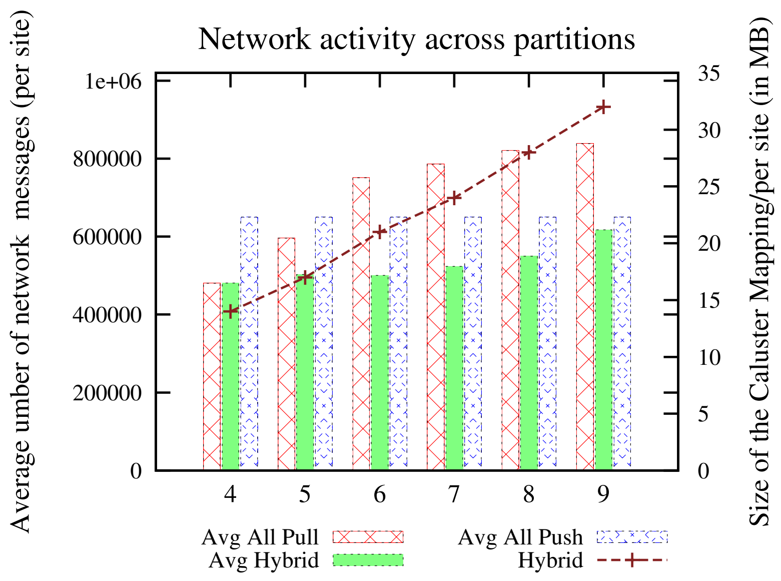


Figure 3.7: Varying the number of clusters

**Varying the Number of Clusters:** Next we study the effect of $k$, the number of clusters in each partition. We varied the number of clusters from 4 to 9, and we show the results in Figure 3.7. Along with the network communication costs (plotted on the *left* y-axis), we also plot the size of the cluster mapping table, the metadata that is needed to decide which of a node's neighbors are replicated (on the *right* y-axis). As we can see, as the number of clusters increases the size of the cluster mapping table increases as expected. What is somewhat counter-intuitive is that the total communication cost also increases beyond 6. We expect that with large numbers of clusters, we can make more fine-grained decisions which should aid in reducing the total network communication cost.

The reason for this is somewhat subtle, and has to do with the way *pulls* are handled in our system. Recall that a single pull actually syncs an entire cluster, i.e., it propagates all updates for a single cluster from the home partition to the partition making the pull.

Thus increasing the cluster sizes results in an decrease in the number of pulls that are required. We expect that if we were counting the number of pulls explicitly, that would result in the behavior as expected (however, in a read-heavy workload, that would imply that the all-push solution would always be better by a margin).

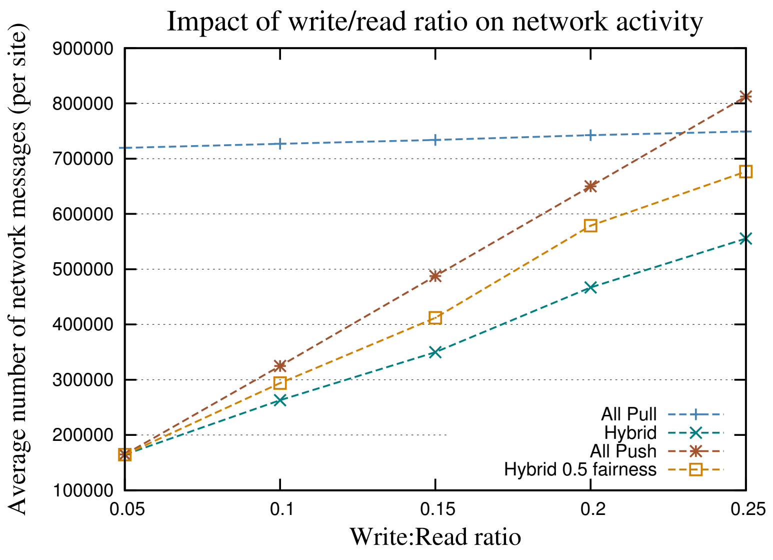


Figure 3.8: Increasing the write:read ratio results in favoring pulls over pushes

**Varying Write-Read Ratio:** We examine how the replication techniques perform for workloads that have different mixes of reads and writes. We simply varied the read/write ratio of the workload and calculated the average cost in terms of total number of communications, incurred by the three approaches. For hybrid, we also plot the costs when the fairness threshold $\tau$ is set to 0.5. Figure 3.8 shows the results of this experiment. We kept the number of reads more or less constant, and varied the number of writes. As we can see, our approach did consistently better than the other two. Since the number of reads were almost constant, the performance of the *all-pull* approach does not change significantly. However, the costs of the other three approaches increase almost linearly, with base *hybrid* showing the best performance. In fact, with low write/read ratio, the hybrid approach is almost equivalent to *all-push*, but as the write frequency increases, pull decisions are favored, and hybrid starts performing much better than *all-push*. With fairness threshold set to $0.5$, the *hybrid* approach does worse than the basic *hybrid* approach, because in order to guarantee fairness, it is forced to do more active replication than optimal.

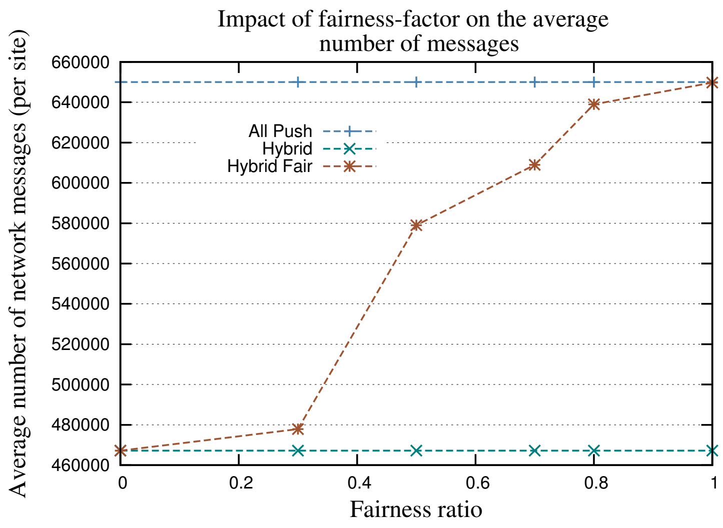


Figure 3.9: Impact of fairness factor on network communication

**Varying the Fairness Threshold:** Next we investigate the impact of fairness threshold (Section 3.1.3). We vary the threshold from 0 (in which case, the approach is the basic hybrid approach) to 1 (equivalent to *all-push*). In general, increasing fairness threshold will result in more push decisions than is optimal. Figure 3.9 illustrates this point where we plot the average network communication cost as before. As we can see, increasing the fairness threshold results in a move toward all-push solution. We do not plot the cost of *all-pull* in this case for clarity.

Figure 3.10 shows the latencies of read queries for the different approaches. As we expect, the latency is lowest for the *all-push* solution, with an absolute value of about

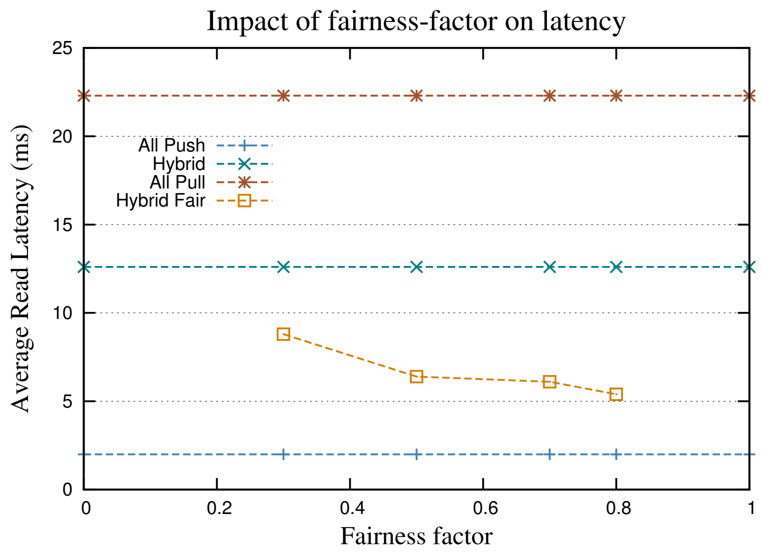


Figure 3.10: Impact of the fairness factor on read query latencies

*2ms*. The cost for the *all-pull* approach is relatively quite high, almost *22ms*. The hybrid approach is somewhere in-between, with somewhat higher latencies than the *all-push* approach but with, as we saw earlier, significantly lower network activity. As expected, when the fairness factor is varied, the average latency drops reaching *2ms* with $\tau = 1$.

This set of experiments not only validates our assertion that fairness threshold is an important novel consideration for highly dynamic graph databases, but also shows the efficiency of our system at processing queries with very low absolute latencies.

**Varying the Graph Density:** We also investigated how our system performs as we increase the density of the graph. We changed the preferential attachment factor (PA) in the graph generator to create graph with same number of nodes but with different densities. Here by density we mean the average number of neighbors of a node. We changed the attachment factor from 5 to 20 and analyzed the performance of different replication strategies. This results in varying the average degree of the graph from about 5 to about 20.

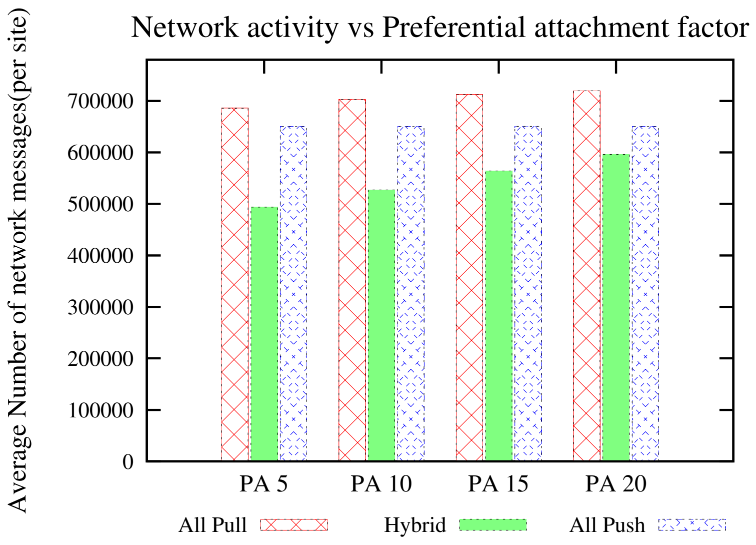


Figure 3.11: Increasing the density of the graph increases the number of reads, and decreases the opportunities to exploit the read/write skew

Figure 3.11 shows that our hybrid replication techniques continues to perform better than *all-pull* and *all-push* approaches. One point to note here is, as we increase the density of the graph the performance of our hybrid techniques degrades, and moves towards *all-push*. The reason for this is, though the write frequency of the nodes remains the same, the cumulative read frequency increases on an average. Thus our already read-heavy workload becomes even more skewed towards reads, resulting a preference for the *all-push* approach. We note that the cost of the *all-pull* approach increases as expected with the increase in the graph density.

**Effect of the Pull Timeout:** Finally we discuss the $timeout$ used for $pull$ in our system. We note that we don't have such a timeout for $push$. It might seem to be an unfair comparison, but given the read-heavy workload it is a natural system design decision. Intuitively the $timeout$ captures the difference in cost between a $push$ and a $pull$ (we could have also captured this by using appropriate values for $H$ and $L$). However we also

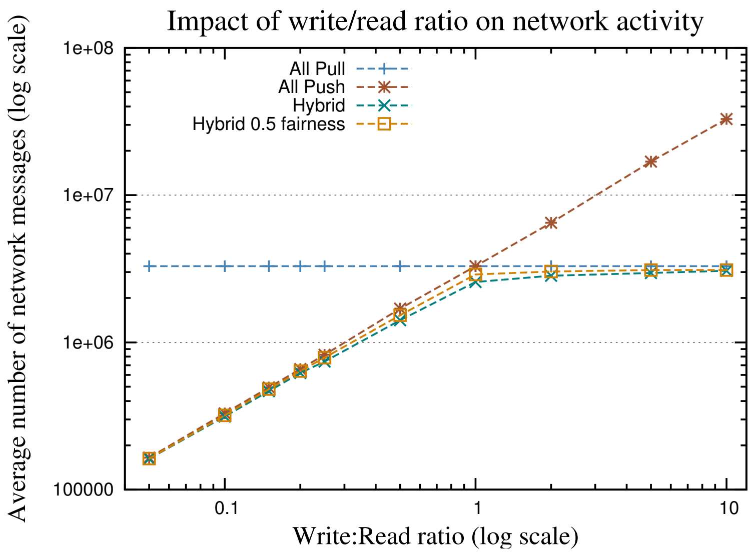


Figure 3.12: Comparing the techniques without push/pull timeout (note that both axes are in log-scale)

performed a set of experiments with no $timeout$ for $pull$ to further evaluate the uneven use of $timeout$ for $push$ and $pull$. Figure 3.12 shows how different schemes perform with no $timeout$ parameter for either $push$ or $pull$ and with the read:write ratio varying from 20:1 to 1:10. We can see that $push$ is favored when the workload is read-heavy, and as the number of writes increases, $pull$ is preferred. Our proposed hybrid approach performs better than the best of those two, with the maximum benefit observed when the workload is balanced in terms of reads and writes – the maximum benefit we see is almost 25% (note that both the axes are in log-scale).

Chapter 4

# Ego-centric Aggregation on Large Dynamic Graphs

In this chapter we describe EAGr, our framework to provide support for ego-centric aggregation queries on large dynamic graphs. We begin with a brief overview of the problem by reviewing the data and the query model (Section 4.1). Then we present the details of our proposed aggregation framework, including the API for supporting user-defined aggregates (Section 4.1.2). Next we analyze the optimization problem of constructing an *overlay graph* (Section 4.2), and propose several efficient heuristics that can scale to very large graphs. Following that, we discuss how we make the dataflow (push/pull) decisions in order to minimize data movement in the overlay graph (Section 4.3). Then we describe our experimental setup and present a comprehensive experimental evaluation in Section 4.4.

## 4.1 Overview

We start with a brief overview of the underlying data and query model, followed by an outline of our proposed aggregation framework.

### 4.1.1 Data and Query Model

**Data model:** As before, $\mathcal{G}(V, E, V_p, E_p)$ denotes the underlying connection graph, with $V$ and $E$ denoting the sets of nodes and edges respectively. Also, as before $A_{\mathcal{G}}$ denotes the data stream produced by the network. For simplicity, in this chapter, we assume that $A_{\mathcal{G}}$ is only consists of five types of events, i.e., four types of structural events (addition/deletion of nodes/edges) and one type of content event. We further assume that the content streams are homogeneous, i.e., all updates are of the same type or refer to the same attribute. It is straightforward to relax both these assumptions. Also, as discussed in Section 1.3.1, we will call any content update by a node a *write*, and user consumption of he results of an aggregate query a *read*.

Various types of continuous queries have been studied in data streams or publish-subscriber systems literature. However, in most of the that prior work, the producers of the data (i.e., writers) and the consumers of the data (i.e., readers) are distinct from each other. In our case, a node in the graph acts both as a writer and a reader. Hence, for clarity of description, when referring to a node $v$ in the rest of the chapter, we often denote its role in the context using a subscript – $v_w$ (similarly, $v_r$) denoting the node as a writer (reader).

**Query model:** In our system an ego-centric aggregate query is specified by four parameters: $\langle \mathcal{F}, w, \mathcal{N}, pred \rangle$, where $\mathcal{F}$ denotes the aggregate function to be computed, $w$ denotes a sliding window over the content data streams, $\mathcal{N}$ denotes the neighborhood selection function (i.e., $\mathcal{N}(v)$ forms the input list to be aggregated for each $v$) , and $pred$ selects a subset of $V$ for which the aggregate must be computed (i.e., $\mathcal{F}$ would be computed for all nodes for which $pred(v)$ is *true*). Following the data streams literature, $w$ may be a time-based sliding window or a tuple-based sliding window; in the former case, we are given a time interval $T$, and the updates that arrive within the last $T$ time are of interest, whereas in the latter case, we are given a number $c$, and the last $c$ updates are of interest. The query may be specified to be a *continuous* query or a *quasi-continuous* query. For a continuous query, the query results must be kept up-to-date as new updates arrive, whereas for a quasi-continuous query, the query result for a node $v$ is only needed when a user requests it (we call this a *read on* $v$); in the latter case, pre-computation may

| Notation | Description |
|---|---|
| $\mathcal{N}()$ | Neighborhood selection function |
| $\mathcal{F}()$ | Aggregate function |
| *write* on $v$ | An update to node $v$'s content |
| *read* on $v$ | A user request to retrieve query result at $v$, i.e., $\mathcal{F}(\mathcal{N}(v))$ |
| $B_{\mathcal{G}}(V', E')$ | Bipartite directed writer/reader graph: for each node $v \in \mathcal{G}(V, E, V_p, E_p)$, it contains two nodes $v_w$ (writer) and $v_r$ (reader), with edges going from writers to readers |
| $O_{\mathcal{G}}(V'', E'')$ | Overlay Graph |
| $\mathcal{I}(ovl)$ | Set of writers aggregated by overlay node $ovl$ |
| $w(v)$ | write frequency of node $v$ |
| $r(v)$ | read (query) frequency of node $v$ |
| $f_h(v)$ | push frequency of node $v$ in an overlay |
| $f_l(v)$ | pull frequency of node $v$ in an overlay |

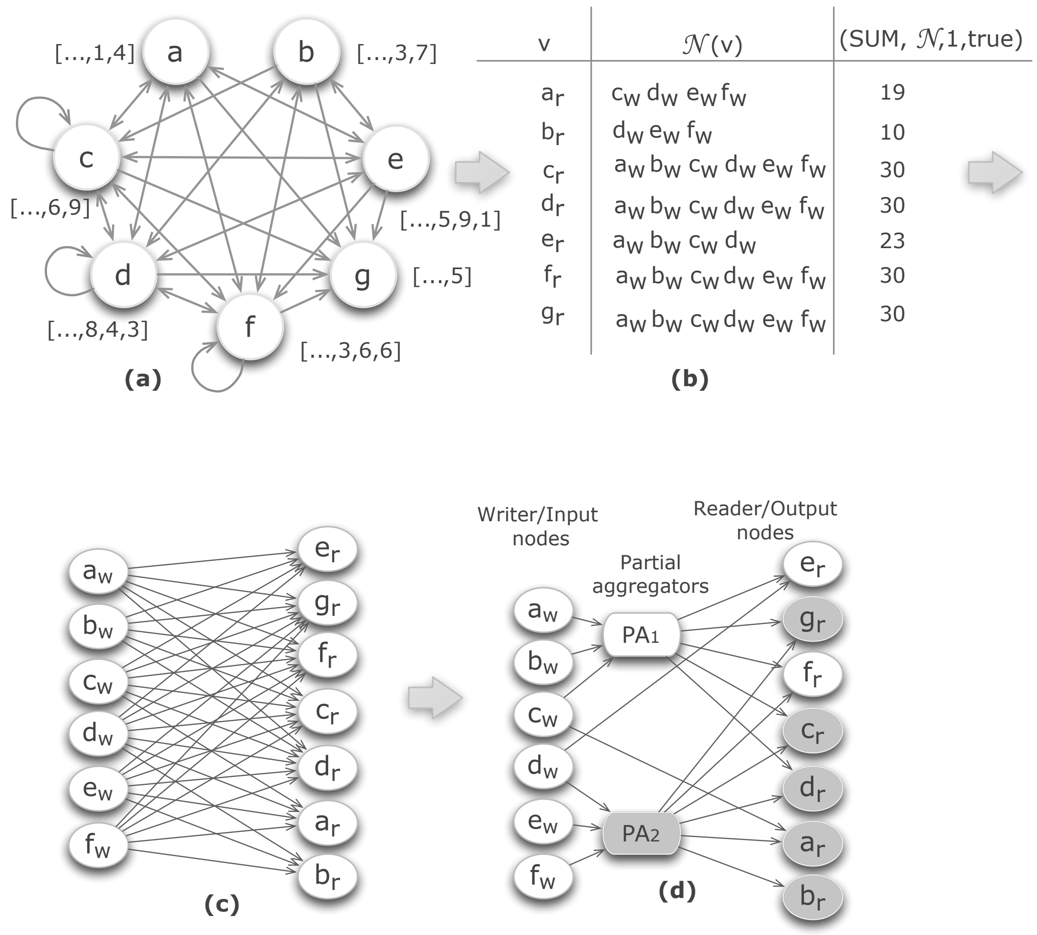
Table 4.1: Notation used in the chapter

be done to reduce not only the user latencies but also the total computational effort.

Since our approach is based on pre-computation and maintenance of partial aggregates, we assume that the aggregate function (and $\mathcal{N}$) are pre-specified. In some cases, it is possible to share the intermediate data structures and partial aggregates for simultaneous evaluation of different aggregates; we do not consider that option further in this chapter.

Our system supports a set of built-in aggregate functions like *sum, max, min, top-k, etc.*, and we allow the user to define arbitrary aggregation functions (Section 4.1.2.3). Our system treats $\mathcal{F}$ as a blackbox, but the user may optionally specify whether the aggregation function is *duplicate-insensitive* and/or whether it supports efficient *subtraction* (Section 4.2.1), and that information will be used to further optimize the computation.

**Example** Figure 4.1 shows an example instance of this problem. Figure 4.1(a) depicts the data graph. $\mathcal{N}(x)$ is defined to be $\{y|y \rightarrow x\}$ (note that, all edges are not bidirectional). The numbers in the square brackets denote individual content streams. For example, there have been two *recent writes* on node $a$ with values $1$ and $4$. The query is $\langle \textsc{Sum}, c = 1, \mathcal{N}, v \in V \rangle$, which states that for each node $v \in V$, the most recent values written by nodes in $\mathcal{N}(v)$ need to be aggregated using $\textsc{Sum}$. Figure 4.1(b) enumerates $\mathcal{N}(v)$ for each $v$. The last column of Figure 4.1(b) shows the results of the *read* queries on each node. For example, here $\mathcal{N}(a)$ evaluates to $\{c, d, e, f\}$, and a *read* query on $a$ returns: $(9) + (3) + (1) + (6) = 19$. Figure 4.1(c) represents the corresponding directed bipartite graph $B_{\mathcal{G}}$ where nodes are duplicated and divided based on their roles; a node might or might not play both the roles.



| v | $\mathcal{N}$(v) | (SUM, $\mathcal{N}$,1,true) |
|---|---|---|
| $a_r$ | $c_w$ $d_w$ $e_w$ $f_w$ | 19 |
| $b_r$ | $d_w$ $e_w$ $f_w$ | 10 |
| $c_r$ | $a_w$ $b_w$ $c_w$ $d_w$ $e_w$ $f_w$ | 30 |
| $d_r$ | $a_w$ $b_w$ $c_w$ $d_w$ $e_w$ $f_w$ | 30 |
| $e_r$ | $a_w$ $b_w$ $c_w$ $d_w$ | 23 |
| $f_r$ | $a_w$ $b_w$ $c_w$ $d_w$ $e_w$ $f_w$ | 30 |
| $g_r$ | $a_w$ $b_w$ $c_w$ $d_w$ $e_w$ $f_w$ | 30 |

Figure 4.1: (a) An example data graph, (b) $\mathcal{N}(v)$ and SUM aggregates for each $v$, (c) Bipartite representation of the graph, i.e, $B_{\mathcal{G}}$ (note, $g$ does not form input to any reader), (d) An overlay graph (shaded nodes indicate *pull* decisions, unshaded ones indicate *push*).

### 4.1.2 Proposed Aggregation Framework

In this section, we describe our proposed framework to support different types of ego-centric aggregate queries. We begin with explaining the notion of an *aggregation overlay graph* and key rationale behind it. We then discuss the execution model and some of the key implementation issues.

#### 4.1.2.1 Aggregation Overlay Graph

Aggregation overlay graph is a pre-compiled data structure built for a given ego-centric aggregate query, that enables sharing of partial aggregates, selective pre-computation, partial pre-computation, and low-overhead query execution. Given a data graph $\mathcal{G}(V, E, V_p, E_p)$ and a query $\langle \mathcal{F}, w, \mathcal{N}, pred \rangle$, we denote an aggregation overlay graph for them by $O_{\mathcal{G}}(V'', E'')$.

There are three types of nodes in an overlay graph: (1) the *writer* nodes, denoted by subscript $_{-w}$, one for each node in the underlying graph that is *generating* data, (2) the *reader* nodes, denoted by subscript $_{-r}$, one for each node in $V$ that satisfies $pred$, and (3) the *partial aggregation* nodes (also called *intermediate* nodes). We use the term *aggregation* node to refer to either a reader node or a partial aggregation node, since both of those may perform aggregation. In Figure 4.1(d), $PA_1$ and $PA_2$ are two partial aggregation nodes that are introduced after analyzing the structure of the network and the query. $PA_1$ corresponds to a partial aggregator that aggregates the inputs $a_w, b_w$, and $c_w$, and serves $e_r$, $g_r$, $f_r$, $c_r$, and $d_r$.

For correctness, there can only be one (directed) path from a writer to a reader in an overlay graph (to avoid duplicate contributions from that writer to the aggregate computed for that reader). However, there are two exceptions to this. First, this is not an issue with the so-called *duplicate-insensitive* aggregates like MAX, MIN, and UNIQUE. We exploit this by constructing overlays that allow such multiple paths if it leads to smaller overlays (in most cases, we observed that to be the case).

Second, we allow an overlay to contain what we call *negative* edges to “subtract” such duplicate contributions. A negative edge from a node $u$ to an aggregation node $v$ indicates that the input from $u$ should be "subtracted" (i.e., its contribution removed) from the aggregate result computed by $v$. Such edges should only be used when the "subtraction" operation is efficiently computable. Although negative edges may appear to lead to wasted work, in practice, adding negative edges (where permissible) can actually lead to significant improvements in the total throughput. We discuss this issue further in Section 4.2.1.

The overlay graph also encodes pre-computation decisions (also called *dataflow decisions*). Each node in the overlay graph is annotated either *pull* or *push*. If a node is annotated *push*, the partial aggregate that it computes is always kept up-to-date as new updates arrive. The writer nodes are always annotated *push*. For an aggregation node to be annotated *push*, all its input nodes must also be annotated *push*. Analogously, if a node is annotated *pull*, all the nodes downstream of it must also be annotated *pull*. In Figure 4.1(d), the push and pull decisions are shown with unshaded and shaded nodes respectively. This overlay graph fully pre-computes the query results for nodes $e_r$ and $f_r$ (thus leading to low latencies for those queries); on the other hand, a read on node $g_r$ will incur a high latency since the computation will be done fully on demand.

Note that, we require that the decisions be made for each node in the overlay graph, rather than for each edge. Thus, all the inputs to an aggregation node are either pushed to it, or all the inputs are pulled by it. This simplifies the bookkeeping significantly, without limiting the choices of partial pre-computation. If we desire to pre-compute a partial aggregate over a subset of a node's inputs, a separate partial aggregation node can be created instead. We discuss more details about this in Section 4.3.

Finally, we note that the aggregation overlay graph can be seen as a pre-compiled query plan where no unnecessary computation or reasoning is performed when an update arrives or a read query is posed. This enables us to handle much higher data rates than would be possible otherwise. We discuss the resulting execution model and related architectural decisions in the following sections.

#### 4.1.2.2 Execution Model

We begin with describing how new updates and queries are processed using the overlay graph, and briefly discuss some of the implementation issues surrounding multi-threaded execution.

**Execution Flow:** We describe the basic execution flow in terms of the *partial aggregate objects (PAOs)* that are maintained at various nodes in the overlay graph. A PAO corresponds to a partial aggregate that has been computed after aggregating over a subset of the inputs. The PAO corresponding to a node labeled *push* is always kept up-to-date as new updates arrive in any of the streams it is defined over, or if the sliding windows shift and values drop out of the window. Specifically, the updates originate at the writer nodes, and propagate through the overlay graph as far as indicated by the dataflow decisions on the nodes. The nodes labeled *push* maintain partial state and perform incremental computation to keep their corresponding PAOs up-to-date. On the other hand, no partial state is maintained at the nodes labeled *pull*. When an overlay node $u$ makes a *read* request from another node $v$ upstream of it, if $v$ is labeled *push*, the partial aggregate is returned immediately without delay. On the other hand, if $v$ is labeled *pull*, it issues *read* requests on all its upstream overlay nodes, merges all the PAOs it receives, and returns the result PAO to the requesting node.

**Single-threaded execution:** A naive implementation of the above execution model is using a single thread, that processes the *writes* and *reads* in the order in which they are received, finishing each one fully (i.e., pushing the wries as far as required, and computing the results for *reads*) before handling the next one. The main advantage of this model is that the partial state maintained at the overlay nodes, and the query results generated, are both well-defined and consistent (ignoring the temporary inconsistencies while an update is being pushed through the overlay). However, this approach cannot exploit the parallelism in the system, is likely to suffer from potential cache misses due to the random access pattern, and is unlikely to scale to the high update and query rates seen in practice.

**Multi-threaded execution:** On the other hand, a multi-threaded version can result in better throughputs and latencies, but requires careful implementation to guarantee correctness. First, the computations on the overlay graph must be made thread-safe to avoid potential state corruption due to race conditions. We can do this either by using thread-safe data structures to store the PAOs or through explicit synchronization. We use the latter approach in our implementation of the aggregates; however, user-defined aggregates may choose either of the two options. A more subtle issue is that of consistency. For example, consider a read on node $a_r$ in Figure 4.1(d). It is possible that the result generated for the query contains a more recent update on node $f_w$, but does not see a relatively older update on node $c_w$ (as $f_w$ is read later than $c_w$). We ignore the potential for such inconsistencies in this work.

We use two thread pools, one for servicing the read requests and one for servicing the write requests. The relative sizes of the two thread pools can be set based on the expected number of reads vs writes; assigning more threads to processing reads may reduce latency, but increases the possibility of stale results.

Further, there are two ways to process a read or a write using multiple threads. The first option is what we call the *uni-thread* model – here a thread that picks up a request (read or write) executes it fully before processing a new request. Alternatively, in the *queueing* model, the tasks are subdivided into micro-tasks at the granularity of the overlay nodes. Each micro-task is responsible for a single partial aggregate update operation at an overlay node (for writes) or a single partial aggregate computation at an overlay node (for reads). Any subsequent updates or required reads are pushed back onto the respective queues. The queueing model is likely to be more scalable and result in better throughputs, but the latencies for reads are substantially higher. We follow a hybrid approach – for writes, we use the queueing model, whereas for reads, we use the uni-thread model.

#### 4.1.2.3 User-defined Aggregate API

One of the key features of our system is the ability for the users to define their own aggregate functions. We build upon the standard API for user-defined aggregates for this purpose [81, 150, 107], and briefly describe it here for completeness. The user must implement the following functions.

- $initialize(PAO)$: Initialize the requisite data structures to maintain the partial aggregate state (i.e., PAOs).

- $update(PAO, PAO\_old, PAO\_new)$: This is the key function that updates the partial aggregate at an overlay node (PAO) given that one of its inputs was updated from $PAO\_old$ to $PAO\_new$.

- $finalize(PAO)$: Compute the final answer from the PAO.

Note that, we require the ability to *merge* two PAOs in order to fully exploit the potential for sharing through overlay graphs – this functionality is typically optional in user-defined aggregate APIs.

## 4.2 Constructing The Overlay

Our overall optimization goal is to construct an overlay graph annotated with precomputation (dataflow) decisions that maximize the overall throughput, given a data graph and an ego-centric aggregate query. To make the dataflow decisions optimally, we also need information about the expected read (query) and write (update) frequencies for the nodes in the graph. However, these two sets of inputs have inherently different dynamics – the data graph is expected to change relatively slowly, whereas the read/write frequencies are expected to show high variability over time. Hence, we decouple the overall problem into two phases: (1) we construct a compact overlay that maximizes the sharing opportunities given a data graph and a query, and (2) then make the dataflow decisions for the overlay nodes (as we discuss in the next section, we allow the second phase to make restricted local modifications to the overlay). The overlay construction is a computationally expensive process, and we expect that an overlay, once constructed, will be used for a long period of time (with incremental local changes to handle new nodes or edges). On the other hand, we envision re-evaluating the dataflow decisions on a more frequent basis by continuously monitoring the read/write frequencies to identify significant variations.

In this section, we focus on the overlay construction problem. We begin with defining the optimization goal, and present several scalable algorithms to construct an overlay. We then briefly discuss our approach to handling structural changes to the data graph.

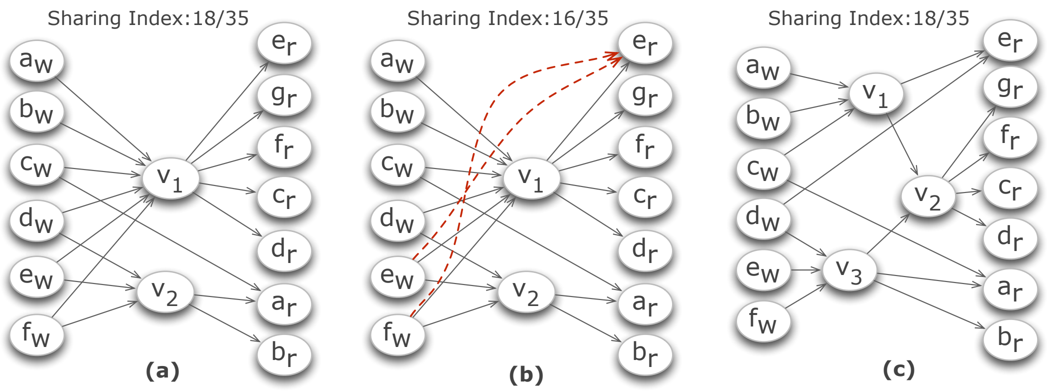


Figure 4.2: (a) A duplicate-insensitive overlay; (b) An overlay with two negative edges; (c) A multi-level overlay.

## 4.2.1 Preliminaries

As a first step, we convert the given data graph $\mathcal{G}(V, E, V_p, E_p)$ into an equivalent bipartite graph $B_{\mathcal{G}}(V', E')$, by identifying the query nodes, and the input nodes for each of the query nodes, given the user-provided query (as discussed in Section 4.1.1). We use the total number of edges in the overlay as our optimization metric, the intuition being that, each edge in the overlay corresponds to a distinct data movement and computation. We justify the use of this somewhat abstract metric by noting that the runtime cost of an overlay is highly dependent on the distribution of the read/write frequencies; for the same query and data graph, the optimal overlays could be wildly different for different distributions of read/write frequencies (which are not available at the overlay construction phase). We believe that the use of an abstract metric that rewards sharing is likely to prove more robust in highly dynamic environments.

More formally, we define the *sharing index* of an overlay to be:

$$1 - \frac{\textit{\# of edges in the overlay}}{\textit{\# of edges in } B_{\mathcal{G}}}$$

Figure 4.2 shows three overlays for our running example, and their sharing indexes. Figure 4.2(a) shows an overlay where there are multiple paths between some reader-writer pairs. As we discussed earlier, such an overlay cannot be used for a duplicate-sensitive aggregate function (like SUM, COUNT, etc.), but for duplicate-insensitive aggregate functions like MAX, it typically leads to better sharing index as well as better overall throughput. The second overlay uses *negative edges* to bring down sharing index. This should only be done for aggregate functions where the subtraction operation is incrementally computable (e.g., SUM, or COUNT). Finally, third overlay is an example of a *multi-level* overlay, and has the lowest sharing index for our running example (without use of negative edges or duplicate paths). In most cases, such multi-level overlays exhibit the best sharing index. Note that multi-level overlays can also be duplicate insensitive or contain negative edges.

The problem of maximizing the sharing index is closely related to the *minimum order bi-clique partition* problem [66], where the goal is to cover all the edges in a bipartite graph using fewest edge disjoint bicliques. In essence, a biclique in the bipartite graph $B_{\mathcal{G}}$ corresponds to a set of readers that all share a common set of writers. Such a biclique can thus be replaced by a partial aggregation node that aggregates the values from the common set of writers, and feeds them to the readers. In Figure 4.1(d), node $PA_1$ corresponds to such a biclique (between writers $a_w, b_w, c_w$ and readers $c_r, d_r, e_r, f_r, g_r$). Finding bicliques is known to be NP-Hard. Sharing index (SI) is also closely related to the *compression ratio* (CR) metric used in many of the works in *representational graph compression* [48] ; specifically, $CR = 1/(1 - SI)$. However, given the context of aggregation and the possibility of having *negative* and *duplicate-insensitive* edges in the overlay, we differentiate it from compression ratio. The problem of finding a good overlay is also closely related to the problem of *frequent pattern mining* [86, 78] as we discuss in the next section.

### 4.2.2 Overlay Construction Algorithms

In this section, we present our algorithms for constructing different types of overlays as outlined in the previous section. Given the NP-Hardness of the basic problem, and further the requirement to scale the algorithms to graphs containing tens of millions of nodes, we develop a set of efficient heuristics to achieve our goal. Our first set of proposed algorithms (called VNM$_A$, VNM$_N$, and VNM$_D$) builds upon a prior algorithm (called VNM) for bipartite graph compression by Buehrer et al. [48], which itself is a adaptation of the well-known FP-Tree algorithm for frequent pattern mining [86, 78]. In our exploratory evaluation, we found that algorithm to offer the best blend of scalability and adaptability for our purposes. Our second algorithm (called IOB) is an incremental algorithm that builds the overlay one reader at a time.

**Background: FP-Tree and VNM Algorithms:** We begin with a brief recap of the *FP-Tree* algorithm for frequent pattern mining, considered to be one of the most efficient and scalable algorithms for finding frequent patterns. We briefly outline the algorithm using the terminology of readers and writers rather than transactions and items. First, the writers are sorted in the increasing order by their overall frequency of occurrence in the reader input sets, i.e., their out-degree in $B_{\mathcal{G}}$. In our running example, the sort order (breaking ties arbitrarily) would be $\{d_w, c_w, e_w, f_w, a_w, b_w\}$. Then all the reader input lists are rewritten according to that sort order; e.g., we would write the input list of $a_r$ as $\{d_w, c_w, e_w, f_w\}$. Next, the FP-Tree is built incrementally by adding one reader at a time, starting with an empty tree. For the reader under consideration, the goal is to find its longest prefix that matches with a path from the root in the FP-Tree constructed so far. As an example, Figure 4.3 shows the FP-Tree built after addition of readers $a_r$, $b_r$, and $e_r$. A node in the FP-Tree is represented by: $x_w\{S(x_w)\}$ where $x_w$ is a writer and $S(x_w)$ is a list of readers that contain $x_w$ in their input lists (called *support set*). Now, for reader $c_r$, the longest prefix of it that matches a path from root is $d_w, c_w, e_w, f_w$. That reader would then be added to the tree nodes in that path (i.e., to the support sets along that path). If the reader input list contains any additional writers, then a new branch is created in the tree (for $e_r$ a new branch will be created with nodes $a_w\{e_r\}$ and $b_w\{e_r\}$).

Once the tree is built, in the *mining phase*, the tree is searched to find bicliques. A path $P$ in the tree from the root to the node $x_w\{S(x_w)\}$ corresponds to a biclique between

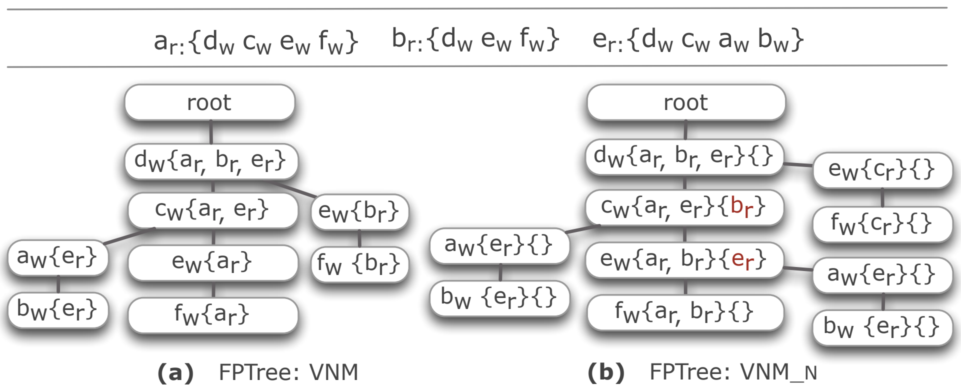


Figure 4.3: An example of FP-Tree construction for VNM and VNM$_N$: (a) Basic version, (b) FP-Tree with negative edges.

the writers corresponding to the nodes in $P$ and the readers in $S(x_w)$. Since our goal is to maximize the number of edges removed from the overlay graph, we search for the biclique that maximizes:

$$benefit(P) = L(P) * |S(P)| - L(P) - |S(P)|,$$

where $L(P)$ denotes the length of the path $P$ and $S(P)$ denotes the support for the last node in the path. Such a biclique can be found in time linear to the size of the FP-Tree. After finding each such biclique, ideally we should remove the corresponding edges (called the *mined edges*) and reconstruct the FP-Tree to find the next biclique with best benefit. Mining the same FP-Tree would still find bicliques but with lower benefit (since the next biclique we find cannot use any of the edges in the previously-output biclique).

We now briefly describe the VNM algorithm [48], which is a highly scalable adaptation of the basic FP-Tree mining approach described above; VNM was developed for compressing very large (web-scale) graphs, and in essence, replaces each biclique with a virtual node to reduce the total number of edges. The main optimization of VNM relies on limiting the search space by creating small groups of readers, and looking for bicliques that only involve the readers in one of the groups. This approach is much more scalable than building an FP-Tree on the entire data graph. VNM uses a heuristic based on *shingles* [58, 60] to group the readers. Shingle of a reader is effectively a signature of its input *writers*. If two *readers* have very similar *adjacency lists*, then with high probability, their shingle values will also be the same. In a sense, grouping readers by shingles increases the chance of finding big bicliques (with higher benefit) within the groups. The algorithm starts by computing multiple shingles for each reader, and then doing a lexicographical sort of the readers based on the shingles. The sorted list is then chunked into equal-sized groups of readers, each of which is passed to the FP-Tree algorithm separately. Mining all the reader groups once completes one iteration of the algorithm. The process is then repeated with the modified bipartite graph (where each biclique is replaced with a virtual node) to further compress the graph. Since the virtual nodes are treated as normal nodes in such subsequent iterations, a biclique containing virtual nodes may be replaced with another virtual node, resulting in connections between virtual nodes; in our context, this gives rise to *multi-level overlays* where partial aggregation nodes feed into other partial aggregators.

**VNM Adaptive ($VNM_A$):** Our first adaptation of the basic VNM algorithm is aimed at addressing a major deficiency of that algorithm, namely lack of a systematic way to choose the chunk size. Our initial experiments with VNM suggested that the effect of the chunk size on the final compression achieved is highly non-uniform across various graphs like web graphs and social graphs. We noticed that a bigger chunk size typically finds bigger bicliques, but it can't find all big bicliques, especially when there is big overlap in the reader sets of two potential bicliques. This is because the reader sets of two subsequent mining phases in VNM are mutually exclusive. Second, a bigger chunk size makes it harder to find small bicliques, which is especially a problem with later iterations; since many of the original edges have been deleted in the first few iterations, only small bicliques remain in the graph. On the other hand, using a small chunk size from the beginning ignores large bicliques that can deliver huge savings.

To address this problem, we develop an adaptive variation of VNM that uses different chunk sizes for different iterations. For the first iteration, we use a large chunk size (100 in our experiments) and dynamically reduce it for future iterations. For the $i^{th}$ iteration, let $c_i$ denote the chunk size, and let $B_i^s$ denote the sum total of the *benefits* (defined in Section 4.2.2) for all the bicliques found in that iteration with reader set size $= s$ (note that, $s \leq c_i$). We choose $c_{i+1} \leq c_i$ to be the smallest $c$ such that: $\sum_{s \leq c} B_i^s > 0.9 \sum_{s \leq c_i} B_i^s$. Although our algorithm also requires setting two parameter values, our extensive experimental evaluation on many real-world graphs showed that the algorithm is not sensitive to the initial chunk size to within an order of magnitude, and to the second parameter between 0.8 and 1.

**VNM with Negative Edges ($VNM_N$):** Next, we present our adaptation to VNM that considers adding negative edges to reduce the overlay size. In essence, we look for *quasi-bicliques* that may be a few edges short of being complete bicliques (this problem is also known to be NP-Hard [104]). For scalability, our algorithm employs the same basic structure as the VNM$_A$ algorithm discussed above; however, we modify the FP-Tree construction and mining algorithms to support negative edges.

Recall that a node in an FP-Tree is represented by $x_w\{S(x_w)\}$ where $x_w$ is a writer and $S(x_w)$ contains the readers that contain $x_w$ in their input lists. To accommodate negative edges, we now represent a node by $x_w\{S(x_w)\}\{S'(x_w)\}$, where $S'(x_w)$ contains readers that do not contain $x_w$ in their input list, but may contain the writers corresponding to the nodes below this node in the FP-Tree. Benefit of a path $P$ in the FP-Tree is now given by:

$benefit(P) = L(P) * |S(P)| - L(P) - |S(P)| - \sum_P |S'(x_w)|$,

where the last term captures the number of negative edges along $P$.

In our proposed algorithm, when an FP-Tree is augmented to include a new reader $r$, we add $r$ along up to $k_1$ paths in the FP-Tree that maximize the benefit given the FP-Tree constructed so far. More specifically, we exhaustively explore the FP-Tree in a breadth-first manner, and for each node visited, we compute the benefit of adding $r$ along the path. We then choose up to $k_1$ paths with the highest benefit and add the reader along those paths. As with the original FP-Tree algorithm, additional branches may have to be created for the remaining writer nodes in $r$.

Figure 4.3(b) shows an example where upto two paths can be added for a reader (i.e., $k_1 = 2$). Both readers $b_r$ and $e_r$ create two paths in the overlay, one of which uses a negative edge. Note that $e_r$ creates a new branch in the tree (apart from the one similar to in the basic version); after the introduction of negative edge for $e_r$ at $a_w$, there are still nodes remaining in $e_r$'s input list. During the mining phase the new FP-Tree finds a biclique of size *3x3*. On the other hand, the basic version can only find a biclique of size *2x2*.

Although our algorithm finds the best paths to add the reader along, it runs in time linear to the size of the FP-Tree constructed so far. However, since the FP-Tree, in essence, now encodes information about $k_1$ times as many readers as it did before, the size of the FP-Tree itself is expected to be larger by about the same factor. To improve efficiency, we stop the breadth-first exploration down a path if more than $k_2$ negative edges are needed to add $r$ along that path (we set $k_2 = 5$ in our experiments). This optimization has little impact on performance since it is unlikely that quasi-bicliques requiring a large number of negative edges will be beneficial.

**Duplicate-insensitive VNM ($VNM_D$):** Next, we discuss our proposed algorithm for finding overlays that exploit the duplicate-insensitive nature of some aggregates and allow for multiple paths between a writer and a reader. There are two natural ways to extend the VNM algorithm for reusing edges in this fashion. First, we can keep the basic structure of the VNM algorithm and modify the FP-Tree algorithm itself to find multiple bicliques in each mining phase, while ignoring the overlap between bicliques. However, by construction, the bicliques mined from a single FP-Tree tend to have very high overlap, and the benefits for additional bicliques found can be very low. It is also not clear how many aggregate nodes to add in a single mining phase; adding all bicliques for which the benefit is non-zero is likely to lead to many partial aggregate nodes, each providing low benefit.

Instead, in our proposed algorithm VNM$_D$, we modify the reader grouping phase itself. In VNM, in each iteration, the readers are grouped into disjoint groups before passing to the FP-Tree construction and mining phase. Instead, we allow the groups of readers to overlap. Specifically, given an overlap percentage $p$ (an algorithm parameter), we allow two consecutive groups of readers to have $p\%$ readers in common. The FP-Tree construction and mining phases themselves are unchanged with the following exceptions. First, instead of representing an FP-Tree node as $x_w\{S(x_w)\}$, we represent it as $x_w\{S^{notmined}(x_w)\}\{S^{mined}(x_w)\}$, where $S^{mined}(x_w)$ contains the readers $r$ such that the edge from $x_w$ to $r$ was present in a previously used biclique. Second, we modify the formula for computing the benefit of a path as follows:

$$benefit(P) = L(P) * |S(P)| - L(P) - |S(P)| - \sum_P |S^{mined}(x_w)|;$$

the last term captures the number of *reused* edges in the biclique.

**Incremental Overlay Building (IOB):** The overlay constructions algorithms that we have developed so far are all based on identifying sharing opportunities by looking for bicliques in $B_{\mathcal{G}}$. However, to make those algorithms scalable, two heuristics have to be used: one to partition the readers into small groups, and one to mine the bicliques themselves. In essence, both of these focus the search for sharing opportunities to small groups of readers and writers, and never consider the entire $B_{\mathcal{G}}$ at once. Next, we present an incremental algorithm for building the overlay that starts with an empty overlay, and adds one reader at a time to the overlay. For each reader, we examine the entire overlay constructed till that point which, as our experimental evaluation demonstrates, leads to more compact overlays.

We begin with ordering the readers using the shingle order as before, and add the readers one at a time in that order. In the beginning, the overlay graph simply contains the (singleton) writer nodes. Let $\langle r, \mathcal{N}(r)\rangle$ denote the next reader to be added. Let

$\langle ovl_n, \mathcal{I}(ovl_n)\rangle$ denote a node in the overlay constructed so far, where $\mathcal{I}(ovl_n)$ is the set of writers whose partial aggregate $ovl_n$ is computing. For reader $r$, our goal is to reuse as much of the partial aggregation as possible in the overlay constructed so far. In other words, we would like to find the smallest set of overlay nodes whose aggregates can be used to compute the aggregate for $r$. This problem is essentially the *minimum exact set cover problem*, which is known to be NP-Complete.

We use a standard greedy heuristic commonly used for solving the set cover problem. We start by finding the overlay node that has maximum overlap with $\mathcal{N}(r)$, and restructure the overlay to make use of that overlap. We keep on repeating the same process until all nodes in $\mathcal{N}(r)$ are covered (since the singleton writer nodes are also considered part of the overlay, we can always cover all the nodes in $\mathcal{N}(r)$). Let $\langle v_1, B\rangle$ denote the overlay node that was found to have the highest overlap with the uncovered part, denoted $A$, of $\mathcal{N}(r)$. If $B \subseteq A$, then we add an edge from $v_1$ to $r$, and repeat the process with $A - B$. Otherwise, we restructure the overlay to add a new node $\langle v_1', A \cap B\rangle$, reroute the appropriate incoming edges (i.e., the incoming edges corresponding to the writers in $A \cap B$) from $v_1$ to $v_1'$, and add a directed edge from $v_1'$ to $v_1$. We then also add an edge from $v_1'$ to $r$. If $A - A \cap B$ is non-empty, then we repeat the process to cover the remaining inputs to $r$.

As with the VNM-based algorithms, we use multiple iterations to improve the overlay. In each iteration (except the 1st iteration), we revisit the decisions made for each of the partial aggregator nodes, and do local restructuring of the overlay if better decisions are found for any of the partial aggregator nodes (using the same set cover-based algorithm as above).

For efficient execution of the algorithm we maintain both a *reverse index* and a *forward index*. For a writer node $w$, the reverse index tells us which are the overlay nodes that are aggregating $w$. For example, $a_w$'s reverse index entry will have both $v_1$ and $v_2$. Note that even though there is no direct edge from $a_w$ to $v_2$, $a_w$'s reverse index entry has $v_2$ because $v_2$ is effectively aggregating $a_w$. This index helps us to find the overlay node that provides maximum cover to a set of input nodes using one single scan of the input list. On the other hand, for any node $n$ in the overlay, the forward index tells us the input list of $n$. For example, $v_2$'s forward index entry will have $v_1$ and $v_3$ in it.

Although the above algorithm could be extended to allow for negative edges and/or duplicate paths, we do not discuss those extensions here. This is because, although IOB finds significantly smaller overlays, the overlays tend to be deep (with many levels) and in our experimental evaluation, the end-to-end throughput for the overlays was lower than for the overlays found with the VNM$_A$ algorithm. Thus, although the IOB algorithm is a better algorithm for finding compact overlays and for compressing bipartite graphs, VNM-based algorithms are better suited for our purpose.

### 4.2.3 Handling Dynamic Changes

We briefly sketch our techniques to incrementally update the overlay in response to structural changes to the underlying data graph.

**Addition of edges:** When a new edge is added to the data graph, we explore the neighborhoods of both the endpoints to construct a list: $\{\langle r_1, \Delta(\mathcal{I}(r_1))\rangle, \langle r_2, \Delta(\mathcal{I}(r_2))\rangle, ...\}$, where $r_i$ denotes a reader whose input list has changed, and $\Delta(\mathcal{I}(r_i))$ denotes the new writer nodes that are added to $\mathcal{I}(r_i)$ (for queries defined over 2-hop neighborhoods or larger, the changes to the input lists can be substantial). We process the change for each of the readers separately. If $|\Delta(\mathcal{I}(r_i))|$ is larger than a prespecified *threshold*, we use the IOB algorithm to add a new aggregate node that computes a partial aggregate over the writers in $\Delta(\mathcal{I}(r_i))$ (in the best case scenario, an existing overlay node may already compute exactly this aggregate); we then add an edge from that node to $r_i$. Otherwise, we add direct edges from the writer nodes in $\Delta(\mathcal{I}(r_i))$ to $r_i$. At the same time, for each reader in the overlay $r$, we also keep a count of direct edges from writers to that reader; this count is updated in the latter case, and if the count is larger than the *threshold*, we use the IOB algorithm to restructure the overlay as above.

**Deletion of edges:** Edge deletions are trickier to handle because significant restructuring of the overlay may be needed, especially for multi-level overlays and complex ego-centric queries. As above, we explore the neighborhoods of the endpoints of the deleted edge to construct a list: $\{\langle r_1, \Delta(\mathcal{I}(r_1))\rangle, \langle r_2, \Delta(\mathcal{I}(r_2))\rangle, ...\}$, and we process each reader independently. For each of the readers, we make a pre-processing pass over the overlay to identify how many of the overlay nodes would need to be modified to accommodate the change. In other words, for reader $r_i$, we count the number of overlay nodes $ovl$ that are upstream of $r_i$ and $\mathcal{I}(ovl) \cap \Delta(\mathcal{I}(r_i)) \neq \phi$. If this number is small ($\leq 5$), we modify the overlay by splitting the overlay nodes and removing edges appropriately (we omit the details here for brevity). Otherwise, we simply remove all incoming edges to $r_i$ (and any partial aggregate nodes that only send values to $r_i$), and use the IOB algorithm to add $r_i$ back in with the modified input list.

**Addition or deletion of nodes:** Addition of a new node, $u$, to the data graph is easy to handle. First, we add a new writer node $u_w$ to the overlay, and then add direct edges from $u_w$ to some of the existing reader nodes as dictated by the new edges added between $u$ and the existing nodes (in most cases, a new node is added with one edge to an existing node). Second, we construct an input list for $u$, and we use the IOB algorithm to add a new reader node $u_r$ with that input list. On the other hand, when a node $u$ is deleted from the data graph, we simply remove $u_w$ and $u_r$ and all their incident edges from the overlay graph. We also remove $u$ from the data structures used for incremental maintenance (i.e., the forward and the reverse indexes).

## 4.3 Making Dataflow Decisions

Next, we discuss how to make the dataflow (i.e., precomputation) decisions to maximize the total throughput given an overlay network, and the expected read/write frequencies for the nodes. Surprisingly, the problem can be solved optimally in polynomial time. We begin with the preliminaries related to the cost of a dataflow decisions and then provide the formal problem definition and present the analysis along with the algorithms that we propose.

**Preliminaries:** For each node $v \in V$ in the data graph, let $r(v)$ denote its read frequency (i.e., the number of times a query is issued at node $v$), let $w(v)$ denote its write frequency (i.e., the number of times $v$ is updated)[1]. Given these, with each node $u \in V''$ in the overlay graph $O_{\mathcal{G}}(V'', E'')$, we associate two numbers, $f_l(u)$ and $f_h(u)$, called *pull*

[1]See Table 3.1 for a summary of notation.

*frequency* and *push frequency*, respectively. $f_h(u)$ captures the number of times data values would be *pushed* to $u$ if all nodes in the overlay are assigned *push* decisions. Similarly, $f_l(u)$ indicates the number of times data values would be *pulled* from $u$ if all nodes in the overlay are assigned *pull* decisions.

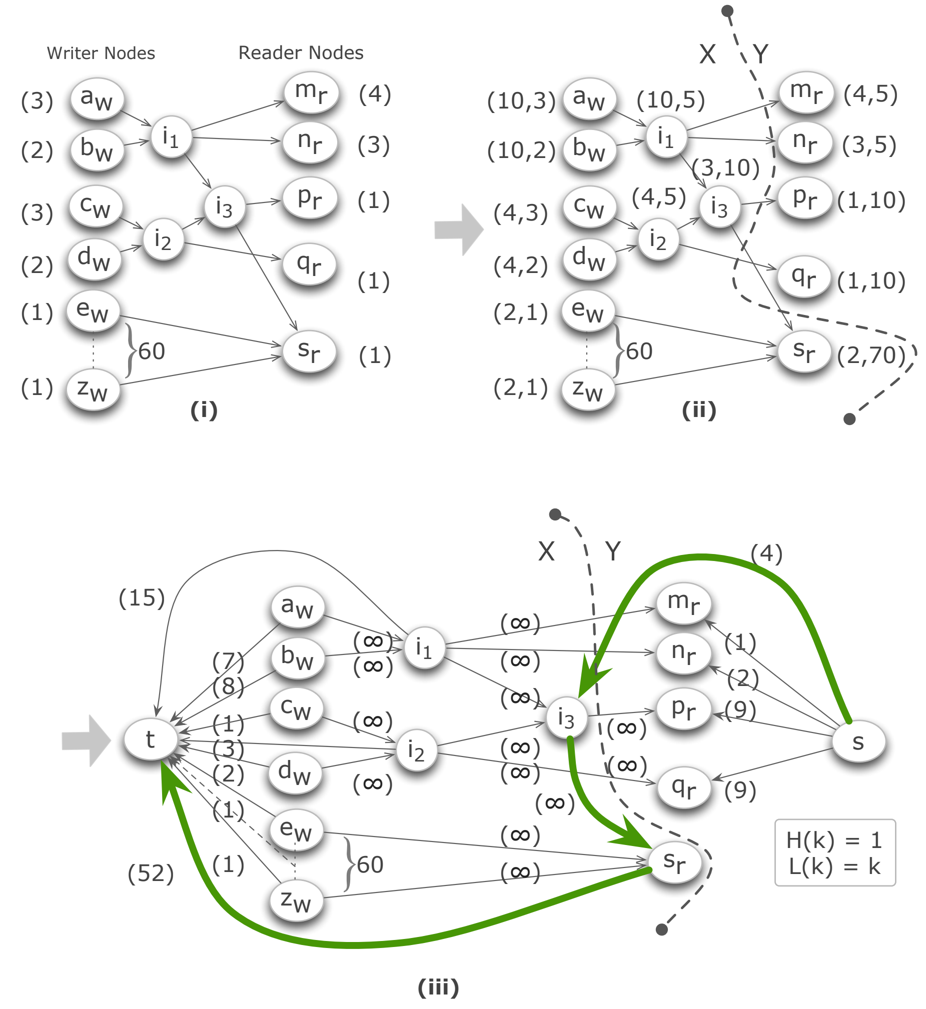


Figure 4.4: (i) An example overlay annotated with read/write frequencies; (ii) Computing (*pull*, *push*) frequencies; (iii) Construction of the *s-t max-flow* graph (with the annotations denoting the edge capacities)

The push and pull frequencies are computed as follows. For computing push frequencies, we start by assigning $f_h(a_w) = w(a_w)$ for all writer nodes $a_w$, then propagate the push frequencies from left to right (downstream). For an aggregation node or a reader node $u$, $f_h(u)$ is computed by summing up the push frequencies for all nodes that are immediately upstream of $u$. Similarly, the pull frequencies are computed by starting with the reader nodes in the overlay, then recursively computing the pull frequencies for the rest of the nodes. Figure 4.4(i)-(iii) illustrates this with an example that we also use to show how our algorithm makes the dataflow decisions.

**Push and Pull Costs:** As discussed before, a *push* decision on a node implies that the aggregate corresponding to that node will be (incrementally) precomputed and will be available for immediate consumption. On the other hand, a *pull* decision on a node implies that the aggregate will be computed on demand when the node is read. In order to reason about the tradeoff between push and pull, we need to be able to compute the cost of a push or a pull. This cost typically depends on the nature of the aggregate, the type and the size of the sliding window [35]. We capture these costs as two functions: $H(k)$ denotes the average cost of one push for an aggregation node with $k$ inputs, $L(k)$ denotes the average cost of one pull for that node. For example, for a SUM aggregate node, we expect $H(k) \propto 1$ and $L(k) \propto k$, whereas for a MAX aggregate node, if we use a priority queue for handling incremental updates, then $H(k) \propto \log_2(k)$ and $L(k) \propto k$. To handle sliding windows, we implicitly assign $w$ inputs to each writer where $w$ is the average number of values in the sliding window at a writer – thus if the sliding window is of size 10, then $PUSH$ and $PULL$ costs of the writer node will be $H(10)$ and $L(10)$ respectively. We assume $H()$ and $L()$ are either provided, or are computed through a calibration process where we invoke the aggregation function for a range of different inputs and learn the $H()$ and $L()$ functions.

**Problem Definition:** The dataflow decisions made by a solution induce a *node partition*, denoted $(X, Y), X \cap Y = \phi$, on the overlay graph, where $X$ contains nodes that are designated *push*, $Y$ contains nodes designated *pull* (Figure 4.4(ii)). Since all nodes upstream of a *push* node must also be designated *push* (and similarly all nodes downstream of a *pull* node must also be *pull*), the partition induced by any consistent set of dataflow decisions must satisfy the constraint that there is no edge from a node in $Y$ to a node in $X$.

For an overlay node $v$, let $PUSH(v) = f_h(v) * H(deg(v))$ denote the cost incurred if it is designated a *push* node, let $PULL(v) = f_l(v) * L(deg(v))$ denote the cost if it is a *pull* node. Although the push/pull decisions cannot be made for the nodes independently (because of the aforementioned constraint), $PUSH()$ and $PULL()$ costs can be computed independently; this is because the computations that happen at a node when it is invoked, do not depend on the dataflow decisions at its input or output nodes. Thus, to minimize the total computational cost, our goal reduces to finding an ($X$, $Y$) partition of the overlay (with no edges going from $Y$ to $X$) that minimizes: $\sum_{v \in X} PUSH(v) + \sum_{v \in Y} PULL(v)$.

**Query Latencies:** Another consideration in making dataflow decision is the impact on query *latencies*. Throughput maximization may lead to higher use of *pull* decisions, especially if reads are less frequent than writes, that may result in high query latencies. As we show in Section 4.4, because our system is entirely in-memory and does not need to do distributed network traversals, the query latencies are quite low even in the worst-case.

**Algorithm:** We design an algorithm for a slightly more general problem, that we describe first. We are given a directed acyclic graph $H(H_V, H_E)$, where each vertex $v \in H_V$ is associated with a weight $w(v)$; $w(v)$ may be negative. For ease of exposition, we assume that $\forall v, w(v) \neq 0$. We are asked to find a graph partition $(X, Y)$, such that there are no edges from $Y$ to $X$, that maximizes: $\sum_{v\in X} w(v) - \sum_{v \in Y} w(v)$. Note that, the solution is trivially $(X = H_V, Y = \phi)$ if all node weights are positive. We also note that, the metric has the maximum possible value if all nodes with $w(v) < 0$ are assigned to $Y$, all nodes with $w(v) > 0$ to $X$. However, that particular assignment may not guarantee that there are no edges from $Y$ to $X$.

To reduce our problem to this problem, we set:

$$w(v) = f_l(v)L(deg(v)) - f_h(v)H(deg(v)) = PULL(v) - PUSH(v)$$

That is, the weight of node $v$ is the "*benefit*" of assigning it a *push* decision (which is negative if $PULL(v) < PUSH(v)$). Then:

$$\begin{aligned}
&\sum_{v\in X} w(v) - \sum_{v\in Y} w(v) \\
&= \sum_{v\in X}(PULL(V) - PUSH(v)) - \sum_{v\in Y}(PULL(V) - PUSH(v)) \\
&= \underline{\sum_{v\in H_V}(PULL(v) + PUSH(v))} - 2(\sum_{v\in X} PUSH(v) + \sum_{v\in Y} PULL(v))
\end{aligned}$$

Since the underlined term is a constant, maximizing this is equivalent to minimizing $\sum_{v\in X} PUSH(v) + \sum_{v\in Y} PULL(v)$.

To solve this more general problem, we construct an edge-weighted graph $H'(H'_V, H'_E)$

from $H(H_V, H_E)$ (in practice, we do not make a copy but rather augment $H$ in place). $H'_V$ contains all the vertices in $H_V$ and in addition, it contains a *source* node $s$ and a *sink* node $t$ (nodes in $H'$ are unweighted). Similarly, $H'_E$ contains all the edges in $H_E$, with edge weights set to $\infty$. Further, for each $v \in H_V$ such that $w(v) < 0$, we add a directed edge in $H'$ from $s$ to $v$ with weight $w'(s, v) = -w(v)$. Similarly for $v \in H_V, s.t.\ w(v) > 0$, we add a directed edge in $H'$ from $v$ to $t$ with weight $w'(v, t) = w(v)$ (see Figure 4.4(iii) for an example).

We note that, this construction may seem highly counter-intuitive, since a lot of nodes in $H'$ have either no outgoing or no incoming edges and there are few, if any, directed paths from $s$ to $t$. In fact, the best case scenario for the algorithm is that: *there is no directed path from $s$ to $t$.* This is because, a path from $s$ to $t$ indicates a *conflict* between two or more nodes. The highlighted path form $s$ to $t$ in Figure 4.4(iii) provides an example. The best decision for node $i_3$ in isolation would be *pull* ($PULL(i_3) = 6, PUSH(i_3) = 10$), but that for $s_r$ is *push* because of its high in-degree and because $L(k) = k$ ($PULL(s_r) = 2 * 60 = 120, PUSH(s_r) = 70$). However, a *pull* on $i_3$ would force a *pull* on $s_r$, hence both of them cannot be assigned the optimal decision in isolation.

After constructing $H'$, we find an $s$-$t$ directed min-cut in this directed graph, i.e., a set of edges $C \in H'_E$ with minimum total edge-weight, such that removing those edges leaves no directed path from $s$ to $t$. Let $Y$ denote the set of nodes in $H'$ reachable from $s$ after removing the edges in $C$ (excluding $s$), let $X$ denote the set of remaining nodes in $H'$ (excluding $t$).

**Theorem 4.3.1.** *$(X, Y)$ is a node partition of $H$ s.t. there are no edges from $Y$ to $X$, $\sum_{v \in X} w(v) - \sum_{v \in Y} w(v)$ is maximized.*

**Proof:** We prove that the original problem of finding an $(X, Y)$ partition of $H$ with the desired properties is equivalent to finding a $s$-$t$ min-cut in $H'$.

Let $(X, Y)$ denote an optimal solution to our original problem, i.e., a partition of $H(H_V, H_E)$ such that the edges are directed from $X$ to $Y$ and $\sum_{v \in X} w(v) - \sum_{v \in Y} w(v)$ is maximized. Let $B = \{v \in H_V | w(v) < 0\}$ denote the vertices that have an edge from $s$ in $H'$, and let $A = \{v \in H_V | w(v) > 0\}$ denote the vertices that have an edge to $t$ in $H'$. Figure 4.5 shows the structure of the optimal solution, where we define $A_1 = A \cap X, B_1 = B \cap X, A_2 = A \cap Y$, and $B_2 = B \cap Y$.

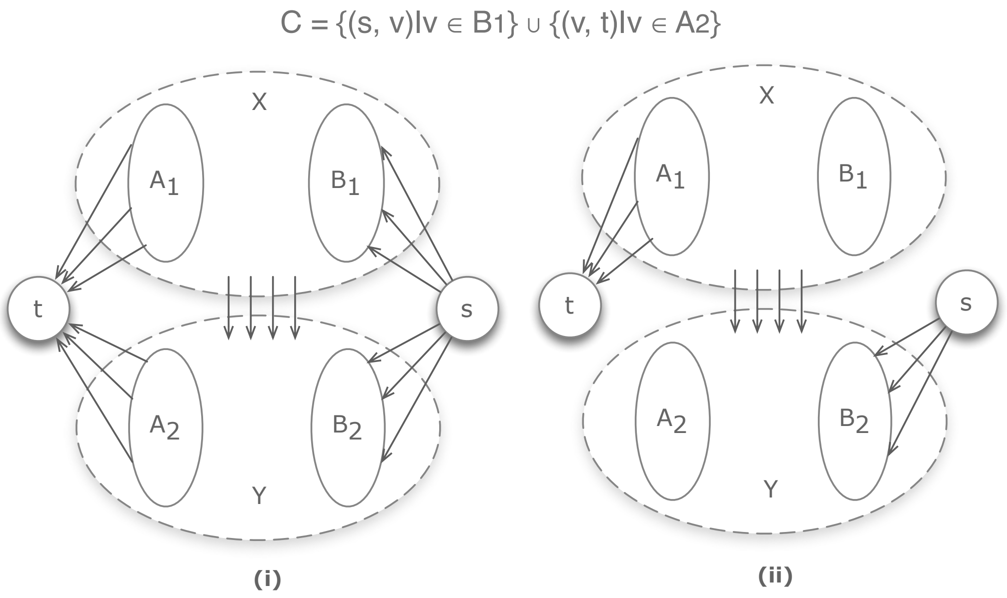


Figure 4.5: (i) Structure of an optimal solution, (ii) Removing C leaves no path between *s* and *t*.

Let $C = \{(s, v) | v \in B_1\} \cup \{(v, t) | v \in A_2\}$. Since $(X, Y)$ is a partition of $H$ such that there are no (directed) edges from $Y$ to $X$, removing $C$ from $H'$ leaves no path from $s$ to $t$ (although there are edges from $s$ to nodes in $B_2$ and from nodes in $A_1$ to $t$, there can be no path from a node in $B_2$ to a node in $A_1$ since $B_2 \subseteq Y$ and $A_1 \subseteq X$). In other words, $C$ is an $s$-$t$ directed cut.

Now:

$$\begin{aligned}
\sum_{v \in X} w(v) - \sum_{v \in Y} w(v) &= \sum_{v \in A_1} w(v) + \sum_{v \in B_1} w(v) - \sum_{v \in A_2} w(v) - \sum_{v \in B_2} w(v) \\
&= \underline{\sum_{v \in A} w(v) - \sum_{v \in B} w(v)} - 2(\sum_{v \in B_1} (-w(v)) + \sum_{v \in A_2} w(v)) \\
&= \underline{\sum_{v \in A} w(v) - \sum_{v \in B} w(v)} - 2(\sum_{e \in C} w'(e))
\end{aligned}$$

Since the underlined term is a constant, the optimal solution is such that $\sum_{e \in C} w'(e)$ is minimized, i.e., $C$ is an $s$-$t$ directed min-cut with the constraint that there is no edge from $Y$ to $X$.

Thus the last thing we need to prove is that: if $C$ is an $s$-$t$ directed min-cut of $H'$, then the corresponding partition $(X, Y)$ of $H_V$ satisfies this constraint.

First we note two things. First, $C$ cannot contain any of the edges from the original edges $H_E$ (that all have weight $\infty$). This is because, the set of all outgoing edges from $s$ is a valid $s$-$t$ cut that has a finite value, so a min-cut must have finite value. Second, given any $s$-$t$ cut $C$ that does not include any of the original edges in $H_E$, we can define a corresponding partition $(X, Y)$ uniquely, where $Y$ contains nodes that are reachable from $s$ (excluding $s$), and $X$ contains the remaining nodes in $H'_V$ (excluding $t$).

Assume to the contrary that, in the $(X, Y)$ partition defined by the min-cut $C$, there is a directed edge $(u, v)$ such that $u \in Y$ and $v \in X$. Since the weight of this edge is $\infty$, it will not be part of the cut. There are four cases:

- $u \in B_2, v \in A_1$: Since $(s,u)$ and $(v,t)$ are $\notin C$, we get a path from $s$ to $t$, contradicting the assumption that $C$ is a cut.

- $u \in B_2, v \in B_1$: This means there is a path from $s$ to $v$ (through $u$) even after deleting the edges in $C$, and thus $(s,v)$ can be removed from $C$ without compromising the property that $C$ is an $s$-$t$ cut. In other words, $C$ is not a min-cut.

- $u \in A_2, v \in B_1$: Note that there must be a path from $s$ to $u$ after removing $C$, otherwise we wouldn't need to include $(u,t)$ in the cut. Then, as above, we have a path from $s$ to $v$, and $C$ is not a min-cut.

- $u \in A_2, v \in A_1$: As above, there is a path from $s$ to $u$ after removing $C$, and combined with $(v,t)$, we get a path from $s$ to $t$ contradicting the assumption that $C$ is a cut.

We thus conclude that, finding an optimal $(X,Y)$ partition of $H$ that minimizes the objective function is equivalent to finding a $s$-$t$ directed min-cut in $H'$.

We use the Ford-Fulkerson algorithm to construct an $s$-$t$ max-flow in $H'$, use it to find the optimal $(X,Y)$ partition of $H$.

**Pre-processing:** Although the above algorithm runs in polynomial time, it is not feasible to run max-flow computations on the graphs we expect to see in practice. However, a simple pre-processing pruning step, run on $H$ before augmenting it, typically results in massive reduction in the size of the graph on which the max-flow computation must be run.

Consider node $a_w$ in the example graph in Figure 4.4(ii). The best decision for that node by itself is a *push* decision (since $PUSH(a_w) = 3 < PULL(a_w) = 10$). Since there is no node upstream of $a_w$ (which is a writer node), we can assign this node a *push* decision without affecting decisions at any other node (any node downstream of $a_w$ can still be assigned either decision), and remove it from the graph. Similarly we can assign push decision to node $b_w$ and remove it from $H$. After that, we can see that node $i_1$ can also now be assigned a *push* decision (optimal for it in isolation) without affecting any other node. Similarly, we can assign *pull* decisions to nodes $m_r, n_r, p_r$, $q_r$ and remove them by an analogous reasoning.

We now state the pruning rules, which are applied directly to $H$ (i.e., before constructing the augmented graph): (P1) recursively remove all nodes $v$ such that $w(v) > 0$ and $v$ has no incoming edges, assign them *push* decisions, (P2) recursively remove all nodes $v$ such that $w(v) < 0$ and $v$ has no outgoing edges, assign them *pull* decisions. This pruning step can be applied in linear time over the overlay graph. The resulting graph after pruning is likely to be much smaller than the original graph, is also likely to be disconnected. We then apply the above max-flow-based algorithm to each of the connected components separately.

**Theorem 4.3.2.** *Use of pruning rules P1 and P2 does not compromise optimality.*

**Proof:** Let $H$ denote the original overlay graph, and let $H^p$ denote the pruned graph after applying rules P1 and P2. Let $H'$ denote the augmented graph constructed from $H$. Let $v$ denote a node that was pruned using rule P1, and thus assigned a *push* decision.

Let $UPSTREAM(v)$ denote the set of all nodes upstream of $v$ in $H$ (including $v$). According to P1, node $v$ being pruned and assigned a *push* decision implies that all nodes in $UPSTREAM(v)$ must also been pruned and assigned a *push* decision. It follows from definition of $UPSTREAM(v)$ that there are no directed edges into $UPSTREAM(v)$ from any of the remaining nodes in $H$. Further, for any node $u \in UPSTREAM(v)$, $w(v) > 0$ (for P1 to apply), and thus there is an edge $(u, t)$ in $H'$ but no edge from $s$ to $u$. Thus, even in $H'$, there is no edge to a node in $UPSTREAM(v)$ from one of the remaining nodes $H'$.

Putting these together, we have that the nodes in $UPSTREAM(v)$ are not reachable from $s$ in $H'$. Thus in any network flow that originates at $s$, no flow can reach the nodes in $UPSTREAM(v)$ and those nodes do not participate in the network flow. As a result, the nodes in $UPSTREAM(v)$ will remain unreachable from $s$ in the residual graph at the end of max-flow algorithm, and hence, they will all be assigned a *push* decision in the end (i.e., same decision as assigned by the pruning step). By an analogous reasoning, we can show that the nodes that are pruned by the pruning rule P2 and assigned a *pull* decision will remain unreachable from $s$ in the residual graph of max-flow as well (this is because there is no path from those nodes to $t$, and hence they don't participate in the max-flow solution either).

Further, since the pruned nodes do not participate in the max-flow in any way, removing them does not change the max-flow solution found either. Hence, the decisions made on the pruned graph $H^p$ will be identical to the ones made on the original graph $H$.

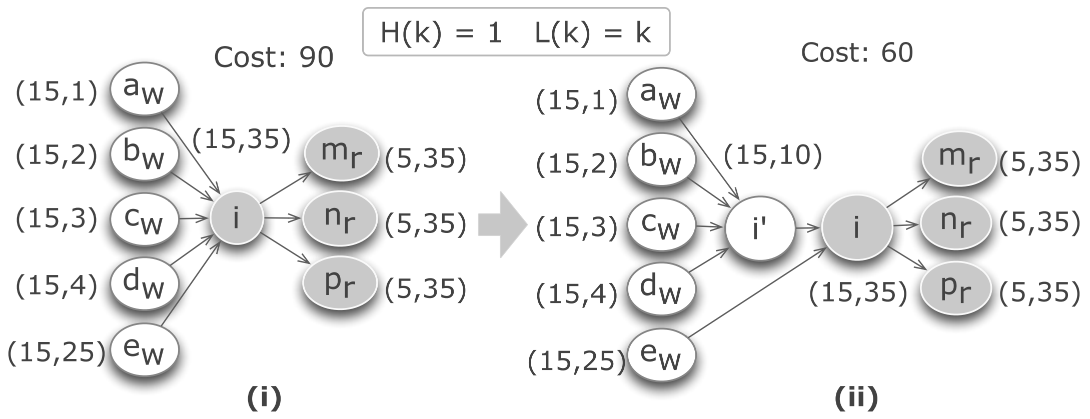


Figure 4.6: Splitting a node based on push-pull frequencies.

**Partial Precomputations by Splitting Nodes:** Making decisions on a per-node basis can lose out on a significant optimization opportunity – based on the push and pull frequencies, it may be beneficial to partially aggregate a subset of the inputs to an aggregate node. Figure 4.6 shows an example. Here, because of the low write frequencies for inputs $a_w, b_w, c_w$, and $d_w$ for aggregator node $i$, it is better to compute a partial aggregate over them, but compute the full aggregate (including $e_w$) only when needed (i.e., on a read).

One option would be to make the pre-computation decisions on a per-edge basis. However, that optimization problem is much more challenging because the cost of an incremental update for an aggregate node depends on how many of the inputs are being incrementally aggregated, how many on demand; thus the decisions made for different edges are not independent of each other. Next we propose an algorithm that achieves the same goal, but in a much more scalable manner.

For every node $v$ in the overlay graph, we consider splitting its inputs into two groups. Let $f$ denote the pull frequency for $v$, let $f_1, ..., f_k$ denote the push frequencies of its input nodes, sorted in the increasing order. For every prefix $f_1, ..., f_l$, of this sequence, we compute: $\sum_{i \leq l} f_i H(l) + f \times L(l)$. We find the value of $l$ that minimizes this cost; if $l \neq 0$ and $l \neq k$, we construct a new node $v'$ that aggregates the inputs corresponding to frequencies $f_1, ..., f_l$, remove all those inputs from $v$, add $v'$ as an input to $v$. As we show in our experimental evaluation, this optimization results in significant savings in practice.

**Adapting the Dataflow Decisions:** Most real-world data streams, including graph data streams, show significant variations in read/write frequencies over time. We propose, empirically evaluate, a simple adaptive scheme to handle such variations. For a subset of the overlay nodes (specified below), we monitor the observed push/pull frequencies over recent past (the window size being a system parameter). If the observed push/pull frequencies at a node are significantly different than the estimated frequencies, then we reconsider the dataflow decision just for that node and change it if deemed beneficial. Dataflow decisions can be unilaterally changed in such a manner only for: *pull* nodes all of whose upstream nodes are designated *push*, *push* nodes all of whose downstream nodes are designated *pull* (we call this the *push/pull frontier*). Hence, these are the only nodes for which we monitor push/pull frequencies (it is also easier to maintain the push/pull frequencies at these nodes compared to other nodes).

## 4.4 Evaluation

In this section, we present a comprehensive experimental evaluation using our prototype system using several real-world information networks. Our results show that overall our approach results in significant improvements, in many cases order of magnitude improvements, in the end-to-end throughputs overall baselines, and that our overlay construction algorithms are effective at finding compact overlays.

### 4.4.1 Experimental Setup

We ran our experiments on a 2.2GHz, 24-core Intel Xeon server with 64GB of memory, running 64-bit Linux. Our prototype system is implemented in Java. We use a set of dedicated threads to play back the write and read traces (i.e., to send updates and queries to the system), and a thread pool to serve the read and write queries.

**Datasets and Query Workload:** We evaluated our approach on several real-world information networks including [2]: (1) LiveJournal social network (*soc-LiveJournal1*: 4.8M nodes/69M edges), (2) social circles from Google+ (*ego-Gplus*:107k/13M), (3) Web graph of Berkeley and Stanford (*web-BerkStan*: 685k/7.6M), (4) *Hollywood-2009* Social Graph (1.1M/114M), (5) *EU2005* Web Graph (862k /19M), and (6) *UK2002* Web Graph (18.5M /298M). In this chapter, we report results for the first two, and the last two.

We report results for three ego-centric aggregate queries: SUM, MAX, and TOP-K, all specified over 1-hop neighborhoods. SUM and MAX queries ask us to compute the total sum and the max over the input values respectively. TOP-K asks for the $k$ most *frequent* values among the input values, and is a holistic aggregate [107].[3]

Since the user activity patterns (i.e., read/write frequencies) are not available for any real-world network that we are aware of, we generate those synthetically using a Zip-

[2]First three are available at `http://snap.stanford.edu/data/index.html`, and the latter three at `http://law.di.unimi.it`.

[3]In other words, TOP-K is a generalization of *mode*, not *max*.

fian distribution; event rates in many applications like tweets in Twitter, page views in Yahoo!'s social platform have been shown to follow a Zipfian distribution [134, 44]. Further, we assume that the read frequency of a node is linearly related to its write frequency; we vary the write-to-read ratio itself to understand its impact on the overall performance. For some of the experiments, we used real network packet traces to simulate user activity [4]: (1) EPA-HTTP, and (2) UCB Home IP Web Traces.

**Evaluation Metric:** Our main evaluation metric is the *end-to-end throughput of the system*, i.e., the total number of read and write queries served per second. This metric accounts for the side effects of all potentially unknown system parameters whose impact might not show up for a specifically designed metric, and thereby reveals the overall efficacy of the system. When comparing the overlay construction algorithms, we also use the following metrics: sharing index (SI), memory consumption, and running time.

**Comparison Systems or Algorithms:** For overlay construction, we compare five algorithms: VNM, $\text{VNM}_A$, $\text{VNM}_N$, $\text{VNM}_D$, and IOB. For overall throughput comparison, we compare three approaches: (1) *all-pull*, where all queries are evaluated on demand (i.e., no sharing of aggregates and no pre-computation), (2) *all-push*, where all aggregates are pre-computed, but there is no sharing of partial aggregates, and (3) *dataflow-based overlay*, i.e., our approach with sharing of aggregates and selective pre-computation. We chose the baselines based on industry standards: *all pull* is typically seen in social networks, whereas *all push* is more prevalent in data streams and complex event processing (CEP) systems.

[4]Available at `http://ita.ee.lbl.gov/html/traces.html`.

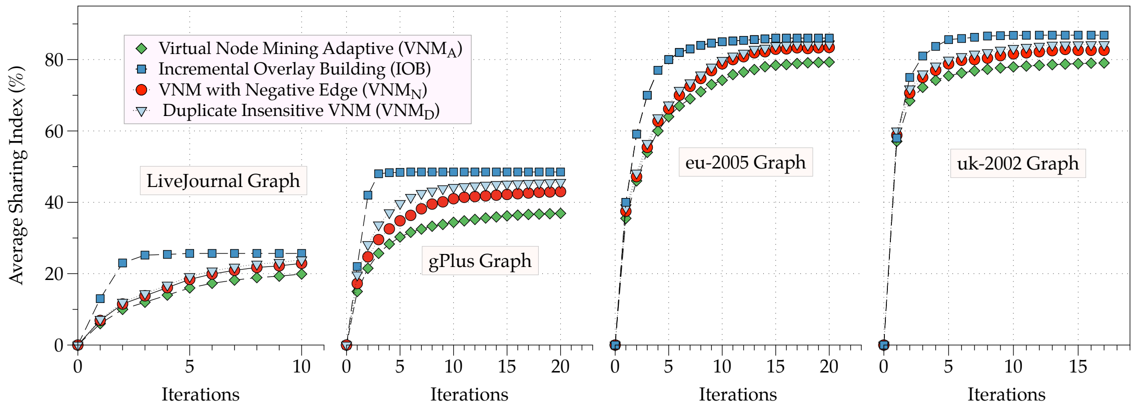


Figure 4.7: Comparing overlay construction algorithms on real networks; (IOB should not be directly compared against VNM$_N$ or VNM$_D$ since it doesn't use negative edges or duplicate paths)

### 4.4.2 Overlay Construction

**Sharing index:** First we compare the overlay construction algorithms with respect to the *average sharing index* achieved per iteration, over 5 runs (Figure 4.7). As we can see, IOB finds more compact overlays (we observed this consistently for all graphs that we tried). The key reason is that: IOB considers the entire graph when looking for sharing opportunities, whereas the VNM variations consider small groups of readers and writers based on heuristical ordering of readers and writers. Note that, IOB should only be compared against VNM$_A$, and not VNM$_N$ or VNM$_D$, since it doesn't use negative edges or duplicate paths. We also note that, for IOB, most of the benefit is obtained in first few iterations, whereas the VNM-based algorithms require many iterations before converging. Further, the overlays found by VNM$_N$ and VNM$_D$ are significantly better than those found by VNM$_A$. This validates our hypothesis that using negative edges and reusing mined edges, if possible, results in better overlays. Another important trend that we see here is that the sharing indexes for web graphs are typically much higher those for the social graphs. Kumar et al. also notice similar difficulties in achieving good structural compression in social networks [58].

**Running time and memory consumption:** Figure 4.8 shows the running time for the different construction algorithms with the increasing number of iterations for the LiveJournal graph. As we can see IOB takes more time for first few iterations, but is overall faster than the $\text{VNM}_A$ and its variations since it converges faster. As expected, both $\text{VNM}_N$ and $\text{VNM}_D$ take more time per iteration than $\text{VNM}_A$. We also compared the total memory consumption of the overlay construction algorithms (not plotted). For LiveJournal, $\text{VNM}_A$ and its variations used approximated 4GB of memory, whereas IOB used 8GB at its peak; this is not surprising considering that IOB needs to maintain additional global data structures.

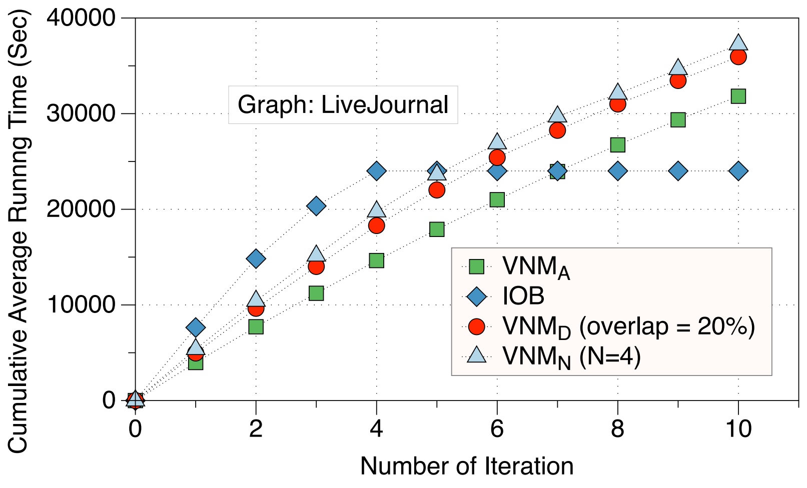


Figure 4.8: Running time comparison of the overlay construction algorithms.

**Comparing VNM and $\text{VNM}_A$:** Figure 4.9 shows SI achieved by our adaptive $\text{VNM}_A$ algorithm and by $\text{VNM}_A$ as the chunk size is varied. As we can see, VNM is highly sensitive to this parameter, whose optimal value is quite different for different data graphs. On

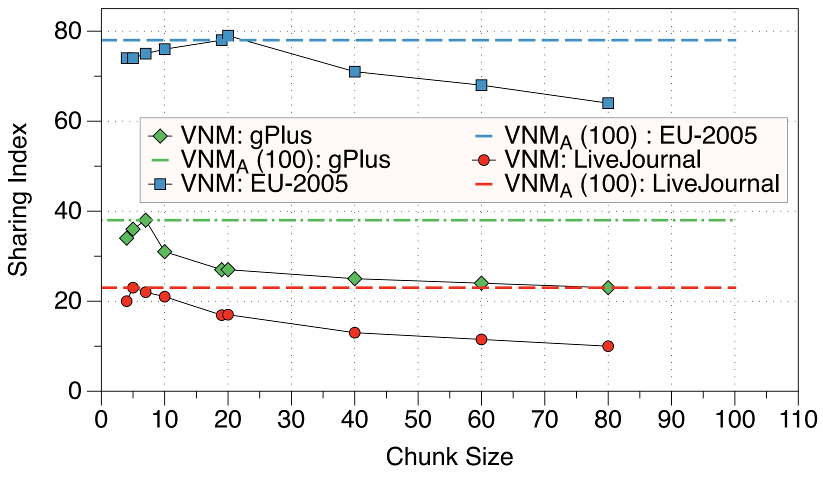


Figure 4.9: Effect of chunk size on VNM;

the other hand, VNM$_A$ is able to achieve as compact an overlay (in some cases, slightly better) as the best obtained by VNM.

**Overlay depth:** Figure 4.10(a) compares the depths of the overlays created by VNM$_A$ and IOB algorithms for one run over the LiveJournal graph. The overlay depth for a reader is defined to be the length of the longest path from one of its input writers to the reader. In the figure, we plot the cumulative distribution of the number of readers at each overlay depth. As we can see, IOB creates a significantly deeper overlay with average depth of 4.66 (vs 3.44 for VNM$_A$); as we will see later, this results in lower end-to-end throughput, even though IOB creates a more compact overlay.

**Microbenchmarking VNM$_N$:** Next we examine the VNM$_N$ algorithm and the impact of the number of negative edges allowed per transaction on the sharing index and running time. We varied the number of allowed negative edges from 1 to 5. As we can see in Figure 4.10(b), allowing for negative edges has a significant impact on the sharing index, and as expected, we do not see much benefit beyond 3 or 4 negative edges. The

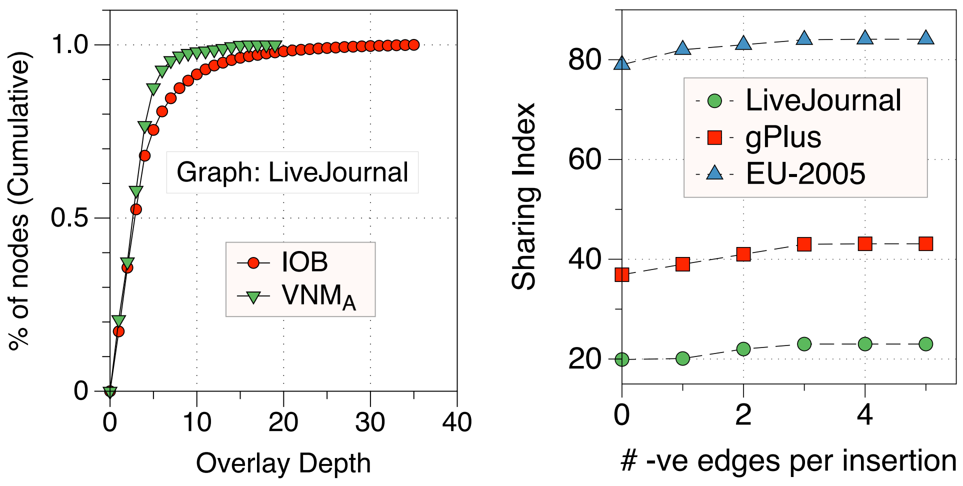


Figure 4.10: (a) Comparison of overlay depth for different overlay construction algorithms. (b) Effect of the number of negative edges on the sharing index.

running time of $\text{VNM}_N$ increases rapidly as we increase the number of negative edges allowed, and almost doubles when we allow 5 negative edges vs none (not plotted). However, we note that overlay construction is a one-time process, and the benefits in terms of increased throughput outweigh the higher initial overlay construction cost.

### 4.4.3 Dataflow Decisions

**Effectiveness of Pruning:** Figure 4.11 shows the effectiveness of our pruning strategy to reduce the input to the max-flow algorithm. Each vertical bar has been divided to show the composition of intermediate overlay nodes and original graph nodes, before and after pruning the overlay that we got using $\text{VNM}_A$ for LiveJournal graph. We get similar results for other overlay construction algorithms and other graphs. The pruning step not only reduces the size of the graph (to below 14% in all cases), but the resulting graph is also highly disconnected with many small connected components, leading to very low running times for the max-flow computations.

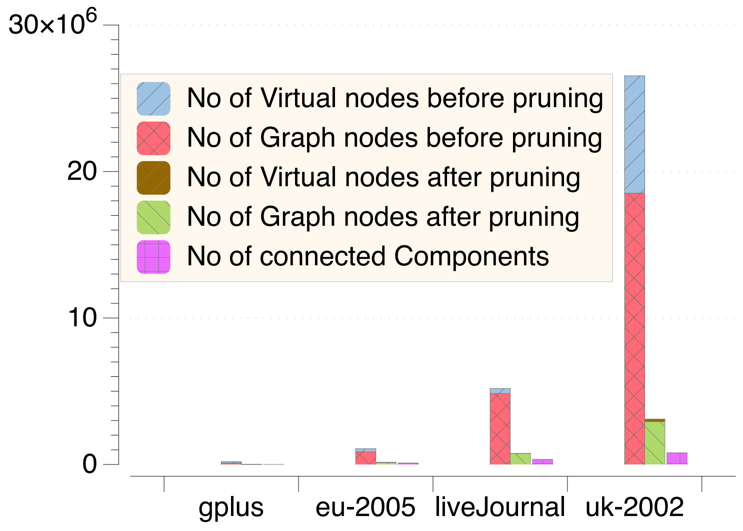


Figure 4.11: Benefits of pruning before running maxflow.

**Adaptive dataflow decisions on a real trace:** Figure 4.12 shows the ability of our proposed adaptive scheme to adapt to varying read/write frequencies. We used the EPA-HTTP network packet trace to simulate read/write activity for nodes. We used average read/write frequencies of the nodes to make static dataflow decisions. At a half-way point, we modified the read/write frequencies by increasing the read frequencies of a set of nodes with the highest read latencies till that point. As we can see, the static dataflow decisions turn out to be significantly suboptimal once this change is introduced. However, our simple adaptive approach is able to quickly adapt to the new set of read/write frequencies.

**Baseline for dataflow decisions:** Figure 4.13 shows the effectiveness of the dataflow decisions on the overlay. In this experiment we kept the number of threads (12) and read/write ratio (1:1) of the queries fixed and computed the average throughput for: (a) overlay with all push, (b) overlay with dataflow decisions, and (c) overlay with all pull. As we can see, for all aggregate functions, overlay with optimal dataflow performs much better than overlay with all pull and all push thereby justifying our hypothesis. We observed

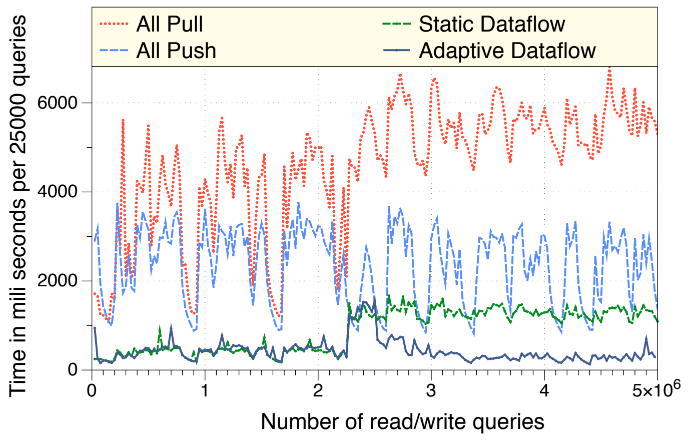


Figure 4.12: Effect of workload variations on different approaches;

similar results for other read/write ratios as well.

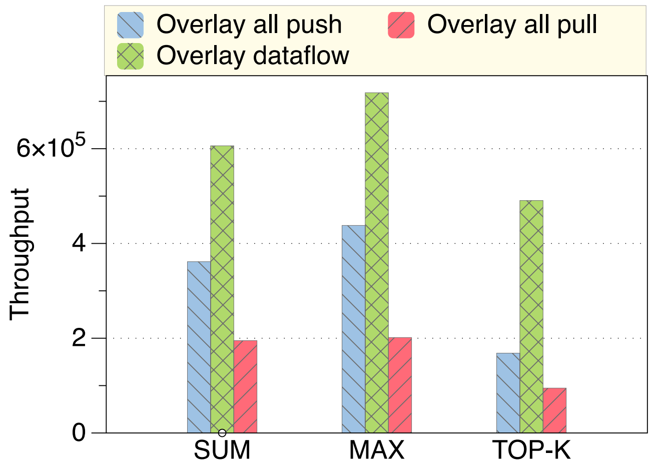


Figure 4.13: Baseline to motivate dataflow decisions;

## 4.4.4 Throughput Comparison

**Varying Read-Write ratio:** Figure 4.14 shows the results of our main end-to-end throughput comparison experiments for the three ego-centric aggregate queries. We plot the throughputs for the two baselines as well as for the overlays constructed by the different algorithms, as the write/read ratio changes from 0.05 (i.e., the workload contains

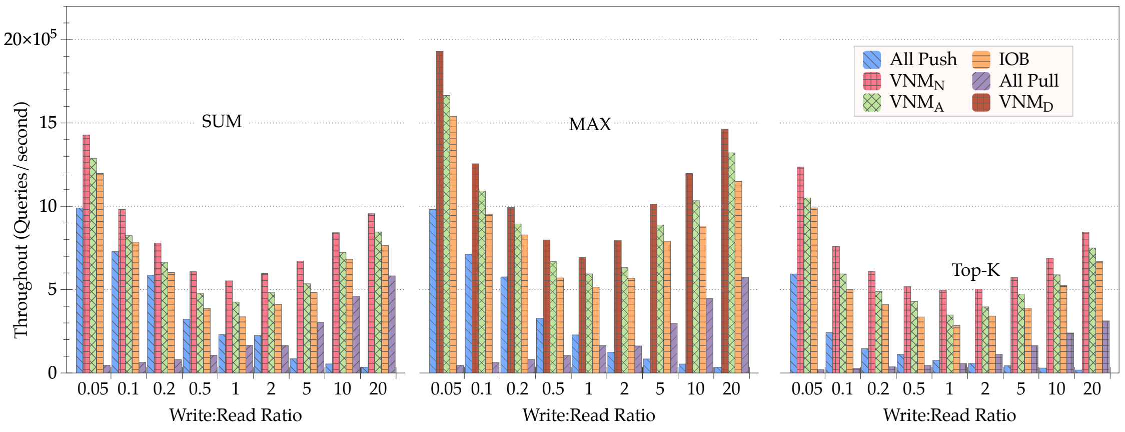


Figure 4.14: End-to-end throughput comparison for different aggregate functions for the LiveJournal graph, with 24 threads;

mostly reads) to 20. As we can see, the overlay-based approaches consistently outperform the baselines in all scenarios. For the more realistic write/read ratios (i.e., around 1), the throughput improvement over the best of the two baselines is about a factor of 5 or 6. For read-heavy workloads, the overlay-based approach is multiple orders of magnitude better than the *all-pull* approach, and about a factor of 2 better than the *all-push* approach, whereas the reverse is true for the write-heavy workloads.

Comparing the different aggregate functions, we note that the performance improvements are much higher for the more computationally expensive TOP-K aggregate function. In some sense, simple aggregates like SUM and MAX represent a worst case for our approach; the total time spent in aggregate computation (which our approach aims to reduce through sharing) forms a smaller fraction of the overall running time.

Comparing the different overlay construction algorithms, we note that VNM$_N$ shows significant performance improvements over the rest of the overlay construction algorithms, whereas IOB is typically the worst; the higher depth of the overlay increases the

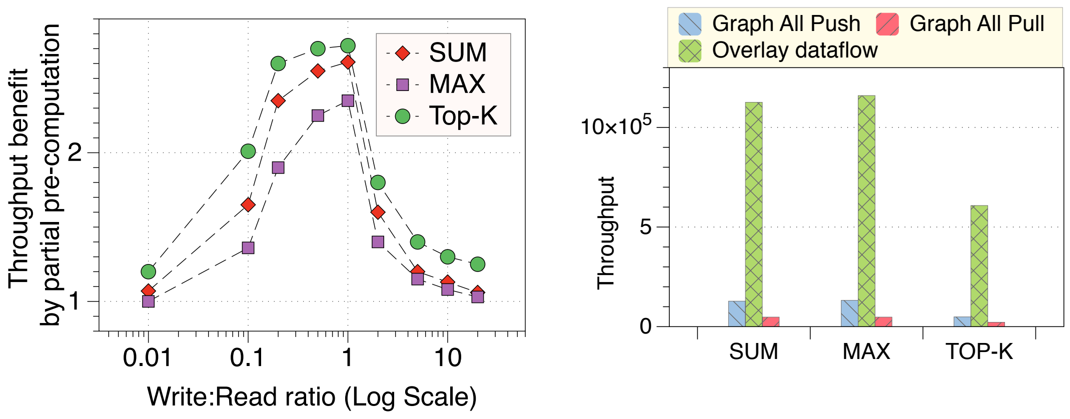


Figure 4.15: (a) Benefits of partial pre-computation through node splitting; (b) Throughput comparison for 2-hop aggregates.

total amount of work that needs to be done for both writes and reads.

**Effect of Splitting Aggregate Nodes:** Figure 4.15(a) shows the effect of our optimization of splitting an overlay aggregate node based on the push frequencies of its inputs (Section 4.3) on the LiveJournal graph. As we can see, for all the aggregate functions, this optimization increases the throughput by more than a factor of 2 when write-to-read ratio is around 1. In the two extreme cases (i.e., very low or very high write-to-read ratios) where the decisions are either all push or all pull, this optimization has less impact.

**Two-hop aggregates:** Figure 4.15(b) shows the throughput comparison for different aggregates specified over 2-hop neighborhoods for $\text{VNM}_A$ overlay compared to all pull and all push; we used the write-to-read ratio of 1 over the LiveJournal graph. The relative performance of the overlay approach compared to all push or all pull is better for 2-hop aggregates than 1-hop aggregate, which can be attributed to better sharing opportunities in such queries.

**Latency:** Figure 4.16 shows the *worst case*, *95th percentile*, and *average* latency for the read queries for TOP-K as the push cost to pull cost ratio is varied. Here we used

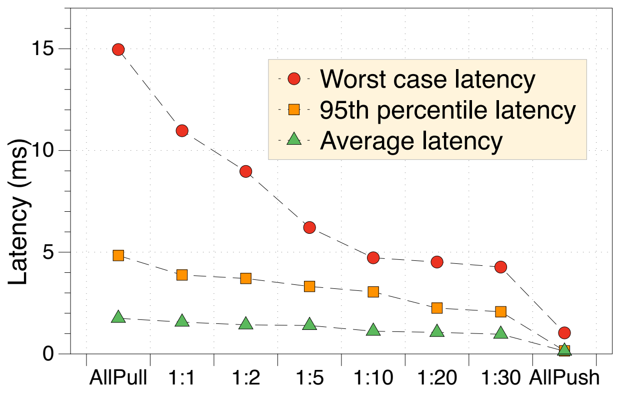


Figure 4.16: Read latency for different push:pull cost;

the network packet trace EPA-HTTP to simulate read/write activity. Since the number of distinct IP addresses in the trace is much smaller than the number of nodes in the (LiveJournal) graph, we randomly split the trace for each IP address among a set of nodes in the graph. We eliminated contention by ensuring that each query or update runs in isolation. As we can see, increasing the *pull cost* bring down the read latencies, as *pushes* get favored while making dataflow decision. We also note that the worst-case latencies in our system are quite low.

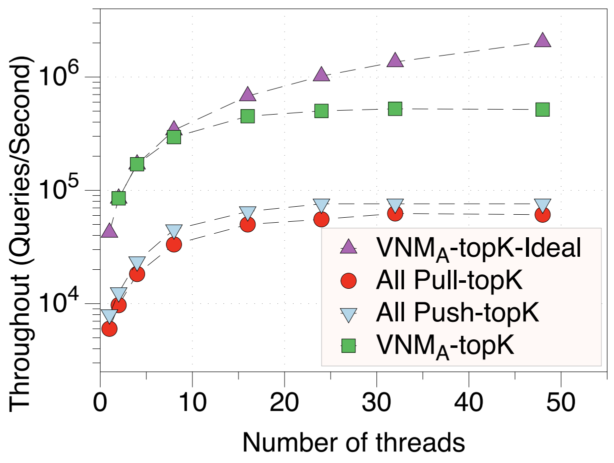


Figure 4.17: Effect of increasing parallelism on throughput.

**Parallelism:** Figure 4.17 shows how the throughput varies as we increase the number of threads serving the read and write requests for the three approaches; we use the TOP-K query over the LiveJournal graph, with write-to-read ratio of 1. Because of the synchronization overheads, we don't see perfect scaleup (note that the $y$-axis is in log-scale); for all three approaches, the throughput increases steadily till about 24 threads, and then plateaus out (our machine has 24 cores with hyperthreading enabled).

Chapter 5

# Activity-based Subgraph Pattern Matching Queries

In this chapter, we present CASQD, which forms a core component of our system and enables succinct specification and continuous identification of active subgraph patterns. We begin with a brief overview of the problem by discussing some background, the data, query, and computation model, and then present the overview of the CASQD framework (Section 5.1). In Section 5.2 we discuss various execution strategies and in Section 5.3 we describe the algorithm for exploring the neighborhood of selected pivots to find the primitive structures. Then we describe our system architecture (Section 5.4), experimental setup and present a comprehensive experimental evaluation (Section 5.5).

## 5.1 CASQD Overview

In this section, we start with providing a background of active subgraph pattern queries, and then discuss the data and query model we use, followed by an overview of our proposed CASQD framework.

### 5.1.1 Background

We start with a toy example (Figure 5.1) of active subgraph pattern queries. Recall that, an active subgraph pattern matching query is defined on the graph structure, as well as on the activities (i.e., message posting activity) of the entities in the graph. The activities could be of different types (not shown here). In this example, the query is to find all 4-cliques such that each node has posted at least one message in the last two hours. The answers need to be produced in a continuous fashion. Given the data in the example, two answers are produced, one at 2 a.m. and the other at 6 a.m., reporting two different 4-cliques. Note that, we want the answer to be reported as soon as possible after the last event that caused a pattern match.

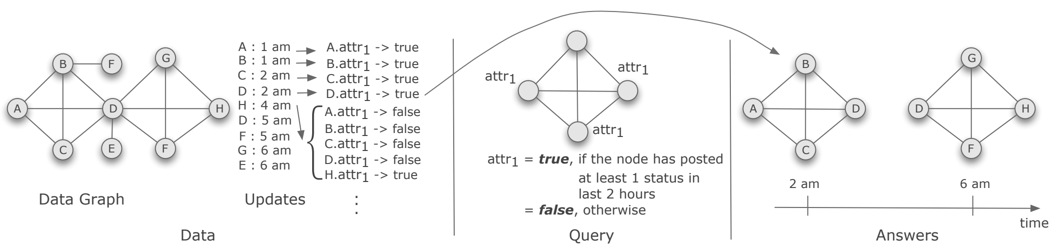


Figure 5.1: Toy example of a *data graph* along with its *activity stream*, *an active pattern query*, and the *answers* produced.

Such queries could be seen as continuous subgraph pattern matching queries over graph-structured data, where the node-attributes are window-based aggregates over their activities (we call these *active attributes*). Surprisingly, to the best of our knowledge, such activity-based subgraph pattern matching queries have not been systematically studied before. The closest line of related work performs subgraph pattern matching on property graphs where node-attributes are categorical labels and they can change over time. While at an abstract level the problems are related, the challenges in query evaluation are fundamentally different. The key distinguishing factor here is the *selectivity of such predicates* and rate of dynamic changes. Categorical attributes in pattern matching queries are typically far more selective than an aggregation-driven acivity-based predicate, and we may expect a few changes every now and then. On the other hand, for an activity-based predicate (like the one in the example above), the number of nodes that satisfy that may constitute a large fraction of all the nodes, and the set of those nodes may undergo rapid and large changes over time. Moreover, computing activity predicates in a streaming fashion can impose significant computational overload by itself.

One natural question is whether these queries can be handled by decomposing them into *structural* and *activity* predicates, and then utilizing existing frameworks to evaluate the two types of predicates independently. Prima facie, this seems to be an obvious direction, considering that there has been much prior work, both in the area of computing streaming aggregates [142, 152, 97] as well as performing subgraph pattern matching. However, such a decomposition-based approach may lose out on key query optimization opportunities. For example, the knowledge about the neighborhood structure of a node, specifically the number of active nodes in its neighborhood, might help in developing effective pruning strategies. Exploiting such optimization opportunities are not supported by either lines of work mentioned above. The stream processing frameworks are excellent at performing stateless aggregation in a distributed fashion, but they don't understand graphs as first-class citizens. Note that our work EAGr [113] (described in Chapter 4) supports computing aggregates of the information produced in the nodes' neighborhoods.

However, EAGr's overlay-based approach is dependent on two key assumptions: (a) the query scope is fixed and pre-defined (e.g., 1-hop, 2-hop ego-net) and (b) the aggregates need to be computed for all the nodes in the network (i.e., typically to support a feature/service for all the nodes in the graph). Such a model is not practical for matching dynamic subgraphs where the query scopes change continuously.

### 5.1.2 Data Model

To reiterate, two main components of our data model are: (a) a property graph that represents the graph structure, and (b) an unbounded activity stream that provides updates to the property graph. However, in this chapter we extend the property graph model by introducing *active attributes* on the nodes and edges. These active attributes are defined on the activity stream. Moreover, there could be multiple active attributes based on different types of events that are present in the activity stream. We also introduce the notion of *graph view* that describes the property graph along with all the user-defined active attributes. We discuss both the notion of *active attribute* and *graph view* next. But, before that we furnish examples of a activity schema and events from activity stream to set a context for the discussions.

Example activity schema: In a cellphone call graph, "phone call activity" could be represented as ⟨**call**, {*caller:String, callee:String, duration:Sec*}⟩. Similarly, when a user $C$ calls $D$ for the first time, it could lead to an entry in the "add edge" activity, with schema ⟨**addedge**, {*caller:String, callee:String*}⟩.

Example events in the activity stream: Examples of two events in a activity stream that conform to the activity schema above are: $\langle$**call**, $\{$*caller:"Alice", callee:"Bob", duration:"10s"*$\}$*, time:"10:00am"*$\rangle$, $\langle$**addedge**, *caller: "Tom", callee:"Nancy"*$\}$*, time:" 1:00pm"*$\rangle$.

**Active Attributes:** Active attributes are defined as aggregation functions over the activity stream. The values of the active attributes change as time progresses. Active attributes could be defined for both nodes and edges. In fact, we let users define composite active attributes using other active attributes. Active attributes are represented using five parameters $\langle\ a_{name}, e_{type}, e_{type}, \mathcal{F}, w\rangle$, where $a_{name}$ is the name of the new attribute; $a_{type}$ and $e_{type}$ denote the activity type and entity type on which $a_{name}$ is defined; and $\mathcal{F}$ represents the aggregation function to compute the value of $a_{id}$ for a specific instance of a graph entity. The aggregation function supports window-based aggregates and the corresponding window size parameter is specified through *w*. Following are some examples:

Example 3: Given the event types in Example 1, following are three different user-defined *active attributes*:

- Node Attribute: An example attribute representing whether a user has made "more than 10 calls in last two hours" would be defined as $(\langle ActiveCaller, Call, Node, SUM, 2\rangle > 10)$.
- Edge Attribute: An example edge attribute representing whether a user has made "more than 50 calls in her lifetime to a friend would be: $(\langle PhoneFriend, Call, Edge, SUM,$-1$\rangle > 50)$ [1].

[1]w = -1 represent time window starting at the beginning of time till now

- Edge Attribute: *Reciprocity* $> 0.5$ of an entity type *Edge* could be defined as $\frac{Min(E.n1.call,\ E.n2.call)}{Max(E.n1.call,\ E.n2.call)} > 0.5$. Reciprocity is maximum (i.e., 1) when $a$ and $b$ call each other equal number of times.

Such multi-level approach not only makes the overall dynamic scope specification simpler, but is also very crucial in query optimization. Arbitrary attribute definition makes it harder to decide on the evaluation order of predicates that involves those attributes. On the other hand, the multi-level approach naturally embeds such evaluation orders and helps achieving the intuitive requirement that the system should compute more expensive activity measures less often than the less expensive ones. Moreover, such approach is naturally suited to handle varying data rates by only making adaptations at the lowest layer. For example, if the data rate increases beyond a system's ingest capability, the system may choose to apply aggressive pruning at the lowest level, whereas the execution logic at other levels will remain intact. Also, as we will see later, such pruning logic is much easier to develop and is more robust for less complex activity measures as compared to the more complex ones. It's important to note that, with aggressive pruning at the lower levels, the system might miss some results. However, this behavior is common for most real-time systems where query-latency is more important and approximate results are accepted. We will revisit this issue again in the search algorithms section.

**Graph View:** Finally, let $\mathcal{G}(V, E, V_p, E_p)$ denote a *graph view*, where $V$ and $E$ represent the sets of nodes and edges, while $V_p$ and $E_p$ represent the set of node- and edge-attributes for each node/edge of the graph. Note this this is effectively an extended property graph model model, however with the modification that, $V_p$ and $E_p$ can contain both active and standard attributes.

To reiterate, the subgraph specification language can only access the graph view to specify the queries, and is hence oblivious of how the active attributes are specified and computed. Even though the active attributes could themselves be programmed, we expect that the two groups using these two interfaces to have different levels of expertise. This decoupling, on one hand, simplifies the query language significantly, making the life of an analyst much easier. And at the same time, hiding away the interface for programming active attributes makes the job of the query optimizer relatively simpler. We believe that such an abstraction is very crucial to support a succinct yet powerful query language on dynamically changing graph structured data.

### 5.1.3 Query Model

A task in CASQD is specified using two things: (a) definition of an active subgraph pattern, (b) an analytical program to be executed on the extracted subgraph.

#### 5.1.3.1 Specifying Active Subgraph Pattern

An important aspect of designing a suitable specification language is the tradeoff between expressibility and the implementation efficiency of the language. From the expressibility point of view, we can adopt any query model that has been used in existing subgraph pattern matching literature. However, adopting a language that can express arbitrarily complex queries might be a significant hindrance to efficient implementation. Keeping that in mind, we take a departure from any existing query language, and focus on a set of high-level primitives, specifically, active cliques, active stars, active bi-partite cliques. The primary reasons behind this were:

**Reason 1:** These primitives cover many of the real-life use cases [27, 87, 82]. More importantly, due to the regular shapes of these structures (in contrast to arbitrary subgraphs), they are relatively easier to execute efficiently against very high-rate data streams. Table 5.1 provides intuitive interpretations and use-cases of the motifs that we used for our primitives.

| Motifs | Interpretation |
|---|---|
| Star | A central node that connects to a set of peripheral nodes. Such a motif typically represents one-to-many relations, for example, celebrities having many followers, a salesperson who calls many potential customers. |
| Clique | A group of nodes where every node is connected to every other nodes in the group. Such a motif represents a close-knit group, for example a group of social network users who went to school together, a set of researchers who colludes to increase their citations by citing each other. |
| Biclique | Two groups of nodes where every node of one group is connected to every node in the other group. Such motifs represent a group of entities, of one type, that are similar w.r.t. another set of entities of a different type. Example includes: a set of Facebook users subscribing to common set of Facebook pages, a set of researchers all of how have published on a same set of research topics, etc. |

Table 5.1: Graph Patterns

**Reason 2:** User-specified analytical tasks could be combined with these *primitives* to significantly improve the expressibility of our approach. Specifically, the result instance of the primitive subgraphs could be used as seed patterns to search for relatively complex query pattern. Figure 5.2 shows an example where the complex subgraph pattern could be decomposed into a primitive structure (i.e., an approximate 4-clique) and some additional

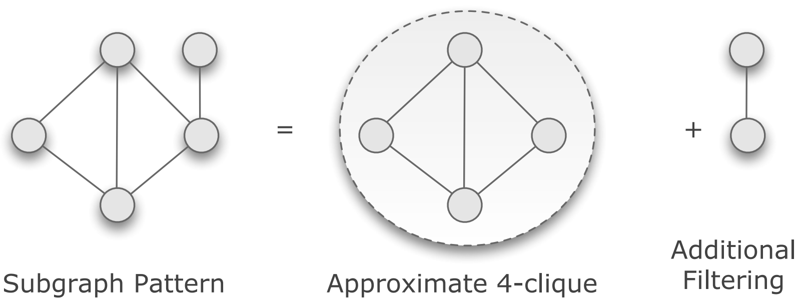


Figure 5.2: Query Decomposition Example

structural predicates. In this case, the analytical task could be programmed to do the additional filtering operations on each of the identified approximate 4-cliques. Another example is to check whether the spoke nodes are well-connected in a detected active star.

**<u>Reason 3:</u>** Surprisingly, these primitives can't be *very succinctly* expressed by the existing query languages. Most of the work [138, 59] uses bijective functions to specify the query graph, following the definition of subgraph isomorphism. Such specifications involve furnishing a set of nodes, edges, and node/edge to attribute mappings. While they can specify any arbitrary subgraph, it could be unnecessarily cumbersome to specify the primitives we consider. For example, specifying an active clique of size *n* using the above approach would be very verbose. Although special constructs could be used to specify complete n-cliques, the issue remains for specifying *approximate n-cliques* (i.e., having $\leq \binom{n}{2}$ edges).

**<u>Reason 4:</u>** These primitives could be incorporated in any query language that deals with subgraph pattern matching queries, essentially providing a balance between expressivity and the implementation efficiency of the query language.

Driven by these reasons, in this work we primarily focus on continuous identification of a set of *primitive active structures (i.e.,* $intermediate$ *nodes).* Our system currently supports the following set of $intermediate$ nodes.

- $Clique(n, Attr_p, Attr_e, f)$: A clique, consisting of $n$ or more nodes, which contains at least $f{*}\binom{n}{2}$ edges. Each node must satisfy the active node attribute $Attr_p$. Each edge in the clique has the optional edge attribute $Attr_e$.
- $Star(n, Attr_p, Attr_s, Attr_e)$: A star, consisting of at least $n$ nodes. The central node must satisfy node attribute $Attr_p$, and the peripheral nodes must satisfy $Attr_s$.
- $Biclique(n, m, Attr_l, Attr_r, Attr_e, f)$: A bipartite clique, equal or bigger than $n \times m$, which contains at least $f * n * m$ edges. The nodes in the left-hand side must satisfy $Attr_l$, and the nodes in the right-hand side must satisfy $Attr_r$.

Note that we leave out *trees* from primitive structures as they can be represented as compositions of *stars*. In this work, we leave it to the user to write appropriate analytical tasks to find tree patterns using results from star patterns. In this dissertation, we mainly focus on the scalable system design and a systematic study of the algorithmic challenges for continuous detection of these three primitives.

#### 5.1.3.2 Analytical Task

The analysis programs are specified as a Java program against the BluePrints API [3], a collection of interfaces analogous to JDBC but for graph data. Blueprints are a generic graph Java API used by many graph processing and programming frameworks (e.g., Gremlin [10], a graph traversal language; Furnace [7], a graph algorithms package; etc.). The analysis task has access to historical information about the graph (which is stored in a separate data store), and can use that to do a thorough analysis of the identified pattern.

### 5.1.4 Answer Reporting Model

Like all continuous query systems, our system also reports answers as soon as they are available. However, the use case here is different from the answering model used in anomaly detection queries. In our case, since the active attributes could be defined using sliding windows, an active pattern might remain valid for a long period of time. Based on this observation, CASQD supports two different answering models: (i) *activity-aware mode*, and (ii) *one-time mode*. In the activity-aware mode, an extracted subgraph *S* is continuously monitored for the satisfiability of the pattern match condition. On the other hand, in the *one-time mode*, the pattern match is reported once and then forgotten. For example, the *activity-aware mode* could be useful for social advertisement targeting where one could continue to target advertisements to a subgraph *S* as long as S remains a match. On the other hand *one-time mode* would be relevant for a *spammer detection* application where the spammer is blocked from the system once a spammer is identified in the matched pattern.

It is important to note that, in both the modes, only *newly active patterns* are reported as results. For example, in case of *one-time mode*, if after detection, a pattern remains active, it won't be reported again. However, if the pattern becomes inactive, and then active again, it will be reported as a result.

### 5.1.5 Computation Model

The scalability of our framework is dependent on the ability to efficiently match *regular active subgraphs*. Keeping that in mind, we took the exploration-based approach over a join-based approach. The main reasons are:

- Join-based approach requires decomposing the query graph and finding answers to the decomposed components, followed by join operations to answer the queries. While join-based approach works well for structural queries with static attributes (infrequently changing attributes), it doesn't work as well in our case. This is due to low selectivity of the active attributes–the size of the intermediate results would too large to handle. Moreover, due to the dynamic nature of the active attributes, there would be too many updates to the indexes needed to answer the joins.
- Join-based approach is also not ideal for the answer reporting model we need to implement, especially the *activity-aware* mode. Join-based approaches are not suitable for identifying when a answer subgraph fails to satisfy the matching condition.

Instead, we use a *monitor, explore and trigger*-based approach.

**1. Monitor:** The models abstract the activity behavior of the nodes and/or the activity behavior of the neighbors of the node. The monitoring strategy could be either *hard* or *soft*. Hard monitoring accurately captures the activity behavior of the nodes and its neighbors, by eagerly evaluating the activity predicates of the nodes as well as by keeping track of how many neighbors are currently active. In contrast, the soft monitoring involves computing a probabilistic "estimate" the activity behavior of the nodes using historical information or other available methods. These estimates for nodes could also be used to maintain an estimate about the number of active neighbors w.r.t. a relevant active predicate.

We emphasize that irrespective of whether the monitoring strategy is soft or hard, they always guarantee correctness under the model we have specified. It is important to note that, soft monitoring, if accurate, can save us a lot of work by avoiding eager computation of activity predicates.

**2. Explore:** The explore phase starts when there is a potential to find a match around a node $n$, based on its model as well as the knowledge about its neighborhood. Such a node is called a *pivot node*. The explore phase search around the pivot node for a match and reports the result if it finds one. Additionally, it also updates the knowledge about the pivot node's neighborhood, based on the information gathered during the exploration. Note that, this is done even when the exploration fails to find a match.

**3. Trigger:** The trigger is used to let other nodes know about any significant change in the activity behavior of a node. For example, if a node $n$'s neighbors' current belief is that $n$ is active w.r.t. an activity predicate, but $n$ finds out it is no longer active, all of $n$'s neighbors may be triggered to update their neighborhood knowledge.

The main advantage of the monitor-explore-trigger approach is that it allows us to systematically prune the search space.

- **Pruning pivots based on node activity:** In this case, we prune out a node from the consideration of being a pivot node, if the node is expected not to satisfy the activity predicate (i.e., the corresponding active attribute evaluates to be false).
- **Pruning pivots based on neighbors' activity:** In this case, we prune a node from consideration depending on information available about the node's neighbors. For example, if we know that the expected number of active neighbors of a node $N$ is

much less than $k$, then we know that $N$ is unlikely to participate in $k$-clique.

- **Pruning new activities based on active predicates:** We can disregard a new activity by a node $n$, if we know that the current value of corresponding active attribute on $n$ evaluates to *true*. Immediately processing the new update to recalculate the value of the active predicate won't increase the chance of finding a pattern match pivoted at $n$. However, the activity may need to be processed if a relevant active attribute is *false*.
- **Pruning new activities based on structural predicates:** We can also disregard a new activity by a node $n$, if we know that $n$ doesn't have enough neighbors (both active or inactive w.r.t. an active predicate) to satisfy a pattern match.

### 5.1.6 Correctness Model

In this section, we discuss our correctness model. There are two primary issues to consider for subgraph pattern matching on dynamic graphs: (a) whether the system *misses* any result, and (b) whether the system reports any *wrong result*. We first describe our assumptions about the system/module that delivers the activity stream to CASQD, and then state our correctness model under both single-threaded and multi-threaded environments.

**Assumptions:** Two very important notions in a stream processing system are *event time* and *stream time*. Event time represents the record of system clock time (for whatever system generated the event) at the time of occurrence. On the other hand, stream time is the time at which an event is observed by the stream processing system, i.e., the current time according to the system clock. In an ideal world, we would expect the event time and stream time to be the same, however, this is not the case for most real-world applications. In our work, we make an assumption that our system sees all the events in *stream time*. Moreover, we assume that there are no out of order events, and the platform that feeds the data to CASQD, takes care of that issue.

**Definition 1.** $\{$*Strong Correctness*$\}$*: Assuming events are perfectly ordered in stream time, strong correctness is characterized as: (a) no results would be missed, and (b) no wrong results would be reported.*

### 5.1.6.1 Correctness in single-threaded Model

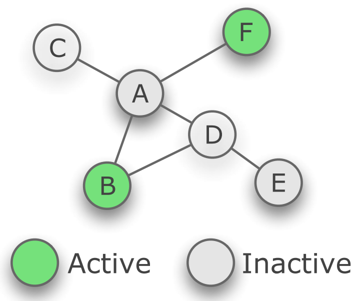
**Correctness Model 1.** *Assuming single threaded execution,* CASQD *guarantees that no results are missed, and no reported results are wrong.*

Specifically, this guarantee holds for all different *monitoring* and *exploration* algorithms we use. Moreover, we exploit the optimization opportunities mentioned in the previous section, without compromising the correctness. We discuss the correctness proof in Section 5.2, where we describe the algorithms in details.

### 5.1.6.2 Correctness in Multi-threaded Model

For multi-threaded environment we start by stating how correctness could be compromised.

**Missing results:** We illustrate this case using the example in Figure 5.3. Figure 5.3(a) depicts the state of the graph at time $t_0$ and 5.3(b) represents a set of new events ordered in stream time. At time $t_0$, the green nodes are active while the others are inactive.



(a) Graph at time $t_0$

| Activity | Stream Time | Assigned Thread |
|---|---|---|
| Node C => Active | $t_1$ | T1 |
| Node A => Active | $t_2$ | T2 |
| Node E => Active | $t_3$ | T3 |
| Node B => Inctive | $t_4$ | T4 |
| Node D => Active | $t_5$ | T5 |

(b) Activity Stream after $t_0$

Figure 5.3: Correctness model example for a toy 4-star query

The query of interest is an active 4-star (i.e., one central and 3 peripheral nodes). Now under the assumption of serial execution, we should have the answer $\langle A, \{C, B, F\}\rangle$ at time $t_2$. However, if the thread $T2$ is scheduled before $T1$, $T2$ would miss the result, as it sees that only 2 of A's neighbors are active.

**Wrong results:** Using the same example, if $T1, T2,$ and $T3$ are executed in order, but, $T_5$ is scheduled before $T_4$, it would wrongly report the result $\langle D, \{A, B, E\}\rangle$. In reality, however node $B$ is not active.

Providing strong correctness guarantees in a concurrent setting is analogous to providing strong consistency (i.e., ACID properties) for concurrent database transactions. But, our problem is much harder. Of the two problems, guaranteeing that no wrong result would be reported is easier to handle than guaranteeing that no result would be missed. Both problems can be solved using locking, but would impose a significant performance overhead. Due to these reasons, we don't provide a strong correctness guarantee under the multi-threaded model. Moreover, occurrences of these events are very rare, and most applications can tolerate a low degree of such error. For completeness, we provide solutions for how strong correctness could be guaranteed under multi-threading.

**Lemma 5.1.1.** *Taking a lock on the neighborhood (1-hop for star and clique queries, 2-hop for biclique queries) of a node, before its neighborhood would be explored, is sufficient to guarantee that no wrong results would be reported.*

**Lemma 5.1.2.** *To guarantee that no results would be missed, it is necessary (a) to take a lock on the neighborhood (1-hop for star and clique queries, 2-hop for biclique queries) of a node before its neighborhood would be explored, and (b) to ensure that all events, that happened before the event that caused the exploration, have already been processed.*

## 5.2 Execution Strategies

In this section, we discuss various execution algorithms. Recall that our execution model follows a 3-step process: (a) explicitly or implicitly *monitor* the behavior of the nodes and decide which nodes qualify to be potential pivots, (b) *explore* the neighborhoods of the pivots to find pattern matches, and (c) decide whether any of the active neighbors of the current pivot should be considered for exploration and *trigger* them if needed.

### 5.2.1 Node Activity-based Execution (TAP)

The simplest execution strategy is to explicitly monitor the behavior of the nodes and explore their neighborhoods whenever they become active.

**Monitor:** This approach requires eager computation of the window-based aggregation predicate whenever there is an activity. While this is a simple strategy, it might perform a lot of explorations without any match, especially if there are many nodes that don't have enough active neighbors, or not enough neighbors. Moreover, computing a window-based aggregate for each node might be very costly, especially when we could use structure-based pruning to rule out nodes that can never be a part of a pattern match. While TAP loses out on potential optimization opportunities, it doesn't have to maintain any additional indices to store pruning-related information, moreover, those structures don't need to be updated in the face of dynamic updates to the graph. Additionally, structural pruning could be often expensive depending on the query. For example, if the node has *n* nodes, finding out whether the node contains any *k*-clique (with $k < n$), would be a computationally expensive task.

**Explore:** The explore phase involves looking for the required primitive pattern in the pivot node's neighborhood. Note that the exploration algorithm will proceed differently depending on whether we are looking for star, clique, or bi-clique. We discuss the details of these three exploration algorithms in Section 5.3.

**Trigger:** In the trigger step, we decide whether any other node in the neighborhood of the pivot node should be explored as well. Let's look at an example (in Figure 5.4) to make things clear. Here, we consider 3 different primitive queries and in all cases, the event we are currently handling is on node *A*, which has become newly active.

In case of the *Star* primitive, we need to trigger all the active neighbors of A, as there could be new active stars centered around each of the active neighbors. In this example, A's exploration won't find a star centered at A. But, the fact that A become active, there is a new active star centered at *D*, as *D* had already two active spokes, and it was waiting

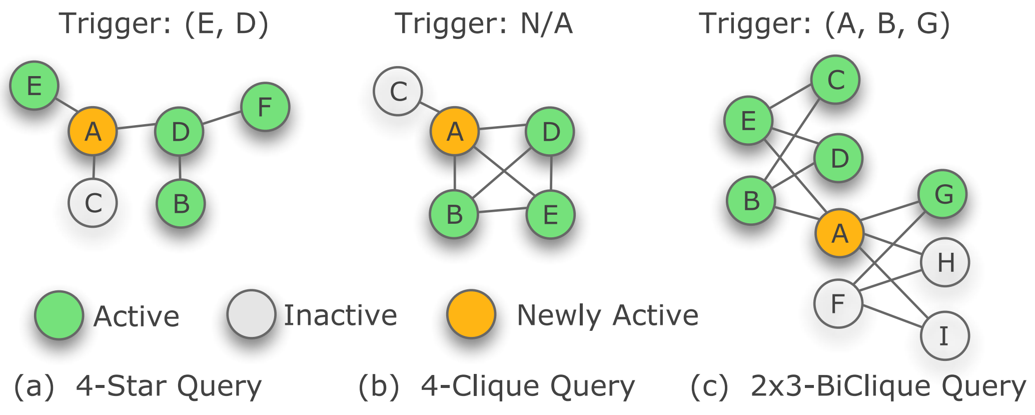


Figure 5.4: Trigger examples for Star vs Clique vs Biclique

for another of its neighbors to become active.

In the case of the *Clique* primitive, however, no trigger is necessary. Exploring A's neighborhood is enough. This because none of A's neighbors can find new 4-Clique without A. If there were 4-cliques without requiring participation from A, it would have been found before.

Finally, in the case of the *Biclique* primitive, again all of A's active neighbors have to be explored as well. Similar to star queries, and unlike clique queries, biclique queries are asymmetric, and hence each of the active neighbors can participate in different bicliques. In this example, A can't find any active 2x3 biclique where it is on the left-hand side. However, A being active would lead to a new 2x3 active biclique where A is on the right-hand side. However, if we notice carefully, not all of A's active neighbors need to be explored. We could apply some optimization here. In this example, only exploring *E* for a biclique suffices, as exploring *B* would lead to a duplicate answer. In such cases, we can perform these neighbor explorations serially, and discard exploration for a node if it has already featured in the left-hand side of an active biclique.

**Correctness proof:**. It is easy to prove that, under serial execution TAP will never miss a result and will never produce a wrong result. The proof follows from the fact that, for *star* queries, a new node N becoming active can lead to two types of answers: (a) where N is the center of the star, and (b) where N is a spoke of a star. Our algorithm ensures both the cases. Also, in case of serial execution of events, the state of the graph doesn't change while exploring N's and its active neighbors' neighborhoods. And hence, there is no chance of missing any result or reporting any wrong result. Similar argument can be given for both *clique* and *biclique* queries and we omit the detailed proof for brevity.

### 5.2.2 Neighbors' Activity-based Execution (TAN)

Now we move to the discussion of the second execution strategy where we try to address the shortcomings of the first approach.

**Monitor:** In this case, our monitoring strategy is a bit complex. We not only monitor the behavior of every node, we also keep track of the number of active neighbors of each node. As before, we eagerly compute the window-based aggregate whenever a node does an activity. Additionally, if a node changes state (i.e., become active from inactive or inactive from active), we update the statistics of each of its neighbors. Note that, in this strategy we are performing a lot more work in the monitor phase, as compared to our first strategy. However, keeping the neighborhood activity statistics updated helps in pruning out unnecessary explorations. Another shortcoming of this strategy is extra memory requirements. We need extra memory to store the number of active neighbors of a node.

In fact, if there are $n_a$ active attributes defined, for each node, $n_a$ neighborhood counters need to be maintained.

**Explore:** The exploration step is initiated when a node becomes active, as long as it currently has a sufficient number of active neighbors. For example, for a *n-star* query we need at least *n-1* active neighbors. In fact, if the node has *n-1* active neighbors, the exploration phase just fetches those neighbors. We provide a detailed discussion of the exploration strategy when more that *n-1* nodes are active in the next section. For a *n-clique* and a *n*x*m-bilcique* queries, we can't guarantee answers whenever the pivot has *n-1* neighbors active. However, we still need to perform the exploration to guarantee that we don't miss any results. If an application can tolerate occasional missing answers, we can increase these thresholds for clique and biclique queries.

**Trigger:** The trigger function for this strategy is same as our first strategy, with the additional pruning information that we only trigger those nodes who are themselves active, as well as have required number of active neighbors.

**Correctness proof:** The correctness argument of this strategy is same as the first strategy as well. Although we perform much less exploration than the first strategy, we only avoid those explorations where it is guaranteed that there is no possibility of finding a result. This guarantees that we won't miss any result due to exploration pruning. Moreover, with the assumption of serial execution, it is guaranteed that, at any given time, the number of active neighbors of any node is accurate.

### 5.2.3 Model-based Execution (TMB)

In this section, we discuss our model-driven execution. The goal here is to model the activity behaviors of the nodes. The high level motivation is to use these models as a proxy for predicting when a node is active, rather than eagerly computing their window-based aggregate to decide whether they are active. A variety of different strategies could be adopted to model activity behaviors. Moreover, different models could be useful for different types of aggregation functions. We won't discuss those issues here, as they are orthogonal to the focus of our current work. Here, without any loss of generality, we will use histograms to model node activities, and we propose to use historical information about the nodes to populate such histograms. The main functionality that a model *M* needs to support is *Model.predict (Attribute attr, nodeId n, timeStamp t)* which provides an expected value of the attribute *attr* at any given time for any node. Such a model could be further used to compute the *expected number of active neighbors* for a given node.

**Monitor:** Before describing the model-based monitoring strategy, we must mention that, if the model is *accurate*, i.e., it never makes any wrong prediction, then we *don't need to compute the values of the active attribute*. The system can run only using the model, and can produce all the correct answers. However, the predictions are treated as best-effort and will likely have errors.

- If a node is predicted to be active, but it's actually inactive, it doesn't hurt the correctness. At most, the exploration pivoted at that node would fail to produce a result, but it won't miss any correct result.

- On the contrary, if a node a predicted inactive, but it's actually active, and if we decide not to explore that node, we might miss a result pivoted at that node.
- Similar arguments hold for the predicted number of active neighbors. If the number is overestimated, we can still provide the correctness guarantee.
- However, if the number is underestimated, we run into the risk of missing results.

Based on these observations, we take a hybrid approach in our model-based monitoring strategy.

- If a node is predicted active, and if the node is also predicted to have a sufficient number of active neighbors, we explore that node. Note that, this step alone can't guarantee correctness; if the number of active neighbors is underestimated, we might miss results.
- On the contrary, if a node is predicted inactive, we always eagerly compute the value of the active attribute for that node, as we can't afford to miss exploring that node when it is actually active.
- Additionally, when a predicted inactive node N becomes active, we proactively let all its neighbors know that N has become active when their models think that N is inactive. This ensures that, the number of active neighbors of a node is always overestimated, and no results are missed due to wrong estimation of the number of active neighbors.

**Explore:** In model-based execution, a node would get explored because of two reasons: (a) the exploration decision was purely based on the model, and (b) the exploration decision was because a node became active when it was predicted to be inactive.

In the first case, we note that the active attribute value of the node is *outdated*, since the exploration decision was purely based on a model. So, we first compute the actual active attribute for the pivot node. Note that, this is a *lazy attribute computation*. By deferring aggregate computation we save on the number of events that need to be ingested by the aggregate computation. This deferred aggregate computation requires that the activity data for those nodes needs to be kept around for the current window. If the attribute computation reveals that the node is indeed active, the exploration continues. During exploration, the neighbors might have eagerly or lazily maintained active attribute. For all the lazily maintained neighbors, we will have to compute their attributes as well.

In the second case, the attribute for the pivot node is already computed, however, due to exploration it may need to perform lazy attribute computation for some of the explored nodes.

**Trigger:** The trigger function of this strategy is same as TAP. For the star and biclque queries, all of the pivot's neighbors are explored. And the exploration algorithm takes care of the rest.

**Correctness proof:** The correctness of the model-based strategy follows from the execution algorithm described above. The keys to correctness are: (a) never discarding a node from exploration based on the model; if the model predicts that a node is inactive, we don't trust the model and evaluate whether the node is actually inactive, (b) never underestimating the number of active neighbors; it follows from the fact that when a node becomes contrary to the model's belief, we update the neighbor statistics according to that.

## 5.3 Exploration Algorithms

In this section, we discuss the exploration algorithms for finding different primitive patterns. The inputs to the exploration algorithms are: (a) the pivot node $p$, around which an active pattern needs to be found, (b) a primitive query instance $q$ (that contains information like whether it's a star/clique/bi-clique pattern, required size, approximation factor, what active attributes it is defined on, etc.), and (c) the reference to the current state of the graph *G*.

### 5.3.1 Star Primitive

Recall that, a typical signature for an active star primitive looks as $Star(n, Attr_p, Attr_s, Attr_e)$, where *n* is the size of the pattern (including the pivot node), $Attr_p$ is the active attribute of the pivot node, $Attr_s$ is the active attribute of the spoke nodes, and $Attr_e$ is the active attribute for the edges. For simplicity, without any loss of generality, we will assume that the active attributes of the pivot and the spoke nodes are same, and that the edge-attribute is true for all edges. We make the same assumption while discussing the rest of the exploration algorithms. Both of these assumptions are straightforward to relax.

**Exploration when n-1 neighbors are active:** Given the above star query, the exploration is straightforward when exactly *n-1* neighbors the pivot are active. The exploration simply reports an answer comprising of the pivot and the active spoke nodes. However, if lazy evaluation is being used (e.g., in TMB), then we first confirm that the neighbors are actually active before reporting the match.

**Exploration when $>$ n-1 neighbors are active:** However, the situation is a bit tricky when $k$ neighbors (where, $k > n - 1$) of the pivot are active. In such a situation, there would be $\binom{k}{n-1}$ number of possible *n-stars* pivoted at *p*. In our case, we report all possible *n-stars*. However, depending on the application requirements, it's easy to configure the system to return only one result. Considering an example from a celebrity account on social media, it's expected that a large number of the celebrity's followers would be active. When the celebrity becomes active, it matches a new active star. In such a situation, the application may prefer one big active star rather than all possible *n-star*s. But, there could be other applications that require all *n-star*s.

### 5.3.2 Clique Primitive

A typical signature for an active clique primitive looks as $Clique(n, Attr_p, Attr_e, f)$, where *n* is the size of the pattern, including the pivot node, $Attr_p$ is the active attribute all the nodes, $Attr_e$ is the attribute filter for the edges, and *f* is the approximation factor. Recall that, *f* indicates that out of $\binom{n}{2}$ edges in a *n-clique*, what percentage of edges would still return a match.

**Exploration when n-1 neighbors are active:** Given, an *n-clique* query, the exploration is relatively straightforward when exactly n-1 neighbors are found active. We just need to count and confirm whether there are $f * \binom{n}{2}$ edges in the subgraph (say, $I_p^g$) induced by *p* and its active neighbors. While counting edges for the neighbors one by one, if at any stage we find that, we are missing an aggregate of more than $(1.0 - f) * \binom{n}{2}$ connections, we terminate the search process, and conclude no active *n-clique* is present pivoted at *p*. Note that, this is assuming $f > 0.5$. If $f < 0.5$, we would return a result as soon as we find more than $f * \binom{n}{2}$ edges.

An interesting question though is, in what order we should start investigating the nodes in $I_p^g$, in order to minimize the number of nodes we need to investigate.

**Lemma 5.3.1.** *The optimal order for clique exploration is to investigate the nodes* $c \in I_p^g$*-*$\{p\}$*, in increasing (decreasing) order of* $Sim_{I_p^g}(c, p)$ *for* $f > 0.5$ *(*$f < 0.5$*), where* $Sim_{I_p^g}(c, p)$ *is the number of common neighbors of* $c$ *and* $p$ *in* $I_p^g$*.*

However, trying to follow that order is mostly useless in our setting, as we can't afford to precompute this order. The induced subgraph $Sim_{I_p^g}$ would would change very dynamically. Moreover, for small size patterns such ordering might not give any significant benefit. But, this ordering would more useful in a static setting where there is need to look for large cliques.

**Exploration when > n-1 neighbors are active:** Similar to the *n-star* case, having more that *n* (say $k$) neighbors active, we will have to look into $\binom{k}{n-1}$ different subgraphs to find active cliques. If there are a large number of active neighbors, then this process may turn out to be very expensive, however, we don't consider that scenario in this work.

## 5.3.3 Biclique Primitive

A typical signature for an active biclique primitive is: $Biclique(n, m, Attr_l, Attr_r, Attr_e, f)$, where *n* is the size of the left-hand side (the side also includes the pivot *p*), *m* is the size of the right-hand side, $Attr_l$ is the active attribute of the left-hand side nodes, $Attr_r$ is the attribute of the right-hand side nodes, $Attr_e$ is the attribute filter for the edges, and $f$ is the approximation factor of the bipartite clique. Here $f$ indicates that out of $n*m$ edges in a *n*$\times$*m-biclique*, what percentage of edges would still return a match.

**Exploration when n neighbors are active (case $f = 1.0$):** Exploring a node to find bipartite cliques is a bit more complex than the other two queries. This is because, for finding bicliques we have to consider nodes that are 2 hops away from the pivot *p*. When exactly *n* neighbors of node *p* is active, we have already found a $n \times 1$ biclique. Now, the goal is to find whether the $n$ neighbors have $m - 1$ more active neighbors. Basically, the problem is to find out the set $\mathcal{S} = \bigcap_{s \in N_p} N_s$, where $N_i$ is the set of all active neighbors of node $i$. If $|\mathcal{S}| \geq m - 1$, then we have $\binom{|\mathcal{S}|}{(m-1)}$ different *n*$\times$*m-biclique*s.

**Exploration when n neighbors are active (case $f < 1.0$):** The problem of finding an approximate bipartite clique is much different from the clique problem. This is because, for cliques the search space is limited to the active neighbors of the pivot *p*. In case of biclique, the search space includes all the 2-hop active neighbors of *p*. If $\mathcal{H}_p^2$ is the set of all 2-hop active neighbors of *p*, we have to consider $\binom{\mathcal{H}_p^2}{(m-1)}$, subgraphs for a matching approximate active bicliques. Similar to the clique case, when $f < 1.0$, we can ask the same question: what is the optimal order for investigating the search space for finding an approximate biclique. To simplify, here the size of the right-hand side is fixed (comprising of n nodes, let's call this set $R_n$), the left-hand side has $m$-1 (let's call this set $L_{m-1}$) nodes (the other one is the pivot *p*), and we want to find out whether there exist an approximate biclique. The problem is to decide the investigation order of the nodes in the left-hand side.

**Lemma 5.3.2.** *The optimal order for biclique exploration is to investigate the nodes $c \in R_{m-1}$, in increasing(decreasing) order of $|R_n \cap N_c|$ for $f > 0.5$ ($f < 0.5$), where $N_c$ is* *the set of all active neighbors of c.*

**Exploration when $>$ n-1 neighbors are active:** Similar to the *n-star* and *n-clique* case, having more that *n* (say $k$) neighbors active, we will have to look into $\binom{k}{n-1}$ different subgraphs to find active bicliques with *p* featuring in the left-hand side of the biclique.

## 5.4 System Architecture

Figure 5.5 shows the high level architecture of CASQD. The key components of the system are as follows.

### 5.4.1 In-Memory Data Structures

The main modules of CASQD work with a set of key in-memory data structures. This leads to a loosely coupled system design, where different modules communicate through the in-memory data structures. Following are the descriptions of the key data structures.

**Graph Data:** The graph data holds mainly the graph structure. We use the adjacency list representation and store them in Java HashMap. Node is used to look up a node and it neighbors.

**Node Activity Index:** The node Activity Index contains the activity levels of various activities defined on the nodes of the graph. It primarily encodes (a) the information about the window-based aggregate values for an active attribute, and (b) whether a node is currently active w.r.t. to the active attribute. This information is stored in an index outside the node object to simplify bookkeeping. Note that, depending on the monitoring and other pruning strategies we would selectively have this information for a set of nodes.

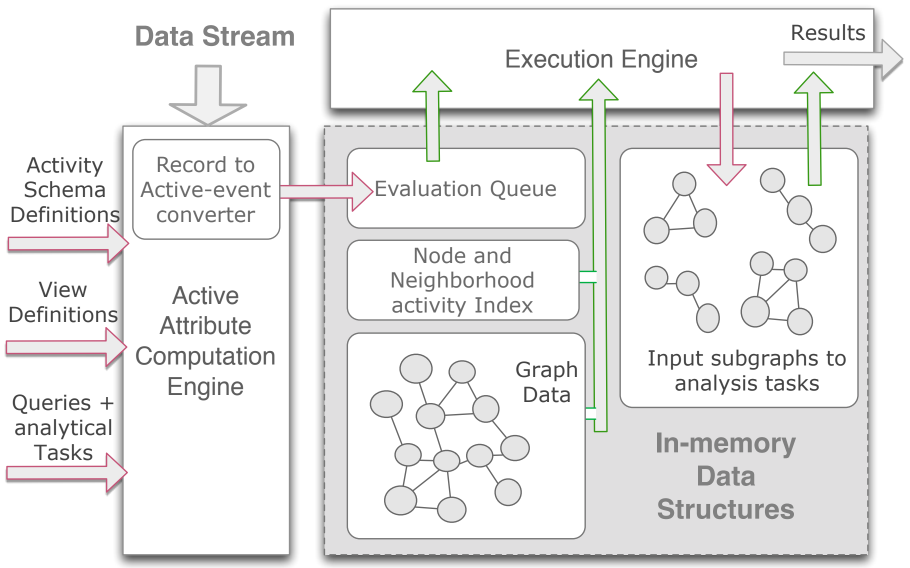

Data Stream
Execution Engine
Results
Activity Schema Definitions
Record to Active-event converter
Evaluation Queue
Node and Neighborhood activity Index
View Definitions
Active Attribute Computation Engine
Queries + analytical Tasks
Graph Data
Input subgraphs to analysis tasks
In-memory Data Structures

Figure 5.5: System architecture of CASQD

The index supports concurrent addition, removal, and updates.

**Neighbor Activity Index:** The neighborhood index contains the activity information of the neighborhoods of the node. Specifically, it maintains how many neighbors are active for a node. Again depending on monitoring strategy we store and maintain the information for a selective set of nodes. The index also supports concurrent addition, removal, and updates.

### 5.4.2 Controllers

**Record to Activity Mapper:** The record to activity mapper is responsible for generating active attribute events from the tuples in the activity stream. The activity mapper uses the knowledge of the *activity schema* as well as the definitions of each of the *active attributes* to generate the activity events. The events are then passed to corresponding node for processing.

**Activity Event Handler:** On receiving an activity event, a node makes a call to the activity handler function. The activity handler consults with the in-memory data structures to decide the course of action. The three high level functions performed by the handler are (a) updating the value of the corresponding active attribute, (b) letting the neighbors know about any change of its active attribute values, and finally (c) invoking an appropriate function to explore the neighborhood of the node. However, depending on the precise monitoring strategy some of the steps might be omitted of augmented with additional task.

**Task Scheduler:** Task scheduler is responsible for scheduling the analytical tasks on the extracted subgraphs. Such tasks are handled in two different ways depending on the *mode of execution*. In case of one-time mode, the analytical task is performed on a copy of the extracted subgraph pattern. If new information is needed to from the rest of the graph, they are copied as well. Similar strategy is also used for activity aware-mode, but the copy is updated continuously with a repeat frequency provided as external parameter.

## 5.5 Evaluation

### 5.5.1 Experimental Setup

We ran our experiments on a 2.2GHz, 24-core Intel Xeon server with 64GB of memory, running 64-bit Linux. Our prototype system is implemented in Java.

**Dataset:** We evaluated our approach on several real-world information networks but here we report results primarily for LiveJournal dataset (*soc-LiveJournal1*: 4.8M nodes/69M edges)[2]. Additionally we generate a set of graphs using preferential model [50] to experiment how our techniques perform for graphs of different average degrees. Apart from the graph structure another important component of the dataset in our case is the activity patterns of the nodes. However, due to unavailability of a suitable activity dataset, we use simulated dataset for this purpose. Such a simulated dataset may not precisely portray any real-world scenario, but they allow efficient micro-benchmarking of the system, due to their parameterized nature. For our simulated activity pattern we used a strategy motivated by some of the earlier work [74, 112]. Similar to the strategy used in [112], we chose 100 Twitter users with sufficient number of tweets and downloaded their timeline. From the access traces we created histograms to represent the daily expected behavior of the nodes. The granularity (i.e., the lowest bucket width) of our histograms is 5 minutes. Once we had the pool of histograms, we assigned them to the nodes in the network. Motivated by the above mentioned work we used the following assignment process. We assign the histogram using a serial ordering. When assigning a histogram to a node we find the majority histogram in the neighborhood of the node, and assign that to the node under consideration with 50% probability. Otherwise we choose any one of the remaining histograms with equal probability. This, in effect, simulates the fact that connected users are more likely to belong to the same timezone and would have correlated histograms. One important point is that we do not use these assigned histograms directly. For each user, we instead randomized the assigned histogram by generating a trace by treating the histogram as a probability distribution, and then building a histogram on the generated trace. This ensures sufficient diversity in the user histograms across the network.

[2]Available at `http://snap.stanford.edu/data/index.html`.

**Evaluation Metric:** Our main aim is to study the following two aspects:

- Behavior of the different execution strategies (discussed in Section 5.2) under different application scenarios.
- Behavior of different active primitives across all execution strategies.

We capture the application scenario using the following two parameters:

**Size of the active patterns:** We anticipate that the size and the number of the active patterns would vary significantly across applications. Some applications might require detection of many active patterns (e.g., social advertisements), while other applications might be looking for active patterns that are relatively rare. Since there is no direct way to vary the number of expected answers for studying the behavior of the execution strategies, we chose size of the patterns as a proxy. Ideally, keeping all the parameters same, smaller patterns are expected to result in more matches.

**Activity threshold:** Activity threshold is a parameter used in the definition of active attributes. Activity threshold indicates the activity level a node needs to reach, before it would be considered active. Such threshold would significantly influence the number of active nodes, the number of pattern matches, the amount of memory requirements, and ultimately the choice of the execution strategy.

As we vary the mentioned parameters to simulate various application scenarios, following are the main metrics we compare.

**Number of subgraph explorations:** Here we count the number of pivot explorations initiated by the execution strategies. This is a good abstract metric for comparing performance of the strategies.

**Number of aggregate computations:** Here we count the number of activity-events that need to be ingested for the computation of the active attributes. Note that an eager strategy to monitor the activity of the nodes would need to ingest every event, while a lazy strategy might let go some of the events, and thereby saving computations.

**Number of neighborhood updates:** This measure counts the number of times a newly active or a newly inactive node communicates its status change to its neighbors. Different execution strategies adopt different mechanisms for such updates.

The other metrics that we consider are **end-to-end throughput the system**, **memory utilization**, and the **scale up** performance.

### 5.5.2 Execution algorithms

First, we compare the number of neighborhood explorations performed by different execution strategies. For all these experiments, we keep all the other system parameters same apart from the one that we explicitly mention.

#### 5.5.2.1 Varying activity threshold

We start by varying the activity threshold (i.e., the activity level at which a node becomes active) of the nodes. We keep the pattern size fixed at 10 (i.e., COUNT aggregate on node events, window width = 3000 second, slide of 500 second). For biclique we look for 10x10 bipartite clique. Figures 5.6(a)-(c) shows the result.

**Number of explorations:** The first observation is that as the activity threshold increases, the number of explorations first increases and then decreases. This is consistent

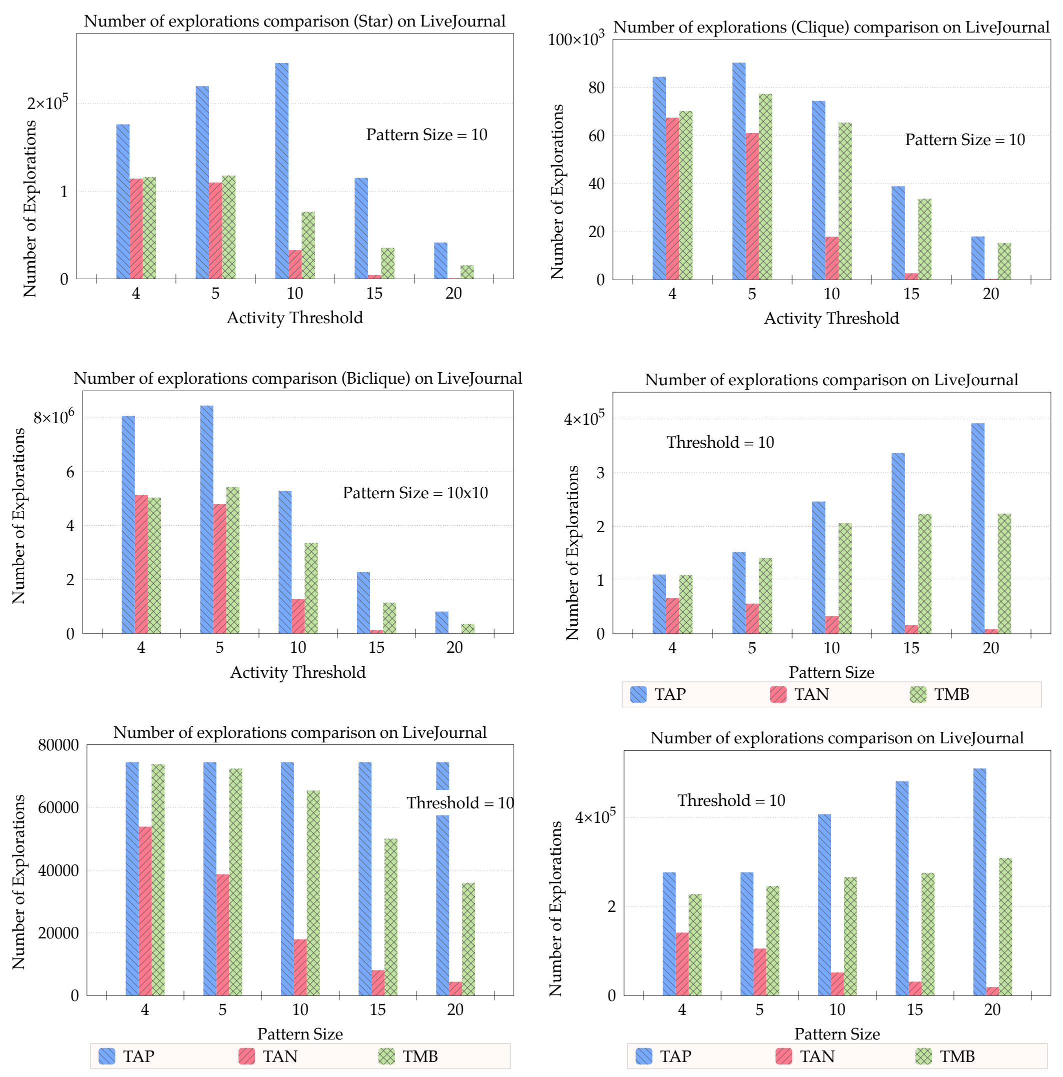


Figure 5.6: (a)-(c) Comparing the number of neighborhood explorations by Tap, Tan, and Tmb for Star, Clique, and Biclique while varying activity threshold. (d)-(f) Doing the same comparisons while varying the pattern size.

for all the execution strategies and for all the active primitives. The reason is that we perform explorations only when a node becomes newly active (i.e., according to our answer model, we only report new pattern matches). When the activity threshold is low, many nodes remain active, and hence many patterns also remain active, thereby causing a decrease in the number of explorations that need to be performed. Note that, the threshold value till which the number of the explorations will increase would completely depend on the workload. Moreover, for different types of primitives the peak could be at different thresholds, as there are no simple relations among the number of explorations in *star*, *clique*, and *biclique* queries. The primary reason behind the difference is the trigger function for these primitives (discussed in Section 5.2.1).

The second observation is that TAP (i.e., the neighborhood-agnostic approach) always perform more explorations than the neighborhood-aware approaches TAN and TMB. However, the difference is much more when the activity threshold is high. This is because, with a low threshold, more nodes are active and which in turn means more neighbors of a node would be active. Hence, the neighborhood-aware approaches satisfy the neighborhood-based cut-off (i.e., the minimum number of active neighbors required to find a match) more often. However, as the activity threshold increases, crossing the neighborhood cut-off becomes less probable, making the neighborhood aware approach more efficient.

Another important observation is that, even considering a fixed activity threshold, the differences in the number of explorations between the neighborhood-agnostic and neighborhood-aware approaches are different for different primitives. The difference is much higher for *Star* and *Biclique* than *Clique* queries. The reason is explained in the discussion about triggers in Section 5.2.1. The trigger step, which investigates whether any neighbor of the currently explored node needs to be investigated, is not necessary for *Clique* queries. For this reason the explorations performed by the TAP for *Clique* query is relatively much less than the other two primitives.

We can see that the model-based approach TMB does more explorations than the eager neighborhood-aware approach TAN. TAN maintains perfect information about the number of active neighbors, and hence is *optimal* in terms of the number of explorations required. Whereas TMB may do an exploration even when the concerned node is inactive, but the model thinks it is active. However, it is important to keep in mind the cost of the model-based explorations might be different. The model-based exploration first needs to compute the node's actual activity, and would quit the exploration if a node is found to be not active.

By comparing Figure 5.6 (a), (b), and (c), we see that, even for the same trace, *biclique* query leads to more explorations than *star* query, which in turn does more explorations than *clique* query. This is again due to the nuances in their trigger functions. Both *star* and *biclique* may initiate another level of explorations, while *clique* doesn't cause such explorations. Moreover, as discussed in Figure 5.4, *biclique* queries might trigger a node 2-hop away from the pivot node, while for *star queries* the trigger search space limited to nodes in the 1-hop neighborhood.

**Number of neighborhood updates:** Figure 5.7(a) shows the number of times a node needs to inform about its status changes to its neighbors. The neighborhood-agnostic approach TAP doesn't take part in any such activities. Comparing between TAN and TMB, TAN performs 1.25x-3.4x more status updates for neighbors. However, the difference is much lower for low activity threshold. TAN takes an eager approach to neighborhood status update, and informs all its neighbors every time its status changes. Whereas for TMB, at the low threshold, relatively more nodes will be active based on the histogram model, thus those explorations do not require any such status updates. However, as discussed in Section 5.2.3, TMB needs to inform its neighbor when a node becomes active in a contradiction to the model-based prediction.

**Number of aggregate computations:** In Figure 5.7(b), we show the number of tuples that were aggregated for the computation of active attributes. For TAP and TAN the total number of such aggregations don't depend on the threshold, and hence they are same across all threshold. However, for TMB as the activity threshold increases, for increasingly higher number of nodes, their active attributes are eagerly maintained. While at lower threshold, the attributes are lazily computed, thereby doing on an average 1.5x-2.75x less aggregate computations. Here we only show results for the *biclique* query; the results for the other two types show similar trends.

#### 5.5.2.2 Varying pattern size

In Figure 5.6(d)-(f), we do a similar comparison as above, but here we vary the size of the patterns. We fix the activity threshold at 10 events in a window.

**Number of explorations:** As before, the trigger step of the execution algorithm plays an important role. As pattern size increases, neighborhood-aware strategy TAN shows the best performance. As discussed before, since TAN maintains perfect information about the neighborhood, as the pattern size increases, it is able to prune out more explorations. On the other hand TAP and TMB suffer from a significant number of failed explorations. This means that many nodes are waiting to be triggered (for the case of *star* and *biclique*) when one of their neighbors is explored. This leads to a higher number of explorations. However, since *clique* queries don't require such triggers, in that situation, number of explorations decreases as pattern size increases for TMB. The number of explorations done by TAP remains the same for all pattern sizes as it is neighborhood-

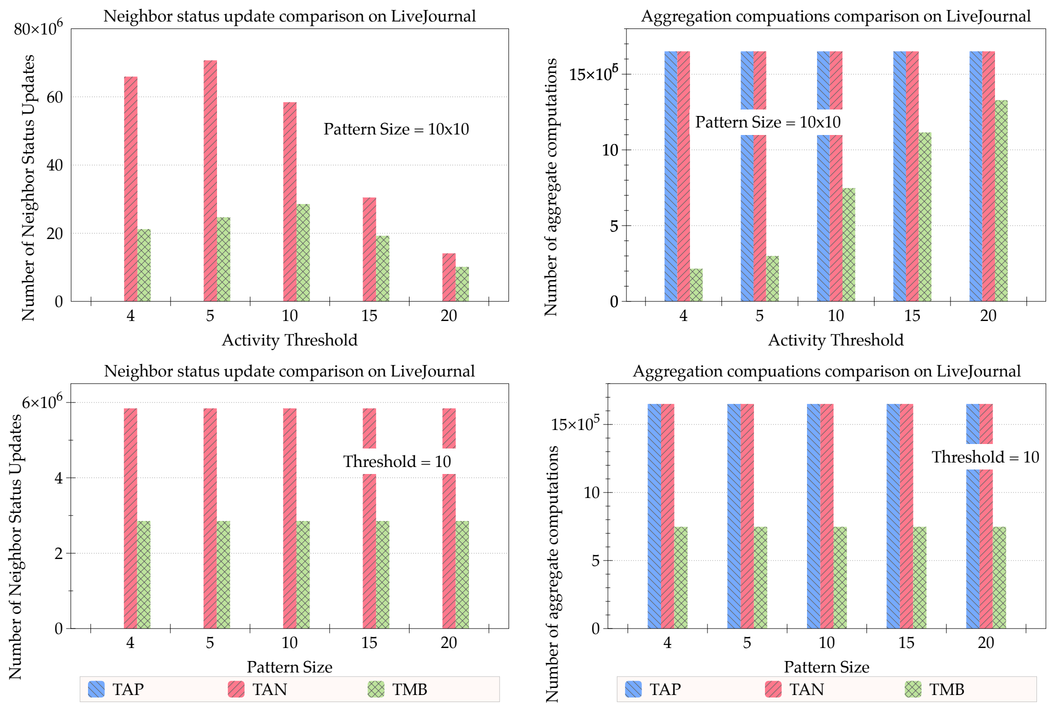


Figure 5.7: Comparing the (a) the number of neighborhood status update, (b) number of aggregates computation while varying activity threshold. Comparing the (c) the number of neighborhood status update, (d) number of aggregates computation while varying pattern size.

agnostic.

**Number of neighborhood updates:** Figure 5.7(c) shows that the number of neighborhood status updates remains the same for both TAN and TMB as the pattern size changes. The number of neighborhood status updates is only dependent on the activity threshold for both the cases.

**Number of aggregation computations:** As above, Figure 5.7(d) shows that the number of aggregation computations is also dependent on the activity threshold and not on the pattern size.

## 5.5.3 Other Experiments

**A scenario of low selectivity and large pattern size:** We believe that many applications would fit a scenario where many nodes are active and change status often, and the sizes of patterns are relatively large. So, we further study this scenario carefully. Figure 5.8, shows the result for different primitives. For all the primitives, TAN performs the lowest number of explorations. However, TAN has very high neighborhood status update cost compared to TMB. Moreover, TMB also performs fewer numbers of aggregation computations.

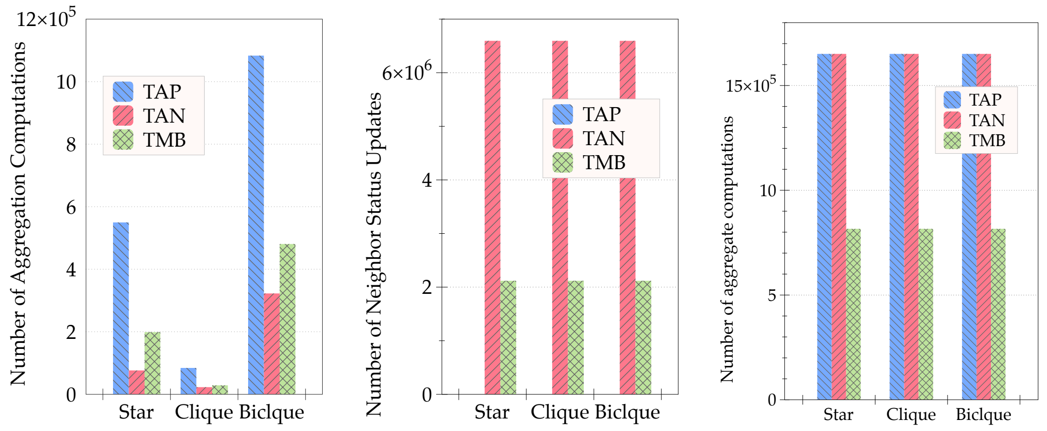


Figure 5.8: Comparing the execution strategies for attribute with low selectivity (threshold = 4) and high pattern size (= 20).

**Throughput:** We use the same workload trace to compute end-to-end throughput, i.e., how many trace events are ingested per second. For these experiments, we treat the *clock time* in the workload trace as logical time steps. For example, if the trace contains two consecutive events one at the 0th second of the day and the next at the 300th second of the day (i.e., after 5 minutes), we don't wait for 5 minutes after handling the 1st event. We

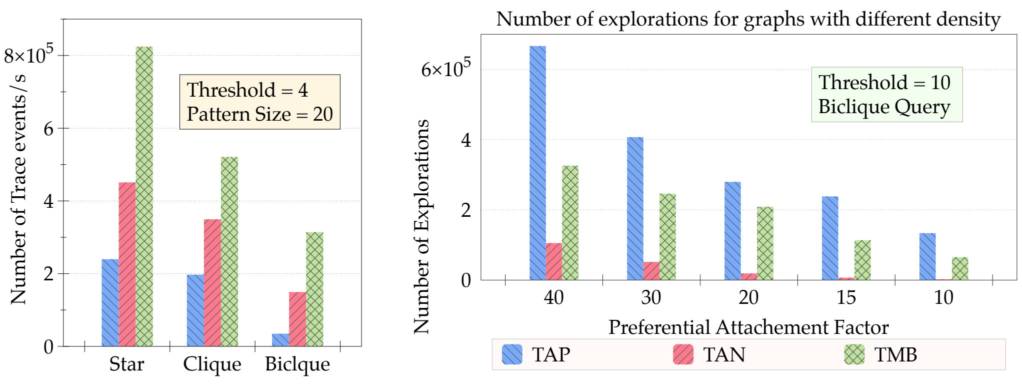


Figure 5.9: (a) Throughput comparison. (b) How graph density affects the number of explorations.

immediately process the next event, but we also compute all the window-based aggregate following that logical time stamp. For example, if we had an active attribute defined using a 300-second slide, we would slide that window when the second event occurred (assuming that the last slide was at 0th second). Figure 5.9(a) show that even in a low selectivity scenario (i.e.,low threshold, and hence more active nodes. Also, the same setting as the experiment described in Figure 5.8), our system is able to reach a throughput of 800K events per second.

**Varying Graph average degree:** In the experimental result shown in Figure 5.9(b), we show the effect of the average degree of a graph on the number of explorations. This result is for biclique queries with threshold fixed at 10, and pattern size 10. We can see that denser graph results in more explorations.

**Effect of structural pruning:** We also compared our system with and without additional structure-based pruning. For structural pruning, the simple strategy is, if a node has less than the number of required neighbors (i.e., less than *n* neighbors for a *n-clique* query), we can mark them not to explore further, unless the node gets more neighbors.

However, our result shows that this simple pruning has not much effect on the overall performance. The reason is that marking a node for not exploration and checking whether the node has a sufficient number of neighbors are computationally comparable. Another, more complex type of structural pruning could be to enumerate whether a node has a structural *n-clique* or *n*×*m-biclique*, but this would computationally be too expensive when a node a more than *n* neighbors.

# Chapter 6

# Conclusion

Modeling data as graphs and analyzing them is increasingly becoming ubiquitous for extracting unique insights that are embedded in the complex connectivity of the data items. In many application domains, the graphs are large and dynamic, and require fast processing of events to generate insights in real or near-real time. In this dissertation, I proposed novel techniques and built systems to support various types of queries on such large and dynamic graph-structured data. Firstly, for management of large graphs and to support low latency querying, I built an in-memory distributed graph database system. A core component of the system is a replication manager that monitors the read and write behaviors of the nodes, and uses an adaptive and hybrid replication strategy that switches between eager and lazy replication modes, in order to minimize data movement cost across machines, and to minimize query latency. I experimentally verified that our techniques allow us to save more than 33% network bandwidth, while maintaining low latency of query execution. I also introduce a novel *fairness criteria*, a single parameter that could be used to trade off data movement cost and query latency. Secondly, I built an in-memory, general-purpose, and user-extensible framework to support ego-centric aggregate queries. I proposed a pre-compiled overlay-based approach, suitable for a continuous query system. The overlay mimics a dataflow-style execution, and allows new events to be ingested and aggregated without any overhead, as it seamlessly encodes sharing of partial aggregates, as well as selective pre-computation of aggregates. I proposed scalable algorithms to build compact overlays, and efficient techniques to make dataflow decisions. I show through experiments that our aggregation framework can handle graphs of size more than 320 million of nodes and edges and can achieve a query throughput of 500K/s. Finally, I built a system, CASQD, that supports efficient detection and analysis of active subgraph patterns. I introduced novel abstractions like *graph-view* and *activity schema*, and designed a system pivoted around efficient detection of regular active primitives. Our thorough experimental evaluation shows that our techniques are scalable for various application scenarios, and are able to achieve an end-to-end throughput of more than 800K events/s using a single server-grade machine.

The work presented in this dissertation will benefit systems research. Firstly, organizations that need to handle large amount of graph-structured data and produce insights in real-time, would benefit from the optimization techniques that we have developed. Our optimization techniques are algorithmic in nature and are independent of data storage engine, operating system, programming language, development platform, etc. Although, some of our techniques are workload-aware, and their performance will depend on the specific nature of the workload, most of our techniques provide simple parameters to control relevant tradeoffs. Apart from the system architects, we envision our system and techniques to be of much use to data analysts. In recent times, we have seen a fast adoption of graph-based data analysis. Analysts are using log-structured data to build graphs by semantically relating data items in the logs, so that they can benefit form graph-based analytics. However, an important problem there is to pick the right entities so the resultant graph would be semantically coherent, and the analytical results will be consumable. It could be really cumbersome to manually process the data to build new graphs. Our system would be really useful in such a scenario. We provide a declarative interface to build a graph by connecting the entities of choice, given an input stream (or log) of events. Analysts can use this interface to build the graph of their choice, and analyze the results, without worrying about the actual implementation of a graph data management system.

In addition to system research, the systematic abstractions that I propose can be helpful in guiding data analysts for engineering activity-based graph features. For example, a simple step-by-step approach to feature engineering could be to, first, build a graph by connecting related data items, and then try to reason about the semantics of different graph structures (e.g., stars, cliques, bicliques) and various interaction patterns on these graph structures. In fact, we have adopted this approach in a recent work [98] where we engineer graph-based features for nodes in a software download graph, and many of the graph-based features had significant influence in distinguishing malicious software from benign ones.

Many of the techniques that I have developed in this dissertation have much broader applicability than real-time analytics on graph data. Our technique of adaptive and hybrid replication of data movement by fine-grained monitoring of read/write patterns of nodes, is applicable to any distributed data management system where data production and access follows a repetitive pattern. The algorithms that we develop for building compact overlay would be of interest to graph compression and data mining communities.

## 6.1 Insights

In this section, we share some of the key insights about real-time analytics on dynamic graphs, learnt during the course of this work.

- While the unstructured and fine-grained nature of the graph data poses a lot of challenges in building efficient and general-purpose systems, smart use of the structural information embedded in the graph structure may lead to efficient system design. Similarly, even though graph dynamicity is a threat to stable system performance, patterns in the dynamic behaviors of the graph entities can be exploited to aid scalable system design.
- Many optimization problems on graph-structured data are hard problems. Considering dynamic scenarios complicates the situation further. Designing a stable, efficient, and robust system that solves the joint optimization problems could be really difficult. In our experience, in such situations, adopting a modular design by decoupling the optimization problems may be more tractable. Moreover, such decoupling is especially helpful, when the individual components of the optimization problem need to be solved with different frequency.
- Implementing general-purpose query language on unstructured data like graph data could be extremely challenging. However, breaking down such a task into multiple components, and building special purpose techniques for the individual modules,

may lead to efficient task specification language, without sacrificing the expressibility too much.

## 6.2 Future Directions

In this section, we discuss some of the interesting and potentially impactful avenues that could be pursued to extend our work.

### 6.2.1 Context-aware Stream Processing

Stream processing systems are optimized for high throughput, and are very efficient at detecting important events by sifting through a large amount of information. However, from the analysts' point of view, often an investigation is not complete until the corresponding event is seen in an appropriate context. *Context* of an event could be seen as additional qualifying information that puts an event in perspective, thereby explaining the event and making it consumable and actionable. At an abstract level, such context-driven analysis requires us to execute an *explanation algorithm* on some accessible form of *context information*. Both the explanation task and the context information, however, depending on the application, could come in various flavors. The simplest and most general form of context information, perhaps, could be the stored history of the nodes that were a part of a detected event. On the other hand, the explanation algorithm might be any arbitrary analysis task on such history.

**Example 7.** ***Influential User:*** *In a social network, influential users are often defined to be the ones that induce a lot of activities among their followers. A single instance of* *such a behavior could be extracted by identifying an active star (i.e., all the nodes are active within a small period of time), especially when the peripheral nodes are not well connected. This, however, can't guarantee that the central node initiated the activities of the peripheral node. But, if the same active star repeats over time (i.e., the central node remains the same, and the peripheral nodes are very similar), then we can say with higher confidence that the central node is responsible for the activities. In connection with the ongoing discussion about context-aware stream processing, in this example, the context information could be a list of all the active stars that have been detected by the system. While the explanation algorithm figures out, for a given center of an active star, whether the central node has participated in many active stars over time, thereby deeming the node as an influential user.*

**Example 8.** ***Synchronized behavior:*** *Synchronized behavior by nodes in a network has been shown to be indicative of anomalous behavior. In a recent work [87], Jiang et al. have shown that new Yelp businesses often buy reviews from users, and the unethical users typically respond within a time period (i.e., they are active within a window) by writing reviews for the concerned business. Moreover, the same set of users tend to write reviews for multiple businesses over time. This again could be seen as an example of active star, where only the set of peripheral nodes (the unethical users) repeats over time in groups and the central node changes (i.e., the new businesses). In this example, the explanation task needs to look for historical evidence for repetitive grouped review writing by a set of users.*

So far, we have discussed some examples of explanation algorithms, however, with a rather simple form of context information, i.e., history-based context which is static in nature. A more complex situation arises when we allow such contexts to be dynamic in nature. For example, detection of the interesting events could be relative to other events detected in the system. An event with low selectivity (i.e., many such events are simultaneously detected) might be of reduced interest, and an application might require them to be filtered out.

There have been some work on combining stream processing in the context of historical data [56, 54, 127]. However, none of that work supports graph-structured data and graph-based event detection. There has been several recent efforts, generally termed as Lambda Architecture [20], that also try to combine stream processing and batch processing, and produces unified results. Even though, at a high level, Lambda Architecture is similar to context-aware stream processing, they are fundamentally different. Lambda architecture is designed to perform both streaming and batch processing on the same data, so that final results could be produced by combining output from both the systems. However, in our case, we need a context server whom our stream processing system can consult on demand.

#### 6.2.1.1 Supporting Composition of the Active Primitives

In this dissertation, I have discussed the benefits of designing active pattern matching techniques around efficient detection of active primitives. A natural way to extend the expressive power of the language would be by supporting composition of the active primitives. However, such composition needs to be dealt with care. Considering the fact that an edge could be thought of as a *1-star*, unrestricted composition would allow users to specify arbitrarily complex active subgraphs. As discussed before, optimizing for such composition-based complex patterns could be difficult in a dynamic setting. However, using a restrictive form of composition, motivated by real-life applications, could be useful, where the system can increase query expressibility without the query optimizer having to lose much control.

#### 6.2.1.2 Approximate Identification for Active Primitives

In our CASQD system, we focused on accurate evaluation of active subgraph patterns, and didn't explore the issue of approximate answers. For example, considering the example of active cliques, CASQD will report all active cliques that a node *n* has participated in. However, certain applications might not require such exact results. For example, an application might prefer us to report only one result if there are two active n-cliques which differ by only a single node. In such a scenario, which is a very practical one, there is an opportunity to apply sophisticated pruning algorithms.

#### 6.2.1.3 Ego-centric Aggrgeation Framework

EAGr is designed for quasi-static scopes, and the techniques can't be directly used for dynamic scopes. An important research question here is how to represent extracted dynamic scopes, especially if there exists significant overlap among the identified scopes.

EAGr's overlay building algorithm mainly considers the structure of the graph, even though it performs local modifications to overlays based on the read/write activity patterns of the nodes. An interesting approach could be to build the overlay in an activity aware fashion, i.e., while building the overlay the active edges are given higher priority than the inactive ones.

Moreover, EAGr uses a limited aggregation model. It assumes all neighbors of a node are of equal importance. However, in many real-world applications this assumption will not be true. For example, in a social network there could be *close friends* vs *acquaintances*. The problem of finding sharing structures under such weighted aggregation models can lead to interesting new challenges. Furthermore, EAGr is evaluated by executing a single type of aggregate (i.e., either MIN, MAX, or TOP-K) query at one time, and it would be interesting to study the execution of multiple types aggregate queries simultaneously.

While EAGr focuses on the scenario where there is one continuous query per node, other models are also possible. For example, Facebook might be interested in finding the trending topics among people who "talk" (e.g., topic attributes for user nodes) about Music, Sports, Politics, Cats, etc. It would be fruitful to find overlaps in these groups to reduce computation overheads. Another important challenge here is that the topic attributes on the users can change dynamically over time, depending on what they post, requiring the overlaps to be maintained dynamically.

### 6.2.2 Distributed Management of Graph Data

While adaptive replication attempts to solve a set of core issues in distributing large content-dynamic graphs, there are a lot of interesting avenues to pursue in the future.

Firstly, the framework needs to be extended for queries with *pivoted* but dynamic scopes (e.g., *search* and *pattern matching*), where the precise query scope is not known beforehand. However, these queries typically have well-behaved query scope (i.e., the search space in most cases is not arbitrarily complex, given a seed node). We plan to make use of the historical information to make an estimate about the scope as well as the query pattern (similar to COSI [46]), to make the replication decisions. The problem of k-way balanced partitioning needs to be revisited, especially in the context of balancing storage and processing load on a system. Such a partitioning scheme needs to be aware of the content generation as well as query patterns of the nodes, along with the behavior of the query scope. Currently, our system assumes that a user is interested in newsfeed from all its friends. However, it might not be the case depending on the application. Many social networks have the notion of *upgraded edges*, e.g., *close friends* on Facebook. Such upgraded edges in the context of Facebook mean that a user should always see updates from his *close friends*, while, for the normal friends the system can decide about whether to show their updates, based on certain optimization criteria. The adaptive replication algorithm that we developed can have interesting implications on such application-imposed constraints and might be worth investigating.

# Bibliography

[1] Apache Flink. `https://flink.apache.org/`.

[2] Apache S4. `http://incubator.apache.org/s4/`.

[3] Blueprints. `https://github.com/tinkerpop/blueprints`.

[4] CouchDB. `http://couchdb.apache.org/`.

[5] Cypher. `http://ineo4j.com/docs/stable/cypher-query-lang.html`.

[6] FlockDB. `https://github.com/twitter/flockdb`.

[7] Furnace. `http://github.com/tinkerpop/furnace/wiki`.

[8] Giraph. `http://incubator.apache.org/giraph`.

[9] GraphBase. `http://www.graphbase.net/`.

[10] Gremlin. `https://github.com/tinkerpop/gremlin`.

[11] HyperGraphDB. `http://www.kobrix.com/hgdb.jsp`.

[12] InfiniteGraph. `http://www.infinitegraph.com/`.

[13] Kinesis. `https://aws.amazon.com/kinesis/`.

[14] Neo4j. `http://neo4j.org/`.

[15] NetworkX. `https://networkx.github.io/`.

[16] OrientDB. `http://orientdb.com/orientdb/`.

[17] Pulsar. `https://github.com/pulsarIO/realtime-analytics/wiki`.

[18] SNAP. `http://snap.stanford.edu/`.

[19] Titan. `http://thinkaurelius.github.io/titan/`.

[20] `http://lambda-architecture.net/`.

[21] `https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model`.

[22] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A data stream management system. In *SIGMOD*, 2003.

[23] Charu C. Aggarwal, editor. *Data Streams: Models and Algorithms*. Springer, 2007.

[24] Charu C. Aggarwal and Haixun Wang, editors. *Managing and mining graph data*. Springer, 2010.

[25] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: fault-tolerant stream processing at internet scale. *VLDB*, 2013.

[26] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *VLDB*, 2015.

[27] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. Oddball: Spotting anomalies in weighted graphs. In *KDD*. 2010.

[28] Lory Al Moakar. *Class-Based Continuous Query Scheduling in Data Stream Management Systems*. PhD thesis, Univ. of Pittsburgh, 2013.

[29] Luıs A Nunes Amaral, Antonio Scala, Marc Barthelemy, and H Eugene Stanley. Classes of small-world networks. *Proceedings of the national academy of sciences*, 2000.

[30] Albert Angel, Nikos Sarkas, Nick Koudas, and Divesh Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB*, 2012.

[31] Renzo Angles. A comparison of current graph database models. In *Data Engineering Workshops (ICDEW)*, 2012.

[32] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, 2011.

[33] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDBJ*, 2006.

[34] Arvind Arasu, Christopher Re, and Dan Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.

[35] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.

[36] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load Shedding for Aggregation Queries over Data Streams. In *ICDE*, 2004.

[37] Albert Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 1999.

[38] D.F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *WWW*, 2009.

[39] Fabrcio Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virglio A. F. Almeida. Characterizing user behavior in online social networks. In *Internet Measurement Conference*, 2009.

[40] Haixun Wang Bin Shao and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[41] Jose A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, 1986.

[42] Vincenet D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics-theory and Experiment*, 2008.

[43] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL: extending SPARQL to process data streams. *The Semantic Web: Research and Applications*, 2008.

[44] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

[45] Matthias Brocheler, Andrea Pugliese, and V. S. Subrahmanian. COSI: Cloud oriented subgraph identification in massive social networks. In *ASONAM*, 2010.

[46] Matthias Brocheler, Andrea Pugliese, and VS Subrahmanian. COSI: Cloud oriented subgraph identification in massive social networks. In *ASONAM*, 2010.

[47] Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[48] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *WSDM*, 2008.

[49] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *CloudDB*, 2012.

[50] A. Capocci, V. D. P. Servedio, F. Colaiori, L. S. Buriol, D. Donato, S. Leonardi, and G. Caldarelli. Preferential attachment in the growth of social networks: The internet encyclopedia wikipedia. *Phys. Rev. E*, 2006.

[51] Ümit Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*. Springer, 2011.

[52] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: laws, tools, and case studies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 2012.

[53] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: a high-performance incremental query processor for diverse analytics. *VLDB*, 2014.

[54] Badrish Chandramouli, Jonathan Goldstein, and Songyun Duan. Temporal analytics on big data for web advertising. In *ICDE*, 2012.

[55] Badrish Chandramouli, Justin J Levandoski, Ahmed Eldawy, Mohamed F Mokbel, Big Data, and Justin Levandoski. StreamRec: a real-time recommender system. In *SIGMOD*, 2011.

[56] Sirish Chandrasekaran and Michael Franklin. Remembrance of streams past: overload-sensitive management of archived streams. In *VLDB*, 2004.

[57] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu,

Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EUROSYS*, 2012.

[58] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, 2009.

[59] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *EDBT*, 2015.

[60] Edith Cohen, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D. Ullman, and Cheng Yang. Finding Interesting Associations without Support Pruning. In *ICDE*, 2000.

[61] O. Cooper, A. Edakkunni, M. Franklin, W. Hong, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, and E. Wu. Hifi: A unified architecture for high fan-in systems. In *VLDB Demonstration*, 2004.

[62] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 2010.

[63] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A system for searching the social graph. *VLDB*, 2013.

[64] Jing Fan, Adalbert Gerald, Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[65] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *TODS*, 2013.

[66] Tomas Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. In *STOC*, 1991.

[67] Huiji Gao, Jiliang Tang, and Huan Liu. Exploring social-historical ties on location-based social networks. In *ICWSM*, 2012.

[68] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *ICDE*, 2014.

[69] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc., 2007.

[70] George F. Georgakopoulos and Kostas Politopoulos. Max-density revisited: a generalization and a more efficient algorithm. *The Computer Journal*, 2007.

[71] Scott A. Golder, Dennis M. Wilkinson, and Bernardo A. Huberman. Rhythms of social interaction: messaging within a massive online network. *CoRR*, abs/cs/0611137, 2006.

[72] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[73] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J

Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[74] Roberto Gonzalez, Ruben Cuevas Rumin, Angel Cuevas, and Carmen Guerrero. Where are my followers? understanding the locality effect in twitter. *CoRR*, abs/1105.3682, 2011.

[75] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[76] Ashish Gupta and Iderpal Singh Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[77] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. SASE: complex event processing over streams. In *CIDR*, 2007.

[78] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[79] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. In *VLDB*, 2014.

[80] Huahai He and Ambuj K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.

[81] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng,

Kun Li, and Arun Kumar. The MADlib analytics library: or MAD skills, the SQL. *VLDB*, 2012.

[82] Keith Henderson, Brian Gallagher, Tina Eliassi-Rad, Hanghang Tong, Sugato Basu, Leman Akoglu, Danai Koutra, Christos Faloutsos, and Lei Li. Rolx: structural role extraction & mining in large graphs. In *KDD*, 2012.

[83] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 1995.

[84] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *SIGARCH*, 2012.

[85] Ryan Huebsch, Minos Garofalakis, Joseph M Hellerstein, and Ion Stoica. Sharing aggregate computation for distributed queries. In *SIGMOD*, 2007.

[86] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, 2000.

[87] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. Catchsync: catching synchronized behavior in large directed graphs. In *KDD*, 2014.

[88] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: your relational friend for graph analytics! In *VLDB*, 2014.

[89] Sudarshan Kadambi, Jianjun Chen, Brian Cooper, David Lomax, Raghu Ramakrishnan, Adam Silberstein, Erwin Tam, and Hector Garcia Molina. Where in the world is my data? In *VLDB*, 2011.

[90] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, 2009.

[91] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 1998.

[92] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, 2013.

[93] Samir Khuller and Barna Saha. On finding dense subgraphs. In *ICALP*, 2009.

[94] S. Koening and R. Paige. A transformation framework for the automatic control of derived data. In *VLDB*, 1981.

[95] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[96] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[97] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, 2015.

[98] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitras. The dropper effect: Insights into malware distribution with downloader graph analytics. In *CCS*, 2015.

[99] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *VLDB*, 2012.

[100] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, March 2007.

[101] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *World Wide Web Conference Series*, 2008.

[102] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Journal of Internet Mathematics*, 2009.

[103] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. Querying Graph Databases with XPath. In *ICDT*, 2013.

[104] Xiaowen Liu, Jinyan Li, and Lusheng Wang. Quasi-bicliques: Complexity and binding pairs. In *Computing and Combinatorics*. Springer, 2008.

[105] Boon Loo, Tyson Condie, Minos Garofalakis, David Gay, Joseph Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[106] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[107] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS*, 2002.

[108] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[109] Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David E. Culler. Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks. In *WMSCA*, 2002.

[110] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[111] Julian Mcauley and Jure Leskovec. Discovering social circles in ego networks. *TKDD*, 2014.

[112] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, 2012.

[113] Jayanta Mondal and Amol Deshpande. EAGr: Supporting Continuous Ego-centric Aggregate Queries over Large Dynamic Graphs. *SIGMOD*, 2014.

[114] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku,

C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[115] Walaa Eldin Moustafa, Galileo Namata, Amol Deshpande, and Lise Getoor. Declarative analysis of noisy information networks. In *ICDE GDM Workshop*, 2011.

[116] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. High-performance complex event processing over xml streams. In *SIGMOD*, 2012.

[117] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 2005.

[118] Jeffrey Naughton, David DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayvel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, June 2001.

[119] M. E. J. Newman. Why social networks are different from other types of networks. *Physical Review E*, 2003.

[120] M. E. J. Newman. Modularity and community structure in networks. In *Proc. of The National Academy of Sciences*, 2006.

[121] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[122] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[123] J.M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine (s) that could: scaling online social networks. In *SIGCOMM*, 2010.

[124] Josep M. Pujol, Vijay Erramilli, and Pablo Rodriguez. Divide and conquer: Partitioning online social networks. *CoRR*, abs/0905.4918, 2009.

[125] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine (s) that could: scaling online social networks. *SIGCOMM*, 2011.

[126] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 1995.

[127] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M Hellerstein. Enabling real-time querying of live and historical stream data. In *SSDBM*, 2007.

[128] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *SSDBM*, 2013.

[129] Semih Salihoglu and Jennifer Widom. Help: High-level primitives for large-scale graph processing. In *GRADES*, 2014.

[130] Jiwon Seo, Stephen Guo, and Monica Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013.

[131] O. Shmueli and I. Itai. Maintenance of Views. In *SIGMOD*, 1984.

[132] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *SIGPLAN*, 2013.

[133] Adam Silberstein, Jeff Terrace, Brian F. Cooper, and Raghu Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, 2010.

[134] Adam Silberstein, Jeff Terrace, Brian F Cooper, and Raghu Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, 2010.

[135] Adam Silberstein and Jun Yang. Many-to-many aggregation for sensor networks. In *ICDE*, 2007.

[136] David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. Large-scale graph analytics in Aster 6: bringing context to big data discovery. In *VLDB*, 2014.

[137] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. Event pattern matching over graph streams. *VLDB*, 2014.

[138] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *VLDB*, 2012.

[139] Aubrey L Tatarowicz, Carlo Curino, Evan PC Jones, and Sam Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*, 2012.

[140] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, 2015.

[141] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "Think Like a Vertex" to "Think Like a Graph". *VLDB*, 2013.

[142] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *SIGMOD*, 2014.

[143] Niki Trigoni, Yong Yao, Alan J. Demers, Johannes Gehrke, and Rajmohan Rajaraman. Multi-query optimization for sensor networks. In *DCOSS*, 2005.

[144] Changliang Wang and Lei Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.

[145] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-Slice: New Paradigm of Multi-query Optimization of Window-based Stream Queries. In *VLDB*, 2006.

[146] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *TODS*, 1997.

[147] Ouri Wolfson and Amir Milo. The multicast policy and its relationship to replicated data placement. In *TODS*, 1991.

[148] Peter T Wood. Query languages for graph databases. *ACM SIGMOD Record*, 2012.

[149] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *SIGMOD*, 2012.

[150] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-pa- rallel computing: interfaces and implementations. In *SIGOPS*, 2009.

[151] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[152] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

[153] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *SIGMOD*, 2005.