# Subgraph-centric Large-Scale Graph Analytics on Spark

Albert Koy
Department of Computer Science
University of Maryland
College Park, MD 20742
akoy93@terpmail.umd.edu

## ABSTRACT

Over the past several years, large-scale graph analytics has become a quickly growing research area. Practical computing problems often deal with large graphs such as social networks and the Web graph, and as these graphs have grown to billions of vertices and trillions of edges, there has been an increased focus on executing graph computations in a distributed fashion. Vertex-centric distributed graph processing platforms like Pregel were some of the first systems meant specifically for distributed graph processing. Although these systems have vastly simplified the implementation of certain distributed graph processing algorithms, they are not well-suited for graph processing tasks that need to consider vertices in the context of their neighbors rather than in isolation. As such, one emerging paradigm is the use of a subgraph-centric view of graph computation in which users write programs at the level of subgraphs rather than at the level of vertices. This paper elucidates the fundamental differences between the vertex-centric and subgraph-centric paradigms, outlines an implementation of a subgraph-centric graph processing system called NScale on Apache Spark, a widely used distributed dataflow system, and introduces a novel interface for subgraph specification unseen in prior subgraph-centric graph processing frameworks.

## 1. INTRODUCTION

Efficiently processing large graphs calls for both distribution and parallelization, but these methods introduce many challenges. Graph algorithms tend to have their computations dictated by the structure of the underlying graph, making parallelization difficult, exhibit poor locality of memory access, have a high data access to computation ratio, and have a varying degree of parallelism during execution [3]. In addition, as graphs become distributed across many machines, the task of processing a large graph inherits many of the classic challenges in distributed systems: maximizing concurrency, load balancing, designing efficient patterns for communication between machines, fault tolerance, and scalability.

In the past several years, many systems have been developed in an attempt to provide scalable solutions to large-scale graph processing tasks. Pregel [4], published in 2010 by Google, is a *vertex-centric* graph processing framework for iteratively executing vertex programs across the vertices of a graph. It was one of the first distributed processing frameworks built specifically for handling large graphs. When Pregel was developed, there were very few options for users who wanted to process a large graph in any type of distributed setting. Users often had to resort to creating custom distributed infrastructure for different graph algorithms and different graph representations, rely on using distributed computing platforms like MapReduce which were poorly suited for graph processing, or use existing parallel graph systems, which at the time did not handle important issues like fault tolerance.

From Pregel, this research area of large-scale graph processing has seen quickly growing interest. In recent years, with systems like Giraph++ [15], NScale [7], and Arabesque [14], there has been a paradigm shift towards *subgraph-centric* implementations. Many of these implementations achieve significant performance gains over Pregel and other vertex-centric variants, however, there are still several challenges in contention between these subgraph-centric systems. For instance, providing a robust approach towards achieving fault tolerance is still a common issue with these systems with many of them opting for snapshot recovery instead. Additionally, these systems all use different approaches for allowing users to specify subgraphs on which to run computations. Giraph++ provides users no control and instead has the system automatically determine subgraphs, Arabesque provides a relatively sophisticated interface for selecting subgraphs, but the paper fails to mention support for large graphs that do not fit on a single machine, and NScale provides a subgraph specification interface that is only suitable for specifying a neighborhood around a vertex rather than subgraph patterns like triangles.

In this paper, we describe the implementation of a performant, scalable, and fault tolerant system for distributed graph processing, which we call NSpark. We hope to unify the major pieces of wisdom gained from prior implementations of graph processing systems. In particular, NSpark provides a subgraph-centric view of computation, provides a flexible programming interface that allows users finer control over specifying the exact subgraphs for the system to process, and is built on top of Apache Spark [19], a dis-

tributed dataflow framework that systems like GraphX [18] have leveraged to great success.

To summarize, the main contributions of this paper are as follows:

1. A survey of related work focused on motivating the purpose of subgraph-centric graph processing systems and uncovering some of the open research areas associated with these systems.

2. An outline of the design and implementation of a novel subgraph specification interface that allows users to easily express complex subgraph patterns.

3. An overview of the implementation and applications of NSpark, the successor of NScale, built on top of Apache Spark.

# 2. RELATED WORK

Pregel, one of the first specialized graph processing systems, [4] introduced the *"Think Like a Vertex"* paradigm in which users implement graph algorithms in a vertex-centric programming model. Each vertex in the graph contains only information about itself and its outgoing edges, and computations are expressed at the vertex level. As Pregel has shown, vertex-centric programming is a very natural and understandable model that has been used to create efficient and scalable implementations of algorithms like PageRank. However, vertex-centric frameworks are not always the most effective for many types of graph algorithms. Vertex-centric systems almost always operate on the entire graph, which is not necessary in many cases. Additionally, since computations in vertex-centric systems happen at the vertex level, these systems are not well-suited for graph algorithms that require information about the neighborhood around vertices due to the communication overhead and network traffic required to fetch this information.

Recognizing the deficiencies associated with many vertex-centric systems like Pregel, researchers have done a lot of work investigating potential optimizations to vertex-centric systems while still maintaining the same general computation model. For instance, researchers have explored potential optimizations to implementations of graph algorithms on Pregel-like systems to reduce communication overhead and the number of supersteps required to execute graph algorithms to completion [12].

On the other hand, systems like GraphX [18] attempt to improve upon the vertex-centric approach by altering the execution model and by leveraging existing distributed data processing frameworks. GraphX is built on top of Spark [19] and uses a Gather-Apply-Scatter decomposition as an execution model with pull-based messaging.

Another notable approach is the use of a single multicore machine, rather than a cluster of machines, for graph processing. The key insight with these systems is that, often times, distributed graph computation systems fail to outperform serial implementations on the same algorithm due to an inherent lack of parallelizability [6]. Ligra [13] is a system that is designed for graph processing on shared memory machines. Compared to distributed memory systems, communication costs are much cheaper and algorithm implementations are much simpler. X-Stream [9] is another shared-memory system that instead uses an edge-centric graph processing model using streamed partitions.

Beyond optimizations to vertex-centric systems and shared-memory systems, there have been a number of systems that have proposed subgraph-centric computation models. These types of systems primarily aim to reduce the message passing overhead and scalability issues associated with vertex-centric systems while still maintaining the benefits of having a highly distributed system. For instance, Giraph++ [15] proposes a subgraph-centric programming model in which user programs are implemented at the subgraph level rather than at the vertex level. Some more evolved systems like NScale [7] and Arabesque [14] also use similar subgraph-centric models ("neighborhoods" in the case of NScale and "embeddings" in the case of Arabesque), but these systems are also focused on some of the issues that arise when adopting a subgraph-centric model such as providing an interface for users to specify subgraphs, distributing subgraphs across machines, and automatic parallelization.

Moving forward, we will go into more depth discussing each of these graph processing paradigms. In particular, we will classify the above systems into one of three categories:

1. Vertex-Centric Approaches

2. Subgraph-Centric Approaches

3. Other Graph Processing Frameworks

Finally, we will present a set of issues and open research areas unaddressed by the current graph processing systems, which we hope to address in this paper.

## 2.1 Vertex-Centric Approaches

In recent years, the biggest graph datasets have been growing exponentially in size. As graphs become too large to store on a single machine, they must be distributed across a cluster of machines and they can no longer be processed in-memory by a single machine. "Think Like a Vertex" graph processing frameworks were developed in response to these challenges of analyzing large graphs on a single machine. Broadly speaking, these vertex-centric frameworks provide a natural way to parallelize computations on large graphs that are stored in a distributed fashion by taking user-defined functions and executing them in parallel on all vertices of the graph. The results of these computations are propagated across the graph until some convergence property is met.

Pregel [4] was one of the first and most influential vertex-centric systems. Typically, Data-Parallel MapReduce [1] systems are poorly suited for graph processing tasks due to the data interdependencies inherent to many graph algorithms. In most graph algorithms, vertex computations are dictated by the results of computation on adjacent vertices, making these algorithms difficult to parallelize. Pregel introduces a Graph-Parallel abstraction in contrast to the Data-Parallel abstraction provided by MapReduce systems.

Pregel uses a master-worker architecture in which a single master machine coordinates all worker machines. The master machine is responsible for managing partitions, coordinating worker activity, and fault tolerance. Pregel is based on the Bulk Synchronous Parallel (BSP) computation model [16]. At the beginning of a graph computation, the master is responsible for distributing the vertices of the graph across different worker tasks running on different machines. The graph processing task is broken down into iterations called supersteps, the worker tasks execute in parallel, and

then all workers synchronize at the end of each superstep. The graph processing task is implemented in a vertex-centric fashion in which user-defined vertex programs are executed on each vertex exactly once in every superstep. During execution vertices may communicate with each other through message passing. Typically, messages are sent to destinations vertices that are accessible from the outgoing edges of the source vertex.

Pregel has been used to create scalable implementations of standard graph algorithms like PageRank, Shortest Path, Semi-Clustering, and Bipartite-Matching.

**Challenges.** Vertex-centric systems such as Pregel limit the user program's access to solely the vertex on which the program is executing by design. Consequently, the biggest fundamental limitation with most vertex-centric systems is the high communication overhead and network traffic needed to execute many classes of graph algorithms. For instance, graph algorithms that work at the subgraph level rather than at the vertex level would have to rely on message passing for each vertex of the subgraph in order to reconstruct the subgraph locally if the vertices are scattered across the network.

On top of communication overhead, many vertex-centric systems almost always process the entire graph, which is not always necessary. For example, a user might want to only analyze the neighborhood around certain vertices of interest. This type of computation is not well-supported by vertex-centric approaches, which tend to rely on all vertices needing to meet a convergence property before execution stops.

**Optimizations.** In response to the big challenges faced by vertex-centric systems, there have been efforts to implement optimizations that address the bottlenecks of vertex-centric systems which are mainly the communication overhead and the number of supersteps required to complete the computation [12]. One of the most effective general approaches has been to finish graph computations serially. Given that some algorithms converge very slowly while working on only a tiny fraction of the input graph, one effective optimization has been to monitor the size of the active graph for computation and, once its size falls below a certain threshold, finish the rest of the computation serially in an effort to reduce the number of supersteps. Other approaches require algorithm-specific optimizations. For instance, "edge cleaning" is a common operation in graph algorithms that has vertices delete neighbors from their adjacency lists based on their values. Pregel handles this operation by having vertices send each other their values in one superstep and clean the adjacency lists in another superstep. One optimization would be to clean edges only as they are discovered, avoiding an unnecessary superstep.

**Variants of Vertex-Centric Systems.** On top of optimizations to vertex-centric systems, there are a set of components that many vertex-centric systems do differently depending on their intended use cases [5]. These components are the following:

1. Timing - How user-defined vertex programs are scheduled for execution. System execution can be synchronous, asynchronous, or hybrid.

2. Communication - How vertex program data is made accessible to other vertex programs. The two primary methods are message passing or shared memory.

3. Execution Model - Implementation of vertex program execution and flow of data. Vertex programs may execute many phases of computation. For instance, Gather-Scatter is a two phase model and Gather-Apply-Scatter is a three phase model. Additionally, data flow can be characterized as push or pull. In a push model, information flows from the active vertex to adjacent vertices. In a pull model, vertex programs ask adjacent vertices for information.

4. Graph Partitioning - How vertices of a graph are divided up and stored across a set of worker machines. Other than basic hash partitioning, there are systems that implement streaming partitioning, vertex cuts (i.e. partitioning a graph by its edges rather than by its vertices), and dynamic repartitioning.

Beyond the standard Pregel implementation, which uses a synchronous, single-phase push execution model with message passing and hash partitioning, many systems have explored different design decisions across these four dimensions. For instance, Ligra [13] uses shared memory for communication and found that the communication overhead was much lower with the tradeoff that extremely large graphs are not supported by the system. GraphX [18] uses a Gather-Apply-Scatter pull execution model and found that these choices led to an improved work balance and reduced data movement and enabled vertex-cut partitioning. In short, the number of possible variations is endless and specific design decisions only depend on the intended use cases for the system.

## 2.2 Subgraph-Centric Approaches

In this section, we shift paradigms from vertex-centric approaches to subgraph-centric approaches. Subgraph-centric approaches work at the subgraph level rather than at the vertex level. Generally speaking, subgraph-centric approaches to distributed graph processing systems seek to offer lower communication overhead, lower scheduling overhead, and lower memory overhead when compared to vertex-centric approaches. Many would argue that subgraph-centric approaches are also more natural and intuitive for many classes of problems [15]. The key insight with these approaches is that by having relevant edges and vertices for a computation grouped together locally on one machine rather than scattered across a network, network communication is vastly reduced, which can also lead to scheduling and memory gains. However, these systems also introduce a new class of problems. Specifically, it is important to consider how best to specify or select subgraphs and how partitioning and parallelization is affected with this new paradigm.

**Giraph++.** One of the first subgraph-centric distributed graph processing systems was Giraph++ [15]. This system primarily addressed the message passing overhead and scalability issues with the BSP model of computation used by systems like Pregel. One of the key issues in systems like Pregel is graph distribution often having a suboptimal number of edge cuts. As such, significant network communication is necessary for graph algorithms that require information from adjacent vertices. Giraph++ addresses this issue by creating subgraph partitions of graphs and distributing them to multiple machines and then running a sequential algorithm in a partition in each superstep.

On well-partitioned data, Giraph++ observes significant performance gains over vertex-centric systems. For instance, a graph-centric connected components algorithm on a graph of 118 million vertices and 855 million edges ran 63x faster than a comparable vertex-centric implementation and also had 204x fewer network messages.

Despite these performance gains, Giraph++ does lack some key features that become important when implementing subgraph-centric graph algorithms. First, Giraph++ does not give the user any means of specifying subgraphs of interest. Graphs are automatically partitioned into subgraphs by the system, which greatly limits the flexibility a user might require for implementing a performant graph algorithm. Secondly, Giraph++ executes computations serially within partitions with no support for parallelization. This leaves users to implement parallelization in user-defined functions, which is an extremely difficult task.

**Arabesque.** One system that addresses the issue of allowing users to specify subgraphs of interest is Arabesque [14]. Arabesque is described as a distributed data processing platform for implementing graph mining algorithms which aims to automate the process of exploring a very large number of subgraphs. The approach Arabesque uses is what the authors of the paper refer to as *"Think Like an Embedding"* in which an *embedding* denotes a subgraph representing a particular instance of a a more general subgraph *pattern*, which represents a set of graphs.

The key contributions of Arabesque are its use of embeddings as the basic building block for graph mining and its filter-process model which allows users to easily specify graph mining tasks. To begin, users provide *filter* and *process* functions and optional aggregation functions. Arabesque computations proceed in a sequence of supersteps in which the filter function is then used to identify embeddings of interest. Once these embeddings are identified, the process function runs on each embedding to generate output. At the end of computation, the aggregation function is executed if provided. Arabesque runs each of these supersteps in parallel by partitioning the set of embeddings over multiple servers.

Arabesque also implements a variety of optimizations to enable a coordination-free exploration strategy, support for storing embeddings compactly, efficient partitioning of embeddings for load balancing, and fast pattern canonicality checking to avoid duplicate work. These optimizations combined with the "Think Like an Embedding" paradigm lead to a scalable system well-suited for problems like frequent subgraph mining, counting motifs, and clique mining. The authors claim that in comparison to vertex-centric systems, Arabesque is clearly better. In comparison to *"Think Like a Pattern"* systems, which use a pattern-centric approach for graph mining, Arabesque is more scalable.

**NScale.** Another subgraph-centric system that provides an interface to allow users to specify subgraphs of interest and also offers better support for parallelization is NScale [7]. The key components of NScale include a *"neighborhood-centric"* programming model, a graph extraction and packing module responsible for subgraph selection and distributing selected subgraphs across machines, and a distributed execution engine that manages parallel execution of graph computations.

NScale provides an interface that allows users to specify subgraphs of interest by having users provide predicates to specify a set of query vertices based on vertex attributes, the radius around the query vertices to consider, and edge and vertex predicates for the neighborhood. Additionally, users specify the graph algorithm to run as a user-defined function to run on the neighborhoods specified by their provided predicates. NScale constructs neighborhoods of interest based on these predicates and then distributes them across machines using a bin packing algorithm. Finally, the distributed execution engine executes the graph algorithm with support for varying degrees of parallelization by running the user-defined function on each neighborhood.

Out of comparisons to Giraph (open-source version of Pregel), GraphLab, and GraphX, GraphX is the best performing alternative to NScale. GraphX does well for applications that use 1-hop neighborhoods, but performance deteriorates as the graph size increases. In short, NScale is a much more scalable solution, as would be expected with many subgraph-centric systems.

## 2.3 Other Graph Processing Frameworks

There have been several other alternatives to vertex-centric graph processing frameworks beyond subgraph-centric approaches.

**X-Stream.** X-Stream [9] is a system that runs on a single shared-memory machine that uses an edge-centric graph processing model with stream-based partitioning. The system uses a synchronous Scatter-Gather execution model in which scatter and gather functions are executed in phases on edges to dictate state changes maintained in the vertices. X-Stream uses streaming partitions because they enable sequential access to slow storage for edge and update streams. Additionally, like Ligra [13], X-Stream avoids the cost associated with network communication by being a single machine, shared-memory system.

Overall, X-Stream has limitations similar to those encountered by vertex-centric systems. The edge-centric view is fundamentally restrictive in the same way the vertex-centric view is for graph algorithms that analyze subgraphs or neighborhoods. Additionally, single machine, shared-memory systems are inherently not as scalable as distributed systems.

**GraphX.** One notable system is GraphX, [18] which attempts to address the limitations of vertex-centric systems by building a graph analytics framework on top of the existing distributed processing framework, Spark [19], and by modifying the vertex-centric model to be less restrictive. Instead of using the Graph-Parallel abstraction of Pregel in which user-defined vertex-programs are executed concurrently on vertices and in which communication between vertices is done through push-based message passing, GraphX uses a Gather, Apply, Scatter abstraction that leads to a pull-based model of message computation. Vertex programs ask adjacent vertices for messages rather than sending them. In addition, on top of GraphX, researchers have explored efficiently implementing high-level primitives for large-scale graph processing [11] to help programmers avoid the common pitfalls in implementing common graph processing tasks on their own.

## 2.4 Open Research Areas

This set of open research areas is not exhaustive, but it represents a few of the issues in contention between the different subgraph-centric systems we have evaluated so far.

We hope to address these issues in this paper.

**Fault Tolerance.** In pursuit of performance, many graph processing systems abandon fault tolerance in favor of snapshot recovery [18]. For instance, Pregel and Giraph++ achieve fault tolerance through checkpointing, and the papers for Arabesque and NScale fail to mention considerations for fault tolerance. GraphX is the only system we have encountered that provides very robust fault tolerance by relying on the lineage-based fault tolerance provided by Spark.

**Graph Partitioning** Another big challenge these systems face is efficient partitioning of graphs. Pregel implements hash-partitioning in which each vertex is assigned to a worker based on its hash value. This creates balanced partitions, but it also creates a suboptimal number of edges that span two different workers. With more efficient partitioning, there are great opportunities to reduce network communication during algorithm execution. Different vertex-centric systems have explored different ways of partitioning such as streaming partitioning, vertex cut partitioning, and dynamic repartitioning. When it comes subgraph-centric systems, partitioning challenges are similar, but there is also the additional challenge of efficiently packing subgraphs onto different machines.

**Subgraph Specification.** When dealing with subgraph-centric systems, interfaces that allow users to specify the subgraphs considered for execution enable more performant implementations of graph algorithms. Systems like NScale and Arabesque provide interfaces for users to specify subgraphs, but these are very nascent and lack flexibility. For example, with NScale's interface, it is hard to specify subgraph patterns like triangles.

# 3. NSPARK PROGRAMMING MODEL

## 3.1 Overview

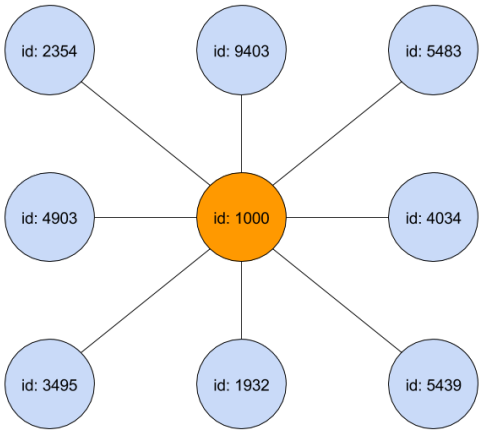
NSpark assumes a standard graph definition in which each graph, *G(V, E)* consists of a set of vertices, *V*, and a set of edges, *E*. One of the main goals of the NSpark system is to offer a programming abstraction that makes implementing graph processing tasks very simple. In order to implement a graph processing task, the user is required to provide (1) subgraphs of interest on which to run the computation, and (2) a user program. The user specifies subgraphs of interest by providing an *extraction query*, which consists of:

1. A list of vertex patterns. Each vertex pattern is implemented as a function that accepts a vertex, implements a filter on the vertex, and returns a Boolean value based on the results of the filter.

2. A list of edge patterns. Each edge pattern is a 2-tuple in which each component in the tuple contains an index value corresponding to the set of vertices associated with each vertex pattern. Each edge pattern asserts that, in the extracted subgraph, there must be an edge present between the two corresponding vertices.

3. A neighborhood size. This is an integer value that indicates the size of the neighborhood to extract around the subgraphs denoted by the vertex and edge patterns.

Within the user program, the user provides the following:




```
// extractOneHopNeighborhood.scala
val extractionQuery = (
   List(
      (v: Vertex) => v.id == 1000
   ),
   List(),
   1
)
```

**Figure 1: The 1-hop neighborhood around vertices with ID 1000.**

1. A map function. This map function executes on each of the extracted subgraphs and outputs a result.

2. A reduce function. The reduce function accepts a list of results from the map function executing on the extracted subgraphs and returns a user-defined aggregate result.

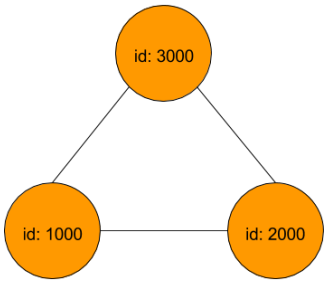
Overall, the key novelty in NSpark's programming model over NScale's is the *extraction query*. In NScale, the user provides a single vertex filter (analogous to vertex patterns in NSpark), which is used to extract *query vertices*, a neighborhood size, and a set of vertex and edge filters for the vertices and edges in the neighborhood of the *query vertex*. The key downside to this approach is that the user would not be able to easily express relationships between different vertices in the extracted subgraph. We address this issue by introducing the notion of *edge patterns* in NSpark's interface.

## 3.2 Subgraph Specification Interface

To illustrate both the power and simplicity of NSpark's subgraph specification interface, we present a few extraction queries and their corresponding extracted subgraphs.

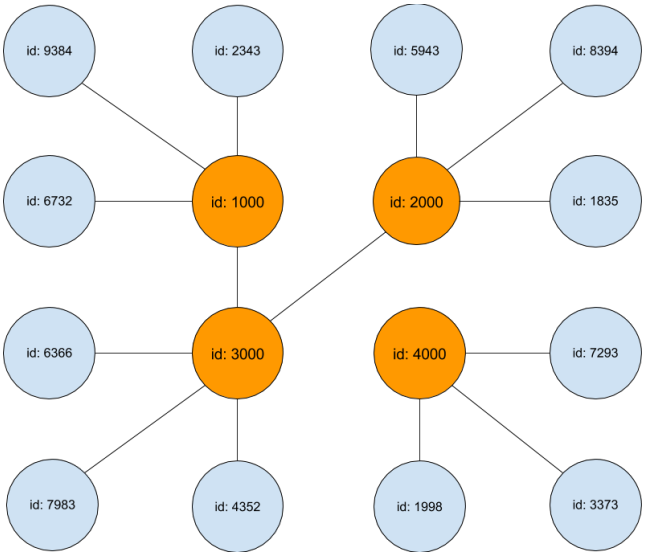
Taking a look at Figure 1, we present a simple subgraph pattern consisting of the vertices with ID 1000 and their 1-hop neighborhoods. This is the type of subgraph that could easily be specified by NScale's subgraph specification interface.

The subgraph pattern in Figure 2 is where we begin to have issues with NScale's subgraph specification interface. A triangle is a very common pattern in graph processing tasks, but since NScale only allows users to provide a filter for a single set of query vertices, it becomes extremely difficult to express patterns like triangles in graphs.




```
// extractTriangle.scala
val extractionQuery = (
   List(
      (v: Vertex) => v.id == 1000,
      (v: Vertex) => v.id == 2000,
      (v: Vertex) => v.id == 3000
   ),
   List((0, 1), (0, 2), (1,2)),
   0
)
```

**Figure 2: A triangle pattern between vertices with ID's 1000, 2000, and 3000.**




```
// extractDisconnectedVertex.scala
val extractionQuery = (
   List(
      (v: Vertex) => v.id == 1000,
      (v: Vertex) => v.id == 2000,
      (v: Vertex) => v.id == 3000,
      (v: Vertex) => v.id == 4000
   ),
   List((0, 2), (1, 2)),
   1
)
```

**Figure 3: The 1-hop neighborhood around subgraphs with vertices with ID's 1000, 2000, 3000, and 4000 and edges between vertices 1000 and 3000 and 2000 and 3000.**

```
// Compute the number of triangles containing each
// vertex
val triCountGraph = graph.triangleCount()
// Determine the number of triangles possible at
// each vertex
val maxTrisGraph = graph.degrees
   .mapValues(d => d * (d - 1) / 2.0)
// Normalize the number of triangles at a vertex by
// the number of triangles possible at the vertex
val clusterCoefGraph = triCountGraph.vertices
   .innerJoin(maxTrisGraph) {
      (vertexId, triCount, maxTris) => {
         if (maxTris == 0) 0 else triCount / maxTris
      }
   }
```

**Figure 4: An implementation of computing the Local Clustering Coefficient using the GraphX programming model from [10].**

The example in Figure 3 shows a subgraph pattern that is impossible to express with NScale's subgraph specification interface. The subgraph pattern consists of three vertices with another disconnected vertex along with their 1-hop neighborhoods.

Throughout these three examples, it is important to note that, despite the increasing complexity of the extracted subgraphs, the complexity of the extraction queries remained the same.

## 3.3 Local Clustering Coefficient

To illustrate an the end-to-end process of implementing a common graph processing task using the NSpark programming model and how it compares to using popular graph processing frameworks like GraphX, we provide two implementations for computing the local clustering coefficient [17].

Clustering coefficients are measures of the degree to which vertices in a graph tend to cluster together. The local clustering coefficient of a vertex is a measure that quantifies how close that vertex and its neighbors are to forming a clique (a complete graph). The local clustering coefficient $C$ for a vertex that has $k$ neighbors and $t$ triangles is:

$$\frac{2t}{k(k-1)} \tag{1}$$

In the GraphX implementation of computing the local clustering coefficient (Figure 4), we need to compute *triCountGraph*, which contains the number of triangles formed at each vertex in the graph, and *maxTrisGraph*, which contains the maximum number of triangles that can be formed at each vertex, which can be derived from the vertex's degree. Once these values are computed, we do a join on the two tables and compute the local clustering coefficient with the joined values.

In the NSpark implementation of computing the local clustering coefficient, we build an extraction query to specify the subgraphs of interest, which include the 1-hop neighborhood around every vertex in the graph, and then we write a map function that computes the local clustering coefficient for each subgraph of interest. This map function computes the local clustering coefficient computing the number of triangles in the subgraph and dividing by the number of max-

```
// Specify the subgraphs of interest, which consists
// of the 1-hop neighborhood around each vertex
val extractionQuery = (
   List((v: Vertex) => true),
   List(),
   1
)
// Specify the map function, which computes the local
// clustering coefficient for the extracted subgraph
val mapFn =
   (subgraph: Graph, queryVertices: List[Vertex]) => {
      val deg = subgraph.getDeg(queryVertices(0))
      val maxTris = deg * (deg - 1) / 2.0
      val triCount = subgraph.triangleCount()
      if (maxTris == 0) 0 else maxTris / triCount
   }
// Specify a reduce function, which is not needed in
// this case
val reduceFn = None
// Run the Local Clustering Coefficient Algorithm
GraphExecutor.run(
   graph, extractionQuery, mapFn, reduceFn
)
```

**Figure 5: An implementation of computing the Local Clustering Coefficient using the NSpark programming model.**

imum possible triangles at the vertex associated with the subgraph, which is derived from the vertex's degree.

As shown in the NScale paper [7], this type of local clustering coefficient implementation is comparable to the one for GraphX, and even vastly outperforms the GraphX implementation on densely connected graphs. Even with this performance boost, the NSpark implementation remains relatively simple compared to the GraphX implementation. In the NSpark implementation, the ease of use of the subgraph specification interface, the subgraph-centric view of computation, and the Map-Reduce [1] model for defining user programs afford the user the luxury of only having to think about the local clustering coefficient computation in the context of a single vertex and its 1-hop neighbors. Contrast this with the GraphX implementation in which the user has to think about various joins and maps on potentially large tables and it becomes clear that NSpark offers a simpler interface for this class of graph algorithms.

# 4. NSPARK IMPLEMENTATION

## 4.1 Overview

NSpark is a subgraph-centric, distributed graph processing framework built on top of Apache Spark [19]. The NSpark system is divided into two key components, which include a Graph Extraction and Packing (GEP) module, which is responsible for extracting subgraphs of interest from the input graph and a Distributed Execution Engine, which is responsible for executing distributed graph processing tasks on the extracted subgraphs. Prior to the execution of any graph processing task, a user will provide an input graph, $G$, an extraction query to specify subgraphs of interest, and a user program in the form of Map and Reduce functions to execute on the subgraph. The GEP module will translate the extraction query in to a series of Spark dataflow operators, which will then be executed to extract the subgraphs of interest out of the interest graph $G$. Next, the GEP module will run a shingle-based graph partitioning algorithm [8] to pack the extracted subgraphs into different partitions across distributed memory using the Spark API. Finally, the Distributed Execution Engine will run the user-provided Map and Reduce functions in parallel on the packed subgraphs.

NSpark is largely based on the work done in NScale [7]. The main distinctions between these two systems are that NSpark is written in the Scala programming language and is built on top of Apache Spark while NScale is written in Java. Additionally, NSpark offers a more sophisticated subgraph specification interface than what is offered by NScale, allowing users to more naturally specify complex subgraph patterns. As shown in the prior NScale paper [7], compared to graph processing frameworks like GraphX, subgraph-centric systems like NScale and NSpark are more suitable for a large number of graph processing tasks, especially those that require high communication between vertices in the graph.

## 4.2 Leveraging Apache Spark

One of the most critical design decisions for NSpark was to build the entire system on top of Apache Spark [19]. Other systems like GraphX [18] have leveraged Apache Spark to great success, observing that there is a very close relationship between common graph operators and the various dataflow operators provided by Spark. With that in mind, we decided to build on top of Spark for the following reasons:

1. **Fault Tolerance.** Many specialized graph processing systems offer very poor guarantees in regards to fault tolerance. These systems often abandon fault tolerance in favor of snapshot recovery to achieve greater performance. By building on top of Spark, NSpark automatically offers the lineage-based fault tolerance of Spark RDD's in which Spark logs the lineage of operations for each RDD, which enables automatic reconstruction upon failures.

2. **Rich Dataflow Operators.** GraphX discovered that many graph computations can be efficiently translated into a series of Spark dataflow operators, which becomes especially useful when dealing with distributed memory. In particular, the Graph Extraction and Packing module in NSpark exclusively uses Spark dataflow operators for extracting subgraphs of interest.

3. **User-Defined Data Partitioning.** Spark offers fine user control over how data is partitioned. This is useful in the context of our Graph Packing implementation, which attempts to find efficient configurations for storing different subgraphs across distributed memory.

4. **High-Level Scala API.** Spark offers an extensible, high-level Scala API, which allows us to implement the Distributed Execution Engine using the Scala Actor system. The Scala Actor system is a well-regarded tool for building high-concurrency systems, which makes perfect sense for NSpark.

## 4.3 Graph Extraction and Packing Module

The Graph Extraction and Packing Module is written using dataflow operators provided by Apache Spark and is responsible for extracting subgraphs of interest and determining an efficient configuration for packing the different subgraphs onto different machines. The user specifies the subgraphs of interest by providing an *extraction query*, which

consists of a list of *vertex patterns*, a list of *edge patterns*, and a neighborhood size, and the GEP module translates this extraction query into a sequence of Spark dataflow operators. These operators are executed first to construct the neighborhoods for each vertex, then to construct the subgraphs corresponding to the vertex and edge patterns, and then finally to join the neighborhoods of each vertex to the constructed subgraphs to produce the final subgraphs of interest. This process is done entirely within Spark's Resilient Distributed Datasets (RDD) [19].

#### 4.3.1 Graph Extraction

To go into more detail on how neighborhood construction is implemented, consider Figure 6. The simplest case is a 1-hop neighborhood, which only requires a single *groupByKey* on the edge list. For neighborhoods of size $k$ with $k$ ¿ *1*, the general process is to initialize an RDD of *neighborhoods* as the adjacency list (an RDD containing a vertex's 1-hop neighbors), re-key the RDD with a *flatMap* transformation such that each neighbor in a vertex's neighborhood becomes a key, and then *join* this RDD with the adjacency list to introduce all of the neighbors of the vertices in the prior iteration's neighborhood into the RDD. This process is repeated for $k$ - *1* iterations. Given this approach, it is important to note that on each iteration, the size of the RDD's being joined increases by a factor of $n$ with $n$ being the average size of the neighborhoods.

In order to construct the subgraph patterns, consider Figure 7. The first step is to filter the *edge list* using each *vertex pattern* to generate a set of vertices corresponding to each vertex pattern. The *subgraphs* RDD is initialized as the first constructed edge pattern. Edge patterns are constructed by joining the left and right sides of the edge list with the corresponding vertex sets retrieved with the vertex patterns. Then, the remaining edge patterns are constructed one-by-one. The final subgraphs RDD is constructed by joining all of the constructed edge patterns incrementally on the initial subgraphs RDD, and then joining in the neighborhood for each vertex in each subgraph from the neighborhoods RDD to complete the extraction query.

#### 4.3.2 Graph Packing

In order to pack the extracted subgraphs onto different machines across distributed memory, we employ the same shingle-based partitioning algorithm used by NScale (Figure 8). The key idea is to choose a set of $k$ different random pairwise independent hash functions, use each to compute the hash of each vertex in a subgraph, and take the minimum result for each hash function as a shingle value. Subgraphs of interest are sorted based on the single values associated with them in a lexicographical fashion, which will place subgraphs with high overlap in close proximity to each other. Finally, a greedy algorithm is used to pack subgraphs into different partitions.

### 4.4 Distributed Execution Engine

The Distributed Execution Engine component of NSpark is responsible for executing the user program, specified as a pair of *Map* and Reduce functions, on the extracted subgraphs in a parallel fashion. The Distributed Execution Engine is written in Scala and makes heavy use of the Scala Actor system, which is based on the Actor model defined by Hewitt [2]. Scala Actors provide a high level of abstraction for writing concurrent and distributed systems. They rely on message passing for communication and free the programmer from handling finer details like explicit locking or thread management, making it easier to write correct programs.

#### 4.4.1 Custom-Built Graph Data Structure

As a first step to building out the Distributed Execution Engine, we need to build a custom graph data structure since Scala fails to provide a graph data structure in its standard library. NScale [7] used the Java BluePrints API[1], which provides a graph data structure with many standard functions for interacting with a graph, so we seek to replicate that functionality in Scala.

The graph data structure we built stores a set of vertices and a set of edges as fields. In addition, it uses the adjacency list representation of graphs and stores a Map that contains the set of edges associated with each source vertex and a Map that contains the set of edges associated with each destination vertex. The graph data structure also provides a set of common functions for interacting with graphs, which include getting the neighbors of a vertex, getting the edges associated with a vertex, getting the degree of a vertex, and adding and removing vertices and edges from the graph.

One notable detail about the graph data structure we implemented is support for denoting different subgraphs. One of the findings from the original NScale paper was that by storing multiple subgraphs in a single graph data structure, memory usage during computation can decrease significantly. The scheme NScale used involved storing a BitMap along with each edge and vertex in the graph. The BitMap would have a size equal to the number of subgraphs packed into the graph data structure, and each index in the BitMap would correspond to a specific subgraph. Vertices and edges were marked as part of a subgraph by having the index corresponding to that subgraph in the BitMap set to 1. This idea eliminates the need to store multiple copies of vertices and edges in the case of overlapping subgraphs. As such, similar to NScale, NSpark stores a Scala BitSet along with each vertex and edge in the graph data structure.

#### 4.4.2 Distributed Execution with Scala Actors

In order to execute graph processing tasks in a distributed fashion, we built a set of Scala classes called *SubgraphExecutor*, *SubgraphMapper*, and *SubgraphReducer*. The general execution scheme is described in Figure 8. The SubgraphExecutor is the starting point for distributed graph computations. It accepts a graph contained within our custom graph data structure, a Map function, and a Reduce function. The SubgraphExecutor will spawn a SubgraphReducer and a set of SubgraphMapper's and send a message to each of the SubgraphMappers consisting of the graph data structure, a subgraph ID, and the Map function. The SubgraphMapper will run the Map function on the graph data structure with the access of the Map function limited to only vertices and edges with a set bit in their BitSet corresponding to the subgraph ID passed in. Once the execution of the Map function is complete, the Subgraph Mapper sends a message to the SubgraphReducer. Once the SubgraphReducer has collected messages from all of the SubgraphMapper's,

[1]https://github.com/tinkerpop/blueprints/wiki

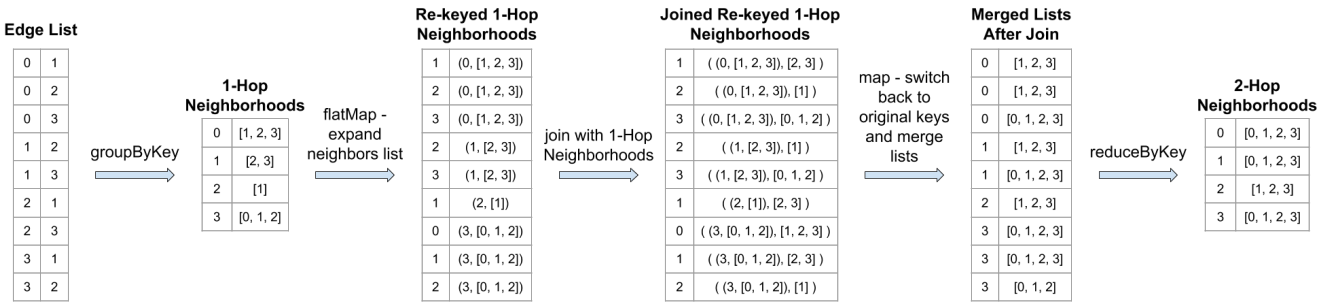



**Figure 6: An example of the sequence of Spark dataflow operators needed to construct 2-hop neighborhoods. Beginning with an edge list, a groupByKey operation can be used to construct the 1-hop neighborhoods. Then, a flatMap to expand the neighbor list as keys, a join with the 1-hop neighborhoods, and a map and reduceByKey can be done to retrieve the 2-hop neighborhoods.**

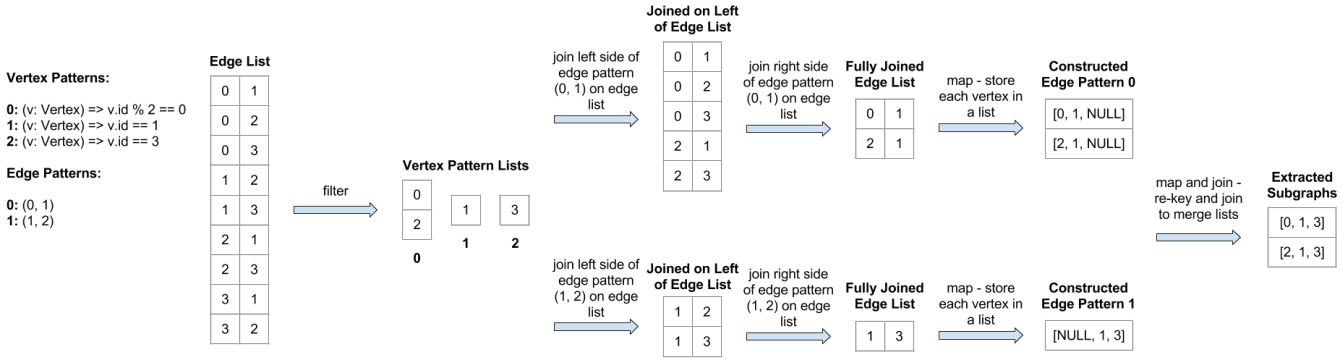



**Figure 7: An example of constructing the subgraphs specified by the sample vertex and edge patterns. First, each vertex pattern is applied to the edge list to get a set of vertices for each vertex pattern. Then, for each edge pattern, the edge list is joined on both sides with the vertices for the corresponding vertex patterns. Finally, the constructed edge patterns will be joined together to produce the set of extracted subgraphs.**

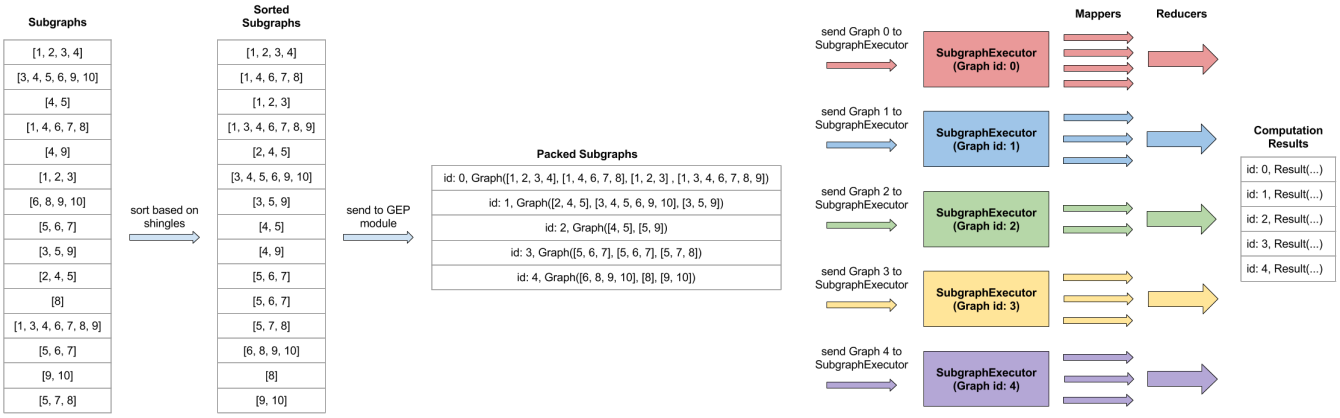



**Figure 8: An illustration of NSpark's shingle-based graph packing algorithm and distributed execution engine. After building an RDD of subgraphs from the extraction query, compute a set of shingle values for each subgraph and sort the RDD based on the lexicographical order of the shingle values associated with each subgraph. Use a greedy graph packing algorithm to load overlapping subgraphs into a custom-built Scala graph data structure, initialize a SubgraphExecutor for each subgraph, and then run the user-provided Map and Reduce functions.**

the SubgraphReducer runs the Reduce function on the collected results. Finally, it sends this final result back to the SubgraphExecutor and the task is finished.

During distributed execution, a SubgraphExecutor is spawned for each graph data structure in the packed subgraphs RDD produced by the Graph Extraction and Packing module. Since the RDD is stored in distributed memory, this ensures that we have both parallelism within a machine and distribution across many machines.

# 5. FUTURE WORK

By taking the best ideas from NScale [7], adding a more flexible subgraph specification interface, and building the system on top of Apache Spark [19], we hope to offer NSpark as an overall improved version of NScale. However, in order to be certain of the correctness of many of our design decisions, it will be necessary to conduct a comprehensive experimental analysis comparing the performance and memory usage of NSpark against NScale and other comparable graph processing systems. We hope to conduct this experimental analysis, use the results to tune the implementations of the algorithms described in this paper, and then eventually bring NSpark to a state in which it would be suitable for releasing as an open-source graph processing system.

# 6. REFERENCES


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

[2] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[3] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[5] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[6] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[7] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric analytics on large graphs. *Proceedings of the VLDB Endowment*, 7(13):1673–1676, 2014.

[8] A. Rajaraman, J. D. Ullman, J. D. Ullman, and J. D. Ullman. *Mining of massive datasets*, volume 1. Cambridge University Press Cambridge, 2012.

[9] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[10] S. Ryza et al. Advanced analytics with Spark. ed. *by Ann Spencer. O'Reilly*, 2014.

[11] S. Salihoglu and J. Widom. High-level primitives for large-scale graph processing. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[12] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[13] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[14] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.

[15] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

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

[17] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[18] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.