HelP: High-level Primitives For Large-Scale Graph Processing

Semih Salihoglu
Stanford University
semih@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

Large-scale graph processing systems typically expose a small set of functions, such as the *compute()* function of Pregel, or the *gather()*, *apply()*, and *scatter()* functions of PowerGraph. For some computations, these APIs are too low-level, yielding long and complex programs, but with shared coding patterns. Similar issues with the MapReduce framework have led to widely-used languages such as *Pig Latin* and *Hive*, which introduce higher-level primitives. We take an analogous approach for graph processing: we propose *HelP*, a set of high-level primitives that capture commonly appearing operations in large-scale graph computations. Using our primitives we have implemented a large suite of algorithms, some of which we previously implemented with the APIs of existing systems. Our experience has been that implementing algorithms using our primitives is more intuitive and much faster than using the APIs of existing distributed systems. All of our primitives and algorithms are fully implemented as a library on top of the open-source GraphX system.

1. INTRODUCTION

Processing large-scale graph-structured data is an important task for many applications across different domains, such as the web [48], social networks [36], biology [53], telecommunications [31], and many others. As graphs grow to sizes that exceed the memory of a single machine, applications perform their computations on distributed and highly-parallel shared-nothing systems, such as *MapReduce* [9] and *Hadoop* [17], Hadoop's iterative extensions [10, 52], or more specialized graph systems such as *Pregel* [28] and *PowerGraph* [14]. At the core of the APIs of these systems is a small set of functions, such as the *map()* and *reduce()* functions of MapReduce, the *compute()* function of Pregel, or the *gather()*, *apply()*, and *scatter()* functions of PowerGraph.

The benefit of these frameworks is that programmers concentrate on implementing a few specific functions, and the framework automatically scales the computation by executing these functions in parallel across machines. However, sometimes the functions are too low-level or restricted for the algorithmic task. For example, as observed in references [18, 20, 40], additional code and workarounds that use global data structures can be required to control the flow of some algorithms, yielding complex and long programs. In addition, custom code can be required for some commonly appearing operations, such as initializing vertex values or checking for convergence of computations. For performing large-scale data analysis tasks on Hadoop [17], there has been an emergence of higher-level languages such as Pig Latin [34] and Hive [47], which have seen wide adoption in the industry and research communities [12, 47]. These languages express data analysis tasks with high-level constructs and primitives, such as filters and grouping, and compile to the lower-level *map()* and *reduce()* functions of Hadoop. We believe similar higher-level primitives for graph analysis tasks would be very useful for programming large-scale graph computations.

In a previous paper [41], we implemented in detail several graph algorithms on an open-source Pregel clone. In the process, we observed certain patterns emerge from our implementations that we believed could be abstracted to a useful set of higher-level graph processing primitives. We implemented an additional suite of algorithms to verify our sense of the most generally useful primitives. This paper reports on the primitives we identified, called *HelP*, our implementation of these primitives using the functions available on the *GraphX* system [50], and the implementation of many graph algorithms using the primitives.[1] For reference, Tables 1 and 2 list our primitives and the algorithms we have implemented using our primitives, respectively.

Our HelP primitives can be grouped broadly into three areas, according to the operations they perform:

1. **Vertex-centric Updates**: Update the values of some or all of the vertices in parallel. An update of a vertex value may use information from edges incident to the vertex (hereafter "local edges"), or from the values of other vertices. Updates can happen in iterations or in a single iteration.
2. **Topology Modifications**: Modify the topology of the graph by removing some vertices or edges based on filtering conditions, or merging multiple vertices together to form new vertices.
3. **Global Aggregations**: Perform a global aggregation operation over some or all of the graph (e.g., find the average degree, or find the vertex furthest from a given one).

All of the primitives in HelP abstract parallel operations that are suitable for scalable distributed implementations. We have implemented the HelP primitives fully in GraphX [50]. We note that some of the primitives in HelP were already part of GraphX; and we specify those primitives clearly later in this paper.

The specific contributions of this paper are as follows:

[1] Our primitives and algorithms target synchronous graph processing engines, e.g., [2, 13, 28, 40]. Identifying high-level primitives for asynchronous graph systems is an interesting future research direction [14, 25].

Primitive	Description
Filter	Removes some vertices or edges from the graph.
Aggregating Neighbor Values (ANV)	Some vertices aggregate some or all of their neighbor values to update their own values. Appears both as a one-step computation and as an iterative process. In the iterative version, some or all of the vertices start propagating a value to their local neighbors, which are aggregated and propagated further by receiving vertices in the next iteration. The propagations continue until vertex values converge.
Local Update of Vertices (LUV)	Updates vertex values, possibly using global information or local edges of each vertex. Used mainly for initializing vertex values.
Update Vertices Using One Other Vertex (UVUOV)	Updates vertex values by using a value from one other vertex (not necessarily a neighbor). Commonly used in matching-like algorithms.
Form Supervertices (FS)	Merges groups of vertices.
Aggregate Global Value (AGV)	Computes a single global value over the graph. A commonly used special case is picking a random vertex from the graph.

Table 1: Primitives.

Name	Filter	ANV	LUV	UVUOV	FS	AGV	Data Primitives
PageRank [28]		x	x				
HITS [24]		x	x			x	
Single Source Shortest Paths [28]		x	x				
Weakly Connected Components [23]		x	x				
Conductance [3]		x				x	
Semi-clustering [28]		x	x			x	
Random Bipartite Matching [28]	x		x	x			
Approx. Betweenness Centrality [1]		x	x			x	
Diameter Estimation (Double Fringe) [38]		x	x			x	
Strongly Connected Components [41]	x	x	x			x	
Minimum Spanning Forest [41]	x		x	x	x		
Approx. Maximum Weight Matching [41]	x			x			
Graph Coloring [41]	x	x	x				
Maximal Independent Set [41]	x	x	x				
K-core [38]	x					x	
Triangle Finding [38]							x
Clustering Coefficient [38]	x						x
K-truss [38]							x
Simple METIS [29]	x			x	x		x
Multilevel clustering [33]	x			x	x		x

Table 2: Algorithms.

- In Section 3, we describe three primitives and their variations, all of which perform vertex-centric updates: (1) `updateVertices`; (2) `aggregateNeighborValues`; and (3) `updateVertexUsingOneOtherVertex`.
- In Section 4, we describe the `filter` and `formSupervertices` primitives and their variations, all of which modify the topology of the graph.
- In Section 5, we describe the `aggregateGlobalValue` primitive, which performs a global aggregation operation over the vertices of the graph.
- In Section 6 we give examples of algorithms that benefit from using additional "data primitives" such as joins and grouping, either instead of or in addition to our HelP primitives. These examples are naturally expressed in an "edge-centric" fashion (rather than "vertex-centric"), such as triangle-finding, clustering coefficient, or finding k-trusses.

Section 2 provides necessary background on Spark [51] and GraphX [50]. Section 7 covers related work. Section 8 concludes and proposes future work. All of the HelP primitives and algorithms described in the paper are implemented on top of GraphX and publicly available [19].

Notably absent from this paper is any experimental evaluation. Evaluating programmer productivity is difficult to do objectively and convincingly. Lines of code is one possible metric to evaluate productivity gains of new APIs and languages, however the shortcomings of LOC as a measure have long been observed [39]. For the algorithms in Table 2, we saw up to 2x code reduction using our primitives against coding in GraphX without them. We could also evaluate the performance of algorithms implemented using our primitives against using other libraries on top of GraphX, or programming in GraphX directly. However, all of these approaches, when programmed carefully, translate to similar GraphX and Spark calls at the lowest levels, and thus similar expected performance. Finally, evaluating HelP against other graph processing systems would yield the same comparison as those systems against GraphX, as covered in [50].

2. BACKGROUND

We review the Spark computation engine and the GraphX system

built on top of Spark.

2.1 Spark

Spark [44, 51] is a general open-source distributed system for large-scale data processing. Spark was developed initially at UC Berkeley, and as of 2013 is an Apache Incubator project. Spark's programming model consists of two parts: (1) an in-memory data abstraction called *resilient distributed datasets* (RDDs); and (2) a set of deterministic parallel operations on RDDs that can be invoked using *data primitives*, such as `map`, `filter`, `join`, and `groupBy`. RDDs can either be created from data residing in external storage, such as a distributed file system, or can be derived from existing RDDs by applying one of Spark's data primitives. For example, the `map` primitive transforms each element `t` in an RDD by applying a user-specified `mapF` function to `t`, and returns a new RDD. RDDs are partitioned across a cluster of machines and each data primitive on RDDs executes in parallel across these machines. Spark is implemented in Scala and RDD elements can contain arbitrary Java objects. A Spark program is a directed acyclic graph of Spark's primitives, which are applied on a set of initial and derived RDDs. Here is a simple example program from the original Spark paper [51], which counts the number of lines that contain errors in a large log file:

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(AggrFn.SUM)
```

Values `file`, `errs`, and `ones` in the above code are RDDs and `count` is an integer numeric value. Spark stores the *lineage* of RDDs, i.e., the chain of operations that were applied to construct an RDD. In case of failures, Spark uses the lineage information to rebuild lost partitions by performing each operation in the lineage consecutively. Programmers can perform iterative computations very easily on Spark by using Scala's loop constructs, such as `for` and `while`. In addition to iterative computations, Spark supports interactive computations through the Spark interpreter.

2.2 GraphX

GraphX is a distributed graph engine built on top of Spark. GraphX stores a graph in RDDs (Section 2.2.1 below) and supports a set of operations on the graph that can be invoked using GraphX's *core graph primitives* (Section 2.2.2). In addition, GraphX contains `Pregel` and `PowerGraph` libraries, which are built on top of the core graph primitives. These libraries implement the Pregel and PowerGraph APIs, which programmers can use to implement graph algorithms. Our HelP primitives are built as another library on top of the core GraphX primitives. Figure 1 shows the hierarchy of primitives belonging to Spark, GraphX, and the libraries on top of GraphX. To set the stage for explaining our library and how it was implemented, in the next subsections we review the RDDs GraphX uses to store a graph, and some of the core graph primitives in GraphX.

2.2.1 GraphX RDDs

GraphX contains three core RDDs to store a graph:

- **`EdgesRDD`:** Stores the edges of the graph along with edge values. `EdgesRDD` is partitioned evenly across machines.
- **`VerticesRDD`:** Stores the vertices of the graph along with vertex values. `VerticesRDD` is also partitioned evenly across the machines. Note that `VerticesRDD` does not contain information about the edges incident on the vertices.
- **`RoutingTableRDD`:** Stores for each vertex `v` the edge partitions that contain the edges where `v` appears as a source or a destination vertex.

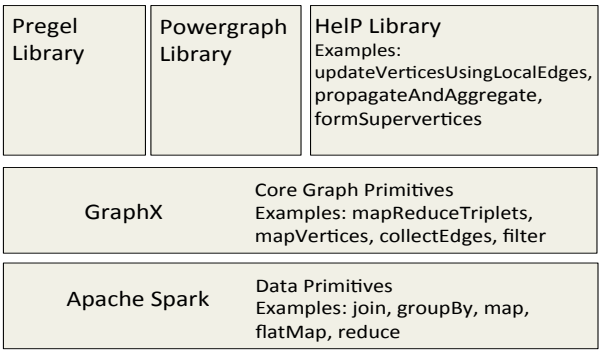

Figure 1: Hierarchy of primitives.

2.2.2 Core GraphX Primitives

We review the core GraphX primitives that we used for implementing our high-level graph primitives. Readers may wish to skip ahead to our high-level graph primitives (Sections 3-5) and come back to this section as a reference when we describe the implementations of our HelP primitives. For a comprehensive list of all core GraphX primitives, please see [15].

- **`filter`:** Takes a vertex predicate `vP` and an edge predicate `eP`. Returns a new graph that contains each vertex `v` for which `vP` evaluates to true, and each edge `e` of the original graph if: (1) `eP` evaluates to true on `e`; and (2) `vP` evaluates to true on both the source and destination vertex of `e`.
- **`mapVertices`:** Takes as input a `mapF` function and applies it to each vertex `v` of the graph to generate a new value for `v`. The edges of the graph remain unchanged.
- **`collectEdges`:** Takes as input an edge direction, one of `In`, `Out`, or `Either`, and returns a new RDD that contains, for each vertex `v` in the graph, a tuple with `v`'s ID, data, and edges in the user-specified direction. The returned RDD is called `ExtendedVerticesRDD`.
- **`getTriplets`:** Returns an RDD called `TripletsRDD` that contains an *edge triplet* of the form (`eval`, `srcObj`, `dstObj`) for each edge `e`=(`u`,`v`). `eval` is `e`'s value, while `srcObj` and `dstObj` contain the ID and value of the source `u` and destination `v`, respectively.
- **`mapReduceTriplets`:** Takes three inputs:
  - `tripletMapF`: A function whose input is an *extended edge triplet*: an edge triplet as above but with an additional field on the `srcObj` and `dstObj` indicating whether `u` and `v` are in the user-specified `optVertices` set (see below). The output of `tripletMapF` is a list of "messages" of the form (`vertexID`, `msg`). The first field is the ID of a vertex, and the second field is the message, which can be any Java object.
  - `reduceMessagesF`: A function that takes as input a list of messages that were "sent" to the same vertex by `tripletMapF`, and returns an aggregated message of the same type.
  - `optVertices`: A possibly empty set of vertex IDs. If the ID of a vertex `v` is in `optVertices`, this information is exposed in the extended edge triplet input to `tripletMapF`.

  `mapReduceTriplets` is generally used in graph operations that are naturally expressed by vertices sending and receiving messages to and from their neighbors. For each edge `e`=(`u`,`v`) in the graph, `mapReduceTriplets` applies `tripletMapF`

to e's extended edge triplet and generates a message to u, v, or both u and v. Then reduceMessagesF is applied to the list of messages sent to each vertex v in the graph to generate a single aggregated message for v. The output of mapReduceTriplets is an RDD called MessagesRDD, that contains for each vertex v that was sent at least one message, one tuple of the form (vertexID, aggregatedMsg). We use mapReduceTriplets in our implementation of the propagateAndAggregate primitive (Section 3.2.2).

- **diff:** Unlike the previous primitives, which operate on the graph, diff operates on two VerticesRDDs, old and new. Given old and new, diff returns a new VerticesRDD that contains each vertex v in old that exists and contains a different value in new, along with v's new value.

3. VERTEX-CENTRIC UPDATES

In this section and the following two we present our HelP primitives. For each primitive we first give a high-level description of the primitive, then we give one or more examples from our algorithms that use the primitive. Finally, we explain how the primitive is implemented in GraphX. We note that except for global aggregations (Section 5), all of our primitives return a new graph. In our code snippets, we omit the assignment of the returned graph to a new variable.

3.1 Local Update of Vertices

One of the most commonly appearing graph operations is to initialize the values of some or all of the vertices, either at the beginning of an algorithm, or at the start of each phase of an algorithm. Depending on whether the local edges of the vertices are used in the updating of vertices, we provide two primitives for this operation: updateVertices and updateVerticesUsingLocalEdges. updateVertices takes two inputs:

- vP: A predicate to select which vertices to update.
- updateF: A function that takes a vertex and returns a new value for the vertex.

Not surprisingly, the behavior of updateVertices is to update the values of all vertices for which vP evaluates to true, with the updated vertex value returned by updateF. Other vertices remain unchanged. updateVerticesUsingLocalEdges takes an additional edge direction input dir which specifies whether the incoming, outgoing, or both types of incident edges are used in updateF.

3.1.1 Examples of Use

Consider the first step of PageRank [4], which initializes the value of each vertex to $\frac{1.0}{|V|}$. Using updateVertices, we can express this operations as follows:

```
g.updateVertices(v → true, v → { v.val.pageRank = 1.0/ g.numVertices; v;}
```

As an example use of updateVerticesUsingLocalEdges, consider the bipartite matching algorithm from [28]. The vertices of the input bipartite graph are divided into *L* and *R*, for left and right, respectively. In each iteration of the algorithm, every unmatched vertex v_l in *L* randomly picks one of its neighbors from *R*, say v_r, and stores the ID of v_r in the pickedNbr field of its value. We can express this operation using updateVerticesUsingLocalEdges as follows:

```
g.updateVerticesUsingLocalEdges(EdgeDirection.Out, v → v.isLeft,
  (v, edges) → {
    v.val.pickedNbr = edges.get(getRandom(edges.length)).dstID; v;})
```

3.1.2 Implementation

GraphX's core primitive mapVertices (Section 2.2.2 above) already contains the functionality of updateVertices, but we added a new primitive for convenience, since mapVertices does not take predicate vP as an explicit input. We implement updateVertices simply by pushing predicate vP into the updateF function of mapVertices.

Implementation of updateVerticesUsingLocalEdges is a bit more complex. We first call GraphX's collectEdges primitive on the graph with the user-specified edge direction. Recall from Section 2.2 that collectEdges returns ExtendedVerticesRDD, an extension of the VerticesRDD containing the local edges of each vertex in the specified direction. Similar to updateVertices, we push predicate vP inside updateF, and we call Spark's map primitive with updateF on ExtendedVerticesRDD to generate a new VerticesRDD.

3.2 Aggregating Neighbor Values

In another common form of vertex-centric update operation, vertices aggregate some or all of their neighbors' values to update their own values. PageRank, HITS, finding shortest paths from a single source, finding weakly-connected components, or computing the conductance of a graph, are some of the example algorithms that perform this operation. Aggregating neighbor values appears in algorithms as a one-step computation, or as an iterative process that continues until vertex values converge. We provide the aggregateNeighborValues primitive for the one-step version, and propagateAndAggregate for the iterative version.

aggregateNeighborValues: The inputs to aggregateNeighborValues are listed in Table 2a. vP is a predicate that selects which vertices to update. If v is a vertex whose value should be updated, the input dir and the nbrP predicate determine the neighbors of v whose values should be aggregated in updating the value of v. Function aggregatedValueF is applied to the values of these neighbors, the outputs of aggregatedValueF are aggregated using aggregateF, and finally updateF is applied on v and the output of aggregateF to compute the new value for v.

propagateAndAggregate: In some computations the aggregation of neighbor values continues in iterations until all vertex values converge. The common pattern of such computations is the following: In the first iteration, one or more vertices propagate a value to their neighbors. Vertices that receive propagated values aggregate them and update their own values. In the next iteration, all vertices whose values have changed propagate a new value to their neighbors. The propagation of values continues in iterations until all vertex values are stable.

The inputs of propagateAndAggregate are listed in Table 2b. Inputs dir, aggregateF, and updateF are the same as in aggregateNeighborValues. The startVP predicate selects which vertices propagate values in the first iteration. (In the weakly-connected components example, which we will discuss in Section 3.2.1, this predicate always returns true. However, in other algorithms, such as single source shortest paths [28] or approximate betweenness centrality [1], this predicate returns true only for a subset of the vertices.) Similar to the aggregatedValueF input to aggregateNeighborValues, propagatedValueF extracts the relevant value to be propagated from a vertex. In addition, there is a propagateAlongEdgeF function that takes a propagated value val from a vertex, and an edge through which val will be propagated, and computes a possibly modified value to be propagated along the edge.

Input	Description
dir	The direction of the local neighbors whose values will be aggregated
nbrP	A predicate to select which neighbors to aggregate
vP	A predicate to select which vertices to update
aggregatedValueF	A function that takes a neighbor vertex value and returns the relevant part to be aggregated, possibly the entire neighbor value itself
aggregateF	The aggregation function
updateF	A function that takes a vertex and an aggregated value of the neighbors and returns a new value for the vertex

(a) aggregateNeighborValues.

Input	Description
dir	The direction of the local neighbors to propagate values to
startVP	A predicate to select which vertices to start propagating from
propagatedValueF	A function that takes a vertex value and returns the relevant part to be propagated
propagateAlongEdgeF	A function that takes two inputs: (1) the propagated value of a vertex; and (2) an edge value, through which the vertex will propagate its value to a neighbor; and computes the final propagated value along the edge
aggregateF	A function to aggregate propagated values
updateF	A function that takes a vertex and an aggregated value of the propagated values from neighbors and returns a new value for the vertex

(b) propagateAndAggregate.

Figure 2: Inputs to aggregateNeighborValues and propagateAndAggregate.

3.2.1 Examples of Use

As an example use of aggregateNeighborValues, consider executing a fixed number of iterations, 10 say, of the HITS algorithm [24] for ranking web-pages. In HITS, each vertex has a hub and an authority value. In one iteration of the algorithm, vertices first compute the sum of all of their incoming neighbors' hub values to update their authority values. Then, vertices aggregate their outgoing neighbors' authority values to update their hub values. We can express this computation using aggregateNeighborValues as follows:

```
for (i = 0; i < 10; ++i) {
 // Aggregate in-neighbors' hub values to update authorities
 g.aggregateNeighborValues(
  EdgeDirection.In /* direction of neighbors */,
  nbr → true /* which neighbors to aggregate */,
  v → true /* which vertices to update */,
  nbrVal → nbrVal.hub /* relevant neighbor value to aggregate */,
  AggrFnc.SUM /* aggregate neighbors' hub values by summation */,
  (v, aggrHubVal) → { v.val.authority = aggrHubVal; v;})

 // Aggregate out-neighbors' authorities to update hub values
 g.aggregateNeighborValues(
  EdgeDirection.Out,
  nbr → true,
  v → true,
  nbrValue → nbrValue.authority,
  AggrFnc.SUM,
  (v, aggrAuthVal) → { v.val.hub = aggrAuthVal; v;})}
```

As an example use of propagateAndAggregate, consider the weakly-connected components algorithm from [23]. The input is an undirected graph, and vertices store a wccID value, initialized to their own IDs. Initially each vertex v propagates its wccID to all of its neighbors. In iterations, each vertex v updates its wccID to the maximum of its own wccID and the wccID values propagated by its neighbors, then propagates its wccID further if it has changed. At convergence, all vertices with the same wccID value belong to the same connected component. We can express this computation using propagateAndAggregate as follows:

```
g.updateVertices(v → true, v → { v.val.wccID = v.ID}
g.propagateAndAggregate(
  EdgeDirection.Either /* direction of neighbors */,
  v → true /* which vertices to start propagating from */,
  vVal → vVal.wccID /* propagated value */,
  /* do not change propagated value along each edge */
  (propagatedVal, edgeVal) → propagatedVal,
  AggrFnc.MAX /* aggregate wccIDs by taking their max */,
  (v, aggrWccIDVal) → {
   v.val.wccID = AggrFnc.MAX(v.val.wccID, aggrWccIDVal); v;})
```

3.2.2 Implementation

We explain our implementation of aggregateNeighborValues for aggregating in-neighbors of vertices, i.e., input dir is In. Implementations of the other directions are similar. The implementation consists of three steps:

1. Call GraphX's mapReduceTriplets primitive. Recall from Section 2.2.2 that mapReduceTriplets applies a tripletMapF function to the extended edge triplet (eval, srcObj, dstObj) associated with each edge (u,v) in the graph and generates a list of messages. The list of messages for each vertex is then aggregated by the reduceMessagesF function to compute a final MessagesRDD. For tripletMapF we use a function that sends a message to the destination vertex, if the source vertex satisfies the nbrP predicate and the destination vertex satisfies the vP predicate. The message contains the output of applying aggregatedValueF on the source vertex's value. For reduceMessagesF we use aggregateF. It returns a MessagesRDD that contains, for each vertex v that satisfies vP, the aggregation of the values of v's in-neighbors.
2. Join the MessagesRDD with VerticesRDD to obtain VerticesMessagesRDD, which contains for each vertex v its aggregated message m.
3. Apply Spark's map primitive on VerticesMessagesRDD using updateF, which takes v and its aggregated message m and generates a new value for v. Finally, return a new graph using v's new value.

Figure 3 shows the implementation of propagateAndAggregate pictorially on an example. The implementation consists of four steps performed in iterations. The first three steps of each iteration are similar to the three steps of aggregateNeighborValues as above, but with a few differences. We again explain our implementation for the case when the input dir is In.

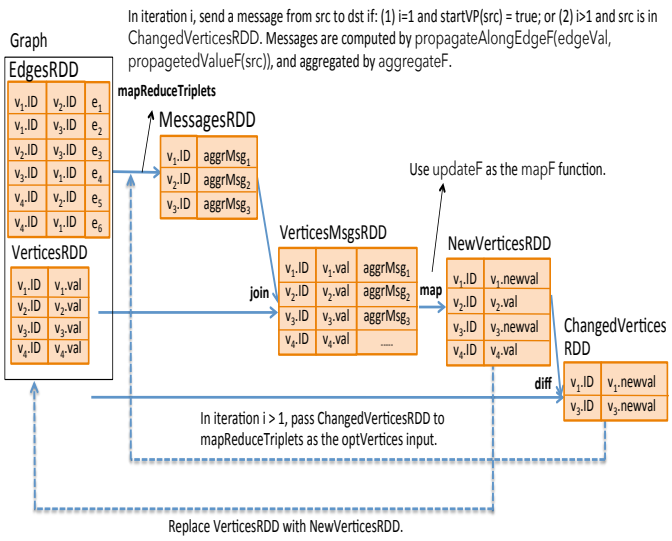

Figure 3: Implementation of `propagateAndAggregate` (to out neighbors) in GraphX.

1. Call `mapReduceTriplets` as in `aggregateNeighborValues`. However, now we send a message from the source `u` to `v` only if `u` satisfies the `startVP` predicate. Moreover, we apply `propagateAlongEdgeF` to the output of `propagatedValueF` and the edge value `eval` in the extended edge triplet to possibly generate a new message.
2. Join the `MessagesRDD` with `VerticesRDD` to obtain `VerticesMessagesRDD`.
3. Apply Spark's `map` primitive on `VerticesMessagesRDD` using `updateF` to generate a new `VerticesRDD`.
4. Call GraphX's `diff` primitive (Section 2.2.2) on the old and new `VerticesRDD`. We refer to the output of `diff` as `ChangedVerticesRDD`.

In subsequent iterations, we repeat the three steps of the first iteration with two differences: (1) we pass `ChangedVertices` to `mapReduceTriplets` as the `optVertices` input (see Section 2.2.2); and (2) instead of sending a message from `u` to `v` if `u` satisfies `startVP`, we send a message if `u` is in `ChangedVertices`. Recall that `mapReduceTriplets` exposes this information in a field of the `srcObj` when `optVertices` input is specified. We continue the iterations until `ChangedVertices` is empty.

3.3 Updating Vertices Using One Other Vertex

In some algorithms, vertices store a pointer to (actually an ID of) one other vertex, not necessarily a neighbor, in a field of their vertex value, and either update their own value using the value of the vertex they point to, or vice-versa. This operation appears commonly in matching algorithms, but it also appears in the minimum spanning tree algorithm from [8, 41], the METIS graph partitioning algorithm [43], and the clustering algorithm from [33]. If `v` stores the ID of `w` in its value, we refer to `v` as the *pointer* vertex and `w` as the *pointed* vertex. Depending on whether the pointer or the pointed vertex is updated, we provide two primitives for this operation: `updateSelfUsingOneOtherVertex` and `updateOneOtherVertexUsingSelf`. The inputs and behavior of the two primitives are very similar, so we specify them only for

Input	Description
`vP`	A predicate to select which vertices will update themselves
`otherVertexIDF`	A function that takes the vertex value, say of `v`, and returns the ID of the other vertex that `v` points to
`relevantPointedVertexValueF`	A function that takes the pointed vertex's value and returns the relevant part of it that will be used in updating the pointer vertex
`updateF`	Takes the vertex `v` and the relevant value of the vertex that `v` points to and returns a new value for `v`

Table 3: Inputs to `updateSelfUsingOneOtherVertex`.

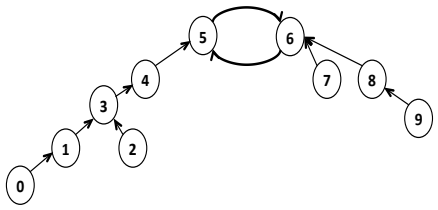

Figure 4: Example of a conjoined-tree.

`updateSelfUsingOneOtherVertex`.

The inputs of `updateSelfUsingOneOtherVertex` are listed in Table 3. The behavior of `updateSelfUsingOneOtherVertex` is to update each vertex `v` for which `vP` evaluates to true in three steps: (1) compute the ID of the vertex `w` that `v` points to by applying `otherVertexIDF` on `v`; (2) apply `relevantPointedVertexValueF` to `w` to extract the relevant value of `w` that will be used in updating `v`; and (3) apply `updateF` on `v` and the output of `relevantPointedVertexValueF` from the second step to compute the new value for `v`. We note that in `updateOneOtherVertexUsingSelf`, multiple vertices might point to and update the same vertex. As a result `updateOneOtherVertexUsingSelf` takes an additional input function `aggregateF`, which aggregates the relevant pointer vertex values, before applying`updateF`.

3.3.1 *Examples of Use*

Consider the minimum spanning tree algorithm from [32, 41]. In each iteration of this algorithm, each vertex `v` points to its minimum-weight neighbor, which is stored in a `pointedVertex` field. As shown in [8], the vertices and their picked neighbors form disjoint subgraphs called *conjoined-trees*: two trees joined by a cycle. Figure 4 shows an example conjoined tree. We refer to the vertex with the smaller ID in the cycle of a conjoined tree `T` as the *root* of `T`, for example vertex 5 in Figure 4. After picking their neighbors, vertices find the root of the conjoined-tree they are part of iteratively as follows. In the first iteration, each vertex `v` discovers whether it is the root by checking whether `v`'s `pointedVertex` `u` points back to `v`, and whether `v`'s ID is smaller than `u`. If both conditions hold, `v` is the root and sets its `pointsAtRoot` field to true. In the later iterations, each vertex `v` copies its `pointedVertex` `u`'s `pointedVertex` and `pointsAtRoot` values to itself until every vertex points to the root. We can express this computation using `updateSelfUsingOneOtherVertex` as follows (we will describe the `aggregateGlobalValue` primitive in Section 4):

```
// discover the root of each conjoined-tree
g.updateSelfUsingOneOtherVertex(
  v → true, /* which vertices to update */
  v → v.val.pointedVertex /* ID of the other vertex */,
```

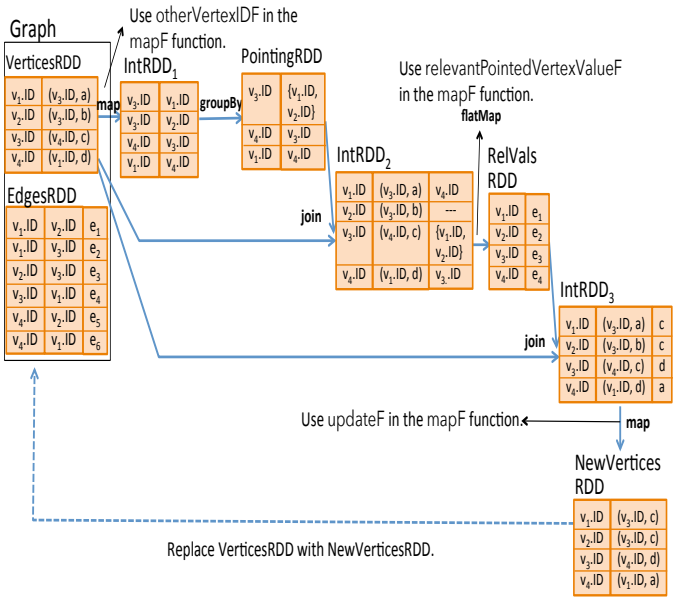

Figure 5: Implementation of `updateSelfUsingOneOtherVertex` in GraphX.

```
// relevant part of the other vertex
otherV → (otherV.val.pointedVertex, otherV.ID),
// updateF: set pointsAtRoot to true if otherV also points
// at v and v's ID is smaller than otherV's ID
(v, (otherVsPointedVertex, otherVID) → {
  if (v.ID == otherVsPointedVertex  v.ID < otherVID) {
    v.val.pointsAtRoot = true}; v;}))

// count the number of vertices that do not point to a root
var numNotPointing = g.aggregateGlobalValue(
  v  → { (v.val.pointsAtRoot) ? 0 : 1), AggrFnc.SUM)
while (numNotPointing > 0) {
  g.updateSelfUsingOneOtherVertex(
    v → !v.val.pointsAtRoot,
    v → v.val.pointedVertex,
    otherV → (otherv.val.pointedVertex, otherV.pointsAtRoot),
    (v, (otherVsPointedVertex, otherVPointsAtRoot) → {
      v.val.pointedVertex = otherVsPointedVertex;
      v.val.pointsAtRoot = otherVPointsAtRoot; v;}}))
  numNotPointing = g.aggregateGlobalValue(
    v  → { (v.val.pointsAtRoot) ? 0 : 1), AggrFnc.SUM)}
```

3.3.2 *Implementation*

Figure 5 shows the implementation of `updateSelfUsingOneOtherVertex` pictorially on an example, which we also use to guide our explanation. (We omit the description of `updateAnotherVertexUsingSelf`, which is similar.) In the figure, vertex values consist of two fields, the first of which stores the ID of the pointed vertex, e.g., v_1 points to v_3, and v_4 points to v_1. In our example, if `u` points to `v`, `u` copies over the second field of `v` to itself.

1. We apply `otherVertexIDF` on each vertex by calling Spark's `map` primitive on `VerticesRDD`, generating an intermediate RDD containing (`pointedID`, `pointerID`) tuples. Since v_1 and v_2 point at v_3 in the figure, two of the tuples in this RDD are (v_3.ID, v_1.ID) and (v_3.ID, v_2.ID). We then group these tuples by `pointedID` to obtain `PointingRDD`, which in our example contains (v_3.ID, {v_1.ID, v_2.ID}).
2. We join each pointed vertex `v`'s value with the IDs of the vertices that point to `v`, by joining `PointingRDD` with the original `VerticesRDD`. We obtain in our example (v_3.ID, (v_4.ID, `c`), {v_1.ID, v_2.ID}), where the second field is v_3's value. We then use Spark's `flatMap` primitive[2] to output a set of (`pointerID`, `relevantPointedVValue`) tuples. In our example, we output (v_1.ID, `c`) and (v_2.ID, `c`), where `c` is the relevant value from v_3. We call this RDD `RelevantValuesRDD`.
3. We join each vertex `v`'s value with the relevant value from the vertex `v` points to (by joining `RelevantValuesRDD` with the original `VerticesRDD`). In our example, we obtain (v_1.ID, (v_3.ID, `a`), `c`) and (v_2.ID, (v_3.ID, `b`), `c`). Finally, using Spark's `map` primitive, we apply `updateF` on the join tuples to obtain new values for each vertex. We get (v_1.ID, (v_3.ID, `c`)) and (v_1.ID, (v_3.ID, `c`)), effectively copying over the `c` from v_3's value to v_1 and v_2.

4. TOPOLOGY MODIFICATIONS

We describe two primitives that change the topology of the graph, either by removing certain vertices and edges, or by merging multiple vertices together to form *supervertices* (explained below).

4.1 Filtering

Filtering operations remove certain vertices and/or edges from the graph. These operations appear especially in algorithms that iteratively find partial solutions on a subset of the vertices or edges of the graph, remove them, then continue until there are no vertices and edges left. We provide two primitives for filtering vertices and one primitive for filtering edges.

- `filterVertices`: Takes as input a predicate `vP` that accepts a vertex value and an ID. The behavior is to remove all vertices from the graph for which `vP` evaluates to false and all edges that are incident on such vertices.
- `filterVerticesUsingLocalEdges`: Same as `filterVertices` except predicate `vP` takes as additional input the local edges of the vertex in a user-specified direction `dir`.
- `filterEdges`: Takes as input a predicate `eP` that accepts an edge value and a source and destination vertex ID and values. The behavior is to remove from the graph all edges for which `eP` evaluates to false.

4.1.1 *Examples of Use*

Consider the approximate maximum weight matching algorithm from [37]. In each iteration of this algorithm, vertices that are unmatched pick their maximum-weight edge neighbor and store it in a `pickedNbrID` field. Then, if two vertices have picked each other, i.e., they have successfully matched, they set their `isMatched` field to true and are removed from the graph. We can express the removal of matched vertices from the graph using the `filterVertices` primitive simply as:

```
g.filterVertices(v → v.val.isMatched == false)
```

For an example use of `filterEdges`, consider Luby's maximal independent set algorithm from [26]. In this algorithm each vertex sets a label value as `InSet`, `NotInSet`, or `Unknown`. In each iteration, some of the `Unknown` vertices are marked as `InSet`, and some as `NotInSet`, while the labels of others remain

[2] `flatMap` is a generalized version of the `map` primitive that can output multiple elements instead of just one.

Input	Description
`supervertexIDF`	A function that takes the vertex value, say of `v`, and returns the ID of `v`'s supervertex
`mergeVertexValuesF`	A function that takes a set of vertex values and returns a single merged value for the supervertex
`mergeEdgeValuesF`	A function that takes a set of edge values for edges between the same pair of supervertices and returns a single merged value for the edge

Table 4: Inputs to `formSupervertices`.

unchanged. At the end of each iteration, the algorithm removes all edges incident on `InSet` and `NotInSet` vertices, i.e., keeps only the edges between `Unknown` vertices. This operation is expressed using `filterEdges` as:

```
g.filterEdges((edgeVal, srcV, dstV) →
  (srcV.val.setType == Unknown) && (dstV.val.setType == Unknown))
```

4.1.2 Implementation

GraphX's core primitive `filter` can already perform the functionalities of `filterVertices` and `filterEdges`, since `filter` takes both an edge and vertex predicate (Section 2.2.2). We nonetheless added new `filterVertices` and `filterEdges` primitives for convenience and clarity: in all of our algorithms either vertices or edges are filtered, never both. `filterVertices` simply calls `filter` with `vP` and a dummy edge predicate that always evaluates to true; similarly for `filterEdges` with `eP` and a dummy vertex predicate.

For `filterVerticesUsingLocalEdges`, we first call GraphX's `collectEdges` primitive with the user-specified direction to obtain an extended `VertexRDD` that contains a list of edges for each vertex. Then we filter this extended `VertexRDD` with the given `vP` to compute a new `VerticesRDD`.

4.2 Forming Supervertices

Another operation that modifies the topology of the graph is to merge groups of vertices into *supervertices*. This operation appears in Boruvka's minimum spanning tree algorithm [32] and in some graph partitioning and clustering algorithms, such as the METIS partitioning algorithm [29, 43] and the clustering algorithm from [33]. In these algorithms, after some computation every vertex identifies a supervertex (possibly itself) that it will merge into. Then:

- All vertices and their values that belong to the same supervertex are merged into a single vertex. How the vertex values are merged is specified by an input function.
- Consider an edge (`u`,`v`) and assume that vertices `u` and `v` are merged into supervertices `s1` and `s2`, respectively. If `s1` = `s2`, then (`u`,`v`) is removed from the graph. Otherwise, (`u`,`v`) becomes an edge between `s1` and `s2`. If there are multiple edges between `s1` and `s2`, then edges are merged. How the edge values are merged is specified by an input function.

The inputs of our `formSupervertices` primitive are listed in Table 4.

4.2.1 Examples of Use

As an example, consider Boruvka's minimum spanning tree algorithm, which we discussed in Section 3.3.1. In each iteration, vertices discover the root of the conjoined-tree they are part of and store its ID in a `pointedVertex` value. The algorithm uses these values to merge all vertices in each conjoined-tree into a single supervertex. For this algorithm, function `mergeVertexValuesF` simply returns a new empty vertex value because the algorithm does not need to merge the values of vertices that form the supervertex. However, edges are weighted; function `mergeEdgeValuesF` takes minimum of the edge values. We can express this computation using `formSupervertices` as:

```
g.formSupervertices(
  v → v.val.pointedVertex /* ID of the supervertex */
  vVals → new EmptyMSTVertexValue(), /* mergeVertexF */
  eVals → AggrFnc.MIN(eVals) /* mergeEdgeF */)
```

4.2.2 Implementation

The implementation of `formSupervertices` generates a new graph from scratch by first computing a new `VerticesRDD` and then a new `EdgesRDD`, as follows.

- We `map` each vertex in the old `VerticesRDD` to a (`supervertexID`, `vertexValue`) tuple. We extract the supervertex ID of each vertex by calling `supervertexIDF` on the vertex. We then use Spark's `reduceByKey` primitive to group the (`supervertexID`, `vertexValue`) tuples by `supervertexID` and aggregate them using `mergeVertexValuesF`, which yields a new `VerticesRDD`.
- For computing the new `EdgesRDD` we use GraphX's `getTriplets` primitive to get the `TripletsRDD`. Recall from Section 2.2.2 that `TripletsRDD` associates each edge `e`=(`u`,`v`) with the IDs and values of `u` and `v`. We then apply Spark's `map` primitive on `TripletsRDD`. If `u` and `v` belong to different supervertices, `map` outputs a tuple of the form ((`newSrcID`, `newDstID`), `eval`), where `newSrcID` and `newDstID` are the supervertex IDs of `u` and `v`, respectively, and `eval` is the value of `e`. Otherwise, if `u` and `v` belong to the same supervertex, `map` does not output any tuples. Finally, we use Spark's `reduceByKey` primitive to group the resulting tuples by (`newSrcID`, `newDstID`) pairs and aggregate the associated edge values of each pair using `mergeEdgeValuesF`, which yields a new `EdgesRDD`.

5. GLOBAL AGGREGATIONS

Many graph algorithms need to compute a global value over the vertices of the graph, such as counting the number of vertices with a particular value, or finding the maximum vertex value. Specifically, each vertex emits a value, and the values are aggregated in some fashion to produce a single value. These computations can thus be seen as special MapReduce computations with a single reducer.

We abstract global aggregation in our `aggregateGlobalValue` and `aggregateGlobalValueUsingLocalEdges` primitives. `aggregateGlobalValue` takes two inputs:

- `mapF`: A function that takes a vertex and produces a value.
- `reduceF`: A function that takes a pair of values and aggregates them into a single value.

Notice that function `reduceF` combines a pair of values, rather than the set of all emitted values. Function `reduceF` is first applied to a pair of mapped values to produce a single value, the function is then applied to the result value with another mapped value, and so on until the entire set of mapped values has been processed to produce the final aggregated value. The order of application is unpredictable, so for correctness of the aggregation operation, `reduceF` should be commutative and associative. Our requirement of a pairwise `reduceF` function is primarily for efficiency: `mapF` is applied to every vertex in the graph, so the set of values to be processed by `reduceF` has size equal to the number of vertices of the graph. Applying `reduceF` to the entire set at once would require collecting all of the values in a single machine. Also, the most efficient implementation of global aggregation in GraphX is

to use Spark's `map` and `reduce` primitives directly (Section 5.2), and Spark likewise limits its `reduce` primitive to pairwise operations for efficiency [45].

Function `aggregateGlobalValueUsingLocalEdges` takes an additional `dir` input, and in this case `mapF` additionally takes the set of edges incident to the vertex in the specified direction.

We note that a frequently used special case of a global aggregation operation is to pick a random vertex from the graph, which we expose as a separate `pickRandomVertex` primitive in GraphX.

5.1 Examples of Use

One example use of `aggregateGlobalValue` is to detect termination of the root finding phase of the minimum spanning tree algorithm. Recall from Section 3.3.1 that the algorithm iterates in this phase until all vertices find their roots. To compute whether all vertices have found their roots, the algorithm counts the number of vertices whose `pointsAtRoot` value is false, using primitive `aggregateGlobalValue` as follows:

```
numNotPointing = g.aggregateGlobalValue(
  v → { (!v.val.pointsAtRoot) ? 1 : 0}, AggrFnc.SUM)
```

Another example is the approximate betweenness-centrality algorithm from [1], which performs a breadth-first search (BFS) from a source vertex and labels each vertex with its level in the BFS tree. Then, the algorithm computes the maximum depth of the tree using primitive `aggregateGlobalValue` as follows:

```
maxDepth = g.aggregateGlobalValue(v → v.val.level, AggrFnc.MAX)
```

5.2 Implementation

The implementation of `aggregateGlobalValue` simply applies Spark's `map` and `reduce` primitives on the `VerticesRDD` with `mapF` and `reduceF` respectively. For `aggregateGlobalValueUsingLocalEdges`, we call GraphX's `collectNeighbors` with the user-specified direction before applying `flatMap` and `reduce`.

6. SUPPLEMENTING GRAPH PRIMITIVES WITH DATA PRIMITIVES

In Sections 3-5 we described our HelP primitives that abstract several commonly-appearing operations in distributed graph computations. However, for certain algorithms, our primitives alone may not be sufficient or the most suitable way of implementing the algorithm. Sometimes, using record-oriented data primitives instead of or in addition to our graph primitives can be beneficial. In the next two subsections, we show example algorithms based on finding subgraphs, such as triangles, rectangles, or *k-trusses*, that benefit from this approach.

We note that GraphX is currently the only system that exposes both graph primitives and data primitives in the same interface. As observed in the original GraphX paper [50], integration of graph and data primitives enables programmers to express the three components of a typical large-scale graph computation pipeline, through one interface: (1) graph construction, i.e., extract, transform, and load (ETL), using data primitives; (2) implementing a graph algorithm, using graph primitives; and (3) analyzing the output of the algorithm and/or feeding the output to another pipeline, using data primitives. We show that combining graph and data primitives can be helpful not only in providing a uniform interface across (1), (2), and (3), but also for implementing algorithms within step (2).

6.1 Using Data Primitives Instead of Graph Primitives

With the exception of `filterEdges`, our library consists of vertex-centric graph primitives: some computation is performed for each vertex, such as an update, a filter, or emitting a value to be aggregated. Some subgraph finding algorithms can easily be expressed using our vertex-centric primitives, such as finding the *k-core* of an undirected graph: a subgraph in which each vertex has at least k edges. We can find the k-core of a graph by using our `filterVerticesUsingLocalEdges` primitive iteratively, removing vertices with fewer than k edges:

```
var previousNumVertices = Long.MaxValue
while (previousNumVertices != g.numVertices) {
  previousNumVertices = g.numVertices
  g.filterVerticesUsingLocalEdges(EdgeDirection.Either,
    (vId, vval, edges) → edges.get.size >= k )}
```

In contrast, some subgraph finding algorithms are "edge-centric". They search for particular edge patterns, such as three or four edge cycles when finding triangles or rectangles, or they require computing a value for each edge, such as finding the *k-truss* of a graph: a subgraph in which each edge participates in at least $k-2$ triangles. Such edge-centric algorithms cannot be expressed easily with our primitives. We take finding triangles as a simple example.

The standard approach to finding triangles [22, 38, 46] takes two steps: (1) compute open-triads, i.e., pairs of edges of the form (`u`,`v`) (`v`,`w`); then (2) search for the closing edges (`u`,`w`). If we used our graph primitives, we would likely perform this computation as follows (to simplify the code, we ignore overcounting of the triangles):

```
// copy local adjacency lists to local value
g.updateVerticesUsingLocalEdges(
  EdgeDirection.Either,
  (vid, vval, edges) => {
    val localEdges = edges.map(edge => {edge.otherVertexId(vid)});
    new Value(localEdges, null /* no nbrs' adjacency lists yet */)})
// copy each neighbor's adjacency lists to local value
g.aggregateNeighborValues(
  EdgeDirection.Either,
  (vId, vval) => true /* nbrP */, (vId, vval) => true /* vP */,
  (nbrId, nbrdata) => Array((nbrId, nbrdata.adjList)),
  // concatenate all (nbrId, nbrAdjList)
  (nbrAdjList1, nbrAdj2) => nbrAdjList1:::nbrAdjList2,
  (vId, vval, nbrsAdjLists) => newValue(vval.adjList, nbrsAdjLists))
// find the triangles each vertex is part of
g.updateVertices((vId, vval) => true,
  (vid, vval) => {
    val nbrsSet = vval._1.toSet
    var triangles = List.empty;
    for ((nbrId, nbrAdjList) <− vval._2) {
    for (otherNbrId <− nbrAdjList) {
      if (nbrsSet.contains(otherNbrId))
        triangles.append((vId, nbrId, otherNbrId)}}
    return triangles})
```

First, each vertex `u` copies its adjacency list to its value using `updateVerticesUsingLocalEdges`. Then, each vertex `u` copies over the adjacency list of each of its neighbors `v` using `aggregateNeighborValues`. The aggregation operation in the above code appends the (`nbrId`, `nbrAdjList`) pairs into an array. At this point, `u`'s value contains its own adjacency list and an array containing the adjacency lists of its neighbors. Assume `v` is a neighbor of `u`, and `v`'s adjacency list contains {`w`, `z`}. Then `u` can infer its open triads with `v`, namely (`u`,`v`) (`v`,`w`) and (`u`,`v`) (`v`,`z`). Finally, `u` finds all the triangles it is part of by searching the closing edges of its open triads, such as (`u`,`w`) and (`u`,`z`), in its own adjacency list using the `updateVertices` primitive. No-

tice that the UDFs and vertex values we use inside our primitives are quite complex, and they get even more complex when finding larger subgraph shapes.

A simpler approach is to view the edges as a table with schema (src, dst), and perform a three-way join on this table using data primitives. Suppose we store the edges table in an RDD called edges, and that an edge (u,v) exists in edges both as (u,v) and (v,u). We again ignore overcounting in the following code:

```
// openTriads contains (u, v, w) tuples; closing edge is (u, w)
val openTriads = edges.join(edges)
// openTriadsClosingEdgesFirstColumn contains ((u, w), v) tuples
val openTriadsClosingEdgesFirstColumn = openTriads.map(
  triad−> ((triad._1, triad._3), triad._2))
val triangles = openTriadsClosingEdgesFirstColumnjoin(edges)
```

We first self-join edges to compute the openTriads table. We then join openTriads with edges to close the triangles. Note that Spark only allows joining tables on the first column, so in between the two join operations we rearrange the columns of openTriads. Finding larger subgraphs can similarly be expressed as a multi-way join using data primitives.

6.2 Using Data Primitives Together With Graph Primitives

Now we show the benefits of directly combining data primitives and graph primitives. Suppose we want to compute the *clustering coefficient* [49] of each vertex v: the ratio of the actual number of triangles v participates in to the maximum number of triangles v can participate in. The maximum is the number of open triads v is in, or $\frac{k(k-1)}{2}$ where k is the degree of v. We can perform this computation using data and graph primitives together as follows: (1) compute the number of triangles for each vertex from triangles by using Spark's flatMap and reduce primitives, yielding a vertexIDNumTriangles RDD; (2) construct a graph object, using edges as EdgesRDD, and vertexIDNumTriangles as VerticesRDD; (3) compute the clustering coefficient of each vertex using updateVerticesUsingLocalEdges. The code is:

```
val vertexIDNumTriangles = triangles.flatMap(tri →
  Iterator((tri._1, 1), (tri._2, 1), (tri._3, 1))
  .reduceByKey(AggrFn.SUM)
val g = Graph(vertexIDNumTriangles, edges)
// vval contains the number of triangles, we return vval/(k(k-1)/2)
g.updateVerticesUsingLocalEdges(v → true,
  (vId, data, edges) => vval/((edges.length∗edges.lengh−1)/2))
```

Continuing with our example, suppose we next want to compute the *global clustering coefficient* of the graph: average clustering coefficient of the vertices. We can perform this computation with aggregateGlobalValue:

```
val sumCoeff = g.aggregateGlobalValue(
  (vId, vval) → v.val,  AggrFn.SUM)
val globalClusteringCoeff = sumCoeff / g.numVertices
```

If we wanted to compute the global clustering coefficient without computing local coefficients explicitly, instead of updateVerticesUsingLocalEdges, we could use aggregateGlobalValueUsingLocalEdges directly.

7. RELATED WORK

No other work we know of proposes and implements high-level primitives for distributed graph computations. Related work divides roughly into four categories: existing distributed systems for large-scale graph computations, high-level languages for distributed data analysis, domain-specific languages for graph computations, and MPI-based graph libraries.

7.1 APIs of Existing Distributed Systems

We first consider existing distributed systems for large-scale graph computations, reviewing specifically the different APIs exposed by these systems. The APIs of Spark (version 0.8.1) and the earlier version of GraphX on which we built our primitives were discussed in Section 2. APIs of existing distributed systems fall into two categories: vertex-centric and MapReduce-based.

7.1.1 Vertex-centric APIs

The APIs of existing synchronous graph processing systems, such as Google's Pregel and its open source versions, such as Giraph [13] and GPS [40], consist of a single vertex.compute() function. In these systems, computation is broken down into synchronous iterations called *supersteps*, and in each superstep the vertex.compute() function gets called on each vertex. Inside vertex.compute(), vertices receive messages from the previous superstep, update their local states, send messages to other vertices, and add or remove edges and vertices in the graph. Programmers express the vertex-centric logic of their algorithms by implementing vertex.compute(). PowerGraph [14] is a distributed message-passing system that can execute computations both synchronously and asynchronously, in the latter case without any clearly defined supersteps. The PowerGraph API is a set of three vertex-centric functions: gather(), apply(), and scatter(). Using these functions, programmers define the messaging behavior of the vertices and how to update vertex values using received messages.

As discussed in the introduction, vertex-centric functions can be too low-level for certain operations, such as forming supervertices or updating vertices using values from another vertex, resulting in complex vertex values and messaging between vertices. Also, as observed in [18, 20, 40], the APIs of these systems do not offer mechanisms for controlling the flow of programs, which makes it challenging to write algorithms composed of multiple operations: workarounds using global data structures are needed to control the flow from one operation to the next. In contrast, multiple operations in our primitives are expressed naturally by writing the primitives one after another.

7.1.2 MapReduce-based APIs

MapReduce [9], Hadoop [17], and systems that extend Hadoop can also be used for for graph computations. To perform graph processing in these systems, the graph structure and the vertex and edge values are encoded in distributed files or in a distributed database. In basic MapReduce or Hadoop, programmers implement one or more pairs of map() and reduce() functions to simulate each graph operation in their algorithms. For computations that iterate until vertex values converge, programmers implement additional map() and reduce() functions to check convergence criteria. The APIs of iterative extensions of Hadoop, such as HaLoop [5] and iMapReduce [52], provide special additional functions for checking convergence criteria.

MapReduce-based APIs have similar shortcomings to vertex-centric ones, and in some sense are even lower level, since they are not part of a graph-specific abstraction. Likewise they do not provide mechanisms for controlling the flow of operations, so an extra script needs to be written to assemble all of the map() and reduce() functions into a single program.

7.2 Higher-level Data Analysis Languages

Another way to implement graph algorithms for execution on MapReduce or Hadoop is through a higher-level language that compiles to these systems, such as Pig Latin [34], Hive [47], or Scalding [42]. When using these languages, the graph structure and ver-

tex and edge values again are encoded in distributed storage. However, instead of writing `map()` and `reduce()` functions, programmers use higher-level constructs. In Pig Latin and Scalding, just as in Spark, programmers use data primitives such as `filter`, `group`, and `join`; in Hive programmers use a declarative SQL-like language, including constructs such as `select`, `update`, and `groupBy`. Shark [11] is similar to Hive, exposing a SQL layer over Spark that can be used to implement graph algorithms that execute on Spark.

Unlike our HelP primitives, which are graph-specific and operate on a graph abstraction, these languages expose relational data primitives and operate on tables consisting of tuples or key-value pairs. Thus, graph operations that we abstract in a single primitive can require a large number of data primitives. For example, if we were to bypass GraphX and use only Spark's primitives, programmers would need to use eight data primitives for `propagateAndAggregate` and six for `updateSelfUsingAnotherVertex`.

7.3 Domain-Specific Languages

Recently, Green-Marl [20], a domain-specific graph language, was introduced specifically for implementing distributed and parallel graph algorithms. Green-Marl is an imperative language with graph-specific constructs, such as `Graph` and `Node` types to represent a graph, and `InBFS` and `InDFS` constructs to express breadth-first and depth-first graph traversals. Programs are written as if the graph is stored in a single machine, but compile to different parallel or distributed backends. Currently Green-Marl compiles to parallel C++ code and to two distributed graph-processing systems: Giraph and GPS. Only a subset of Green-Marl, called *Pregel-canonical* programs, can be compiled to run on GPS or Giraph [21], so programmers need to be aware of which code patterns and constructs violate the restrictions of the Green-Marl compiler. In contrast, all of our primitives abstract distributable operations.

7.4 MPI-based Graph Libraries

Finally, there are several graph libraries based on *Message Passing Interface* (MPI). MPI is a standard interface for building parallel and distributed message-passing programs, which can be used for graph processing. All MPI-based systems and libraries can compile to and execute on distributed architectures that have an MPI implementation, such as Open MPI [35] or MPICH [30].

PBGL [16] exposes a set of templates for distributed data structures, such as adjacency lists, and also a library of existing algorithms, such as single-source shortest paths and weakly-connected components, which the programmer can invoke directly. However, the configuration of these templates requires programmers to code very low-level details, such as how to distribute the vertices and adjacency lists, and how to traverse them during the execution. In addition, programmers need to write manual code to describe which processors send each other messages during execution.

Combinatorial BLAS (CBLAS) [6] is another MPI-based library that exposes a set of linear algebra primitives, such as primitives that multiply a sparse matrix with a vector, or multiply two sparse matrices. Many graph algorithms can be implemented using linear algebra primitives. However, programmers need expertise in the linear algebra abstraction of graphs, as opposed to the classic graph abstraction consisting of nodes, edges, and values on nodes and edges, which our primitives are designed for. KDT [27] is a library of graph algorithms built on top of Combinatorial BLAS, which can be used to perform graph analysis tasks, but it does not have an API to develop custom algorithms.

8. CONCLUSIONS AND FUTURE WORK

We presented HelP, a set of high-level graph processing primitives, which we believe abstract the most commonly appearing operations in distributed graph computations. We described the implementations of our primitives on the GraphX system, and the implementations of many graph algorithms using our primitives. Our experience has been that implementing algorithms using the HelP primitives is more intuitive and much faster than using the APIs of existing distributed graph systems (discussed in Section 7). We also illustrated the benefits of using data primitives, such as joins and grouping, together with the HelP primitives when implementing distributed versions of some graph algorithms.

We outline two broad directions for future work.

- **Extending Existing High-level Languages:** Some of the existing high-level data analysis languages, such as Pig Latin and Scalding, could be extended first with graph-specific data structures, and then with our HelP primitives. For example, in hypothetically extended Pig Latin, an algorithm that first loads a web graph from raw files and then finds its weakly-connected components could be expressed by a program like:

  ```
  vid_links = LOAD '/raw_links.txt' AS (fromId, toId)
  g = LOAD GRAPH vid_links WITH VERTEX VALUE wccId
  g = UPDATE VERTICES wccId = ID
  g = PROPAGATE AND AGGREGATE wccId FROM ALL
      USING MAX IN g
  STORE graph INTO 'graph_wcc_output.txt'.
  ```

  Primitives that already exist in Pig Latin are in normal font and additional HelP primitives in bold. Scalding could similarly be extended to support our primitives. Scalding is a Scala API on top of Cascading [7], a framework for writing Hadoop workflows. Since GraphX and Scalding are both written in Scala, if we extended Scalding with our primitives, the weakly-connected components pipeline would look similar to our GraphX code from Section 3.2.1. Graph primitives in Pig Latin and Scalding could compile directly to Hadoop, or to systems that run on Hadoop, such as Giraph [13].
- **Optimizations:** The current implementations of our primitives in GraphX assume that each primitive executes in isolation. However, in general each primitive is called as part of a larger program that may contain multiple primitives. Analyzing the entire program could enable the system to perform certain optimizations before execution. As a very simple example, consider an algorithm that calls `updateVerticesUsingLocalEdges` twice in a row, first with `updateF1` and then with `updateF2`. In our current implementation, we call `collectEdges` and perform a join operation of the `VerticesRDD` and `EdgesRDD`, then apply `updateF1` on each joined row to generate a new `VerticesRDD`, then repeat with `updateF2`. Since `updateVerticesUsingLocalEdges` does not change the local neighborhood of vertices, one can instead call `collectEdges` only once, then execute `updateF1` and `updateF2` consecutively on each joined row. There are multiple approaches to analyzing an entire program, e.g., we could use static program analysis in advance, or extend the API with an `execute()` primitive that first optimizes and then executes.

In addition to the future topics outlined above, another direction is to consider asynchronous graph computations, where it may be possible to similarly identify and implement commonly-appearing operations as high-level primitives.

9. ACKNOWLEDGEMENTS

We thank members of the GraphX team—Dan Crankshaw, Ankur Dave, Michael Franklin, Joseph Gonzalez, Akihiro Matsukawa, Ion Stoica, and Reynold Xin—for useful discussions and feedback on

the primitives themselves, and the implementations of the primitives and our algorithms in GraphX.

10. REFERENCES

[1] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating Betweenness Centrality. In *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph*, 2007.

[2] Bagel Programming Guide. https://github.com/mesos/spark/wiki/Bagel-Programming-Guide/.

[3] B. Bollobas. *Modern Graph Theory*. Springer, 1998.

[4] S. Brin and L. Page. The Anatomy of Large-Scale Hypertextual Web Search Engine. In *Proceedings of the International Conference on The World Wide Web*, pages 107–117, 1998.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *Proceedings of the International Conference on Very Large Databases*, 2010.

[6] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *International Journal of High Performance Computing Applications*, 25(4), 2011.

[7] Cascading. http://www.cascading.org/.

[8] S. Chung and A. Condon. Parallel Implementation of Boruvka's Minimum Spanning Tree Algorithm. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation*, 2004.

[10] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *International Symposium on High Performance Distributed Computing*, 2010.

[11] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[12] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2), 2009.

[13] Apache Incubator Giraph. http://incubator.apache.org/giraph/.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[15] GraphX Source Code. https://github.com/amplab/graphx.

[16] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *In Parallel Object-Oriented Scientific Computing*, 2005.

[17] Apache Hadoop. http://hadoop.apache.org/.

[18] P. Haller and H. Miller. Parallelizing Machine Learning-Functionally: A Framework and Abstractions for Parallel Graph Processing. In *2nd Annual Scala Workshop*, 2011.

[19] Help Primitives. https://github.com/semihsalihoglu/incubator-spark/tree/Help_Primitives_1.

[20] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[21] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *International Symposium on Code Generation and Optimization*, 2014.

[22] C. Jonathan. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11(4), 2009.

[23] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System – Implementation and Observations. In *Proceedings of the IEEE International Conference on Data Mining*, 2009.

[24] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46, 1999.

[25] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[26] M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4), 1986.

[27] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-source Toolbox for Scalable Complex Graph Analysis. In *Proceedings of the SIAM Conference on Data Mining*, 2012.

[28] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2011.

[29] METIS Graph Partition Library. http://exoplanet.eu/catalog.php.

[30] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[31] A. A. Nanavati, S. Gurumurthy, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea, and A. Joshi. On the Structural Properties of Massive Telecom Call Graphs: Findings and Implications. In *International Conference on Information and Knowledge Management*, 2006.

[32] J. Nešetřil, E. Milková, and H. Nešetřilová. "Otakar Borůvka on Minimum Spanning Tree Problem Translation of Both the 1926 Papers, Comments, History". *"Discrete Mathematics "*, 233(1–3), 2001.

[33] S. Oliveira and S. C. Seok. Multilevel Approaches for Large-scale Proteomic Networks. *International Journal of Computer Mathematics*, 84(5), 2007.

[34] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[35] Open MPI. http://www.open-mpi.org/.

[36] O. Phelan, K. McCarthy, and B. Smyth. Using Twitter to Recommend Real-time Topical News. In *Proceedings of the ACM Conference on Recommender Systems*, 2009.

[37] R. Preis. Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *Proceedings of the Symposium On Theoretical Aspects of Computer Science*, 1998.

[38] L. Quick, P. Wilkinson, and D. Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining*, 2012.

[39] J. Rosenberg. Some Misconceptions About Lines of Code. In *Proceedings of the International Symposium on Software Metrics*, 1997.

[40] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, 2013.

[41] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-like Systems. In *Proceedings of the International Conference on Very Large Databases*, 2014.

[42] Scalding Github Repository. https://github.com/twitter/scalding.

[43] K. Schloegel, G. Karypis, and V. Kumar. Parallel Static and Dynamic Multi-constraint Graph Partitioning. *Concurrency and Computation: Practice and Experience*, 14(3), 2002.

[44] Apache Spark. http://spark.incubator.apache.org/.

[45] Spark Programming Guide 0.8.1. https://spark.incubator.apache.org/docs/0.8.1/scala-programming-guide.html.

[46] S. Suri and S. Vassilvitskii. Counting Triangles and the Curse of the Last Reducer. In *Proceedings of the International Conference on World Wide Web*, 2011.

[47] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2), 2009.

[48] S. Vadapalli, S. R. Valluri, and K. Karlapalem. A Simple Yet Effective Data Clustering Algorithm. In *Proceedings of the IEEE International Conference on Data Mining*, 2006.

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

[50] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, 2013.

[51] Zaharia, M. and Chowdhury, M. and Franklin, M. J. and Shenker, S. and Stoica, I. Spark: Cluster Computing with Working Sets. In *2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[52] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapreduce: A distributed computing framework for iterative computation. *DataCloud*, 2011.

[53] C. Zhong, D. Miao, and R. Wang. A Graph-theoretical Clustering Method Based on Two Rounds of Minimum Spanning Trees. *Pattern Recognition*, 43(3), 2010.