GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has lead to the development of new *graph-parallel* systems (*e.g.,* Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

AIQL 中提到的timely, 这里的交互式很符合要求

We introduce GraphX, which combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. We leverage new ideas in distributed graph representation to efficiently distribute graphs as tabular data-structures. Similarly, we leverage advances in data-flow systems to exploit in-memory computation and fault-tolerance. We provide powerful new operations to simplify graph construction and transformation. Using these primitives we implement the PowerGraph and Pregel abstractions in less than 20 lines of code. Finally, by exploiting the Scala foundation of Spark, we enable users to interactively load, transform, and compute on massive graphs.

1. INTRODUCTION

From social networks to advertising and the web, big graphs can be found in a wide range of important applications. By modeling the relationships between users, products, and ideas, graphs allow us to identify communities, target advertising, and decipher the meaning of documents. In response to the growing size and importance of graph data, a range of new large-scale distributed *graph-parallel* frameworks (*e.g.,* Pregel [9], PowerGraph [6], and others [5, 3, 11]) have emerged. Each framework introduces a new programming abstraction that allows users to compactly describe graph algorithms (*e.g.,* PageRank, Belief Propagation, ...) and a corresponding runtime engine that efficiently executes these algorithms on multicore and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few frameworks have built-in support for interactive graph computation.

Alternatively *data-parallel* systems like MapReduce and Spark [12] are designed for scalable data processing and are well suited to the task of graph construction (ETL). By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. More recent systems like Spark even enable interactive data processing. However, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement that does not exploit the graph structure.

To address these challenges we introduce GraphX, a graph computation system which runs in the Spark data-parallel framework. GraphX extends Spark's Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG), which associates records with vertices and edges in a graph and provides a collection of expressive computational primitives. Using these primitives, we implement PowerGraph [6] and Pregel [9], two of the most widely used graph-processing frameworks. In addition, we provide new operations to view, filter, and transform graphs, that substantially simplify the process of graph ETL and analysis. The GraphX RDG leverages advances in distributed graph representation and exploits the graph structure to minimize communication and storage overhead. Our primary contributions are:

1. a new graph abstraction called the Resilient Distributed Graph (RDG) that supports a wide range of graph operations on top of a fault-tolerant, interactive platform.
2. a tabular representation of the efficient vertex-cut partitioning described by [6] and data-parallel partitioning heuristics.

启发式到底怎么理解？

3. implementations of the PowerGraph and Pregel graph-parallel frameworks using RDGs in less than 20 lines of code each.
4. preliminary performance comparisons between a popular data-parallel and graph-parallel frameworks running PageRank on a large real-world graph.

Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013, New York, New York, USA.

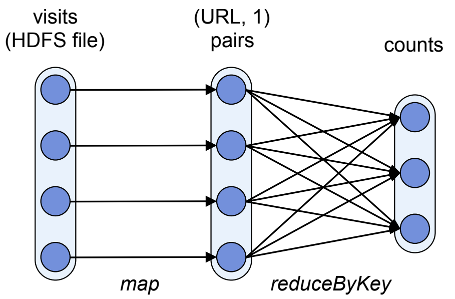


Figure 1: **Lineage graph** for the RDDs in our Spark example. The oblong ovals represent RDDs, while circles show partitions within a dataset. Lineage is tracked at the granularity of partitions.

2. SPARK AND RDDS

Spark is a MapReduce-like data-parallel computation engine open-sourced by UC Berkeley. Spark has several features that differentiate it from traditional MapReduce engines such as Hadoop:

1. It supports general computation DAGs beyond the two-stage MapReduce topology.
2. The execution engine can tolerate the loss of any set of worker nodes and can automatically re-execute lost tasks. This is explained further below.
3. It provides an in-memory storage abstraction called Resilient Distributed Datasets (RDDs) that lets applications keep data in memory, and automatically reconstructs lost partitions upon failures.
4. It integrates with the Scala command-line shell to enable interactive operations on RDDs.
5. RDDs allow the optional specification of a data partitioner, and the execution engine can exploit this to co-partition RDDs and co-schedule tasks to avoid data movement.

可供我的
写作参考

data-
parallel
operators

RDDs are immutable, partitioned collections that can be created through various data-parallel *operators* (*e.g., map*, *group-by*, *hash-join*). Each RDD is either a collection residing in an external storage (*e.g.,* on disks), or a derived dataset created by applying operators to other RDDs. For example, given an RDD of (visitID, URL) pairs for visits to a website, we might compute an RDD of (URL, count) pairs by applying a *map* operator to turn each event into an (URL, 1) pair, and then a *reduce* to add the counts by URL. RDD operations are invoked through a functional interface in Scala, Java, or Python. For example, the Scala code for the query above is:

```
val visits = spark.hadoopFile("hdfs://...")
val counts = visits.map(v => (v.url, 1))
                   .reduceByKey((a, b) => a + b)
```

Spark RDDs can contain arbitrary objects (since Spark runs on the JVM, these elements are Java objects), and are automatically partitioned across the cluster, but they are immutable once created, and they can only be created through Spark's deterministic parallel operators. These two restrictions, however, enable efficient fault recovery. In particular, instead of replicating each RDD across nodes for fault-tolerance, Spark maintains the *lineage* of the RDD (the graph of operators used to build it), and recovers lost partitions by *recomputing* them from base data. For example, Figure 1 shows the lineage graph for the RDDs computed above. If Spark loses one of the partitions in the (URL, 1) RDD, it can recompute it by rerunning the *map* on just the corresponding partition of the input.

In the next section, we discuss how we leverage various properties of RDDs and Spark to implement the Resilient Distributed Graph (RDG) abstraction.

```
class Graph[V, E] {
 def vertices(): RDD[(Id, V)]

 def edges(): RDD[(Id, Id, E)]

 def filterVertices(f: (Id, V)=>Bool): Graph[V, E]

 def filterEdges(f: Edge[V, E]=>Bool): Graph[V, E]

 def mapVertices(f: (Id,V)=>(Id,V2)): Graph[V2, E]

 def mapEdges(
   f: (Id, Id, E)=>(Id, Id, E2)): Graph[V, E2]

 def updateVertices(tbl: RDD[(Id, A)],
   func: (Id, V, A)=>(Id, V2)): Graph[V2, E]

 def aggregateNeighbors(
   mapFunc: (Id, Edge[V, E]) => A,
   reduceFunc: (A, A) => A): RDD[(Id, A)]

 def reverseEdgeDirection(): Graph[V, E] =
   mapEdges(e => (e.dst, e.src, e.data))

 def degree(): RDD[(Id, Int)] =
   aggregateNeighbors((id, e) => 1, (a, b)=> a + b)
}
```

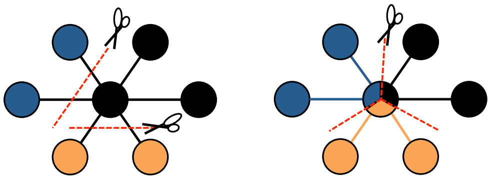
Listing 1: **Resilient Distribute Graph (RDG) Interface (in Scala):** The RDG encodes both the directed adjacency structure and contains the attributes associated with each vertex (`V`) and edge (`E`). Each vertex is identified by a unique vertex `Id`. The `Edge[V,E]` type represents an edge with its attributes as well as the attributes of *both* vertices and Spark RDDs over records of type `T` are denoted by `RDD[T]`. The method `edges()` returns the set of all edges as an RDD of three-element-tuples consisting of the source `Id`, target `Id`, and the edge attribute `E`. Alternatively, `mapVertices` returns a new graph by applying the function `f` which maps vertex id and attribute pair `(Id, V)` to a new vertex id and attribute pair.

3. RESILIENT DISTRIBUTED GRAPHS

Analogous to data-parallel computation which adopts a record centric view of data, graph-parallel computation typically adopts a vertex (and occasionally edge) centric view of computation. However, in contrast to data-parallel systems which define program logic as functional transformations on collections, existing graph-parallel systems restrict program logic to the level of vertex processes. Retaining the data-parallel metaphor, program logic in the GraphX system defines *transformations on graphs* with each operation yielding a new graph. As a consequence, the core data-structure in the GraphX systems is an immutable graph.

The GraphX graph consists of both the *directed adjacency structure* as well as user defined *attributes* associated with each vertex and edge. Programs in the GraphX system describe transformations from one graph to the next either through operators which transform vertices, edges, or both in the context of their neighborhoods (adjacent vertices and edges).

In Listing 1 we present the RDG graph interface which exposes methods to view, filter, and transform the graph. The methods `vertices()` and `edges()` provide tabular views of the vertices and edges. The methods `filterVertices(pred)` and `filterEdges(pred)` construct the sub-graphs satisfying the predicate `pred`. The methods `mapVertices(f)` and `mapEdges(f)` apply a user-defined function `f` to vertices and edges and return new graphs. In addition, the method `updateVertices(tbl,f)` transforms the vertices by first



(a) Edge-Cut (b) Vertex-Cut

Figure 2: **Edge-Cut vs Vertex-Cut:** An edge-cut **(a)** splits the graph along edges while a vertex-cut **(b)** splits the graph along vertices. In this illustration we partition the graph across three machines (corresponding to color).

joining the user supplied table and then mapping the result. Finally the method `aggregateNeighbors(m,r)` joins the vertex and edge data, maps the joined edges using the `m` function, and then reduces by the destination vertex id using the `r` function.

In the next section we demonstrate that the RDG interface is sufficiently expressive to easily implement the Pregel and PowerGraph programming abstractions.

3.1 Partitioning

Unlike data-parallel computation in which data is processed in isolation, graph-parallel computation requires each vertex or edge to be processed in the context of its neighborhood. Moreover each transformation depends on the result of *distributed joins* between vertices and edges. As a consequence, indexing and data layout are important steps in achieving an efficient distributed execution. Because the graph structure describes data movement, distributed graph computation systems rely on graph partitioning and efficient graph storage to minimize communication and storage overhead, and ensure balanced computation.

3.1.1 From Edge-Cuts to Vertex-Cuts

Most graph-parallel systems partition the graph by constructing an **edge-cut**. An edge-cut uniquely assigns vertices to machines while allowing edges to span across machines (see Figure 2a). The communication and storage overhead of an edge-cut is directly proportional to the number of edges that are cut. Therefore we can reduce communication overhead and ensure balanced computation by minimizing both the number of cut edges as well as the number of vertices assigned to the most loaded machine.

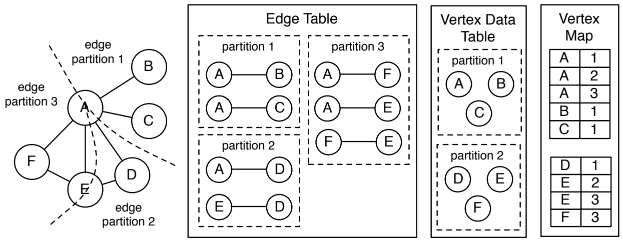
However, for most large-scale real-world graphs, constructing an optimal edge-cut can be prohibitively expensive. As a consequence, many graph computation systems have adopted the strategy of randomly distributing vertices across the cluster, *i.e.,* constructing a **random edge-cut**. However as [6] demonstrated that while random edge-cuts achieve nearly optimal work balance they also achieve nearly worst-case communication overhead, cutting *most* of the edges in the graph.

In contrast to edge-cuts which evenly assign vertices to machines, **vertex-cuts** evenly assign edges to machines and allow vertices to span multiple machines. In Figure 2b we illustrate the vertex-cut for the same graph. The communication and storage overhead of a vertex-cut is directly proportional to the sum of the number of machines spanned by each vertex. Therefore, we can reduce communication overhead and ensure balanced computation by evenly assigning edges to machines in way that minimizes the number of machines spanned by each vertex. In contrast to edge-cuts which have been shown [8, 1, 7] to perform poorly on real-world graphs, there are theoretical [2] and experimental [6] results indicating that real-world graphs have good vertex-cuts.

While constructing optimal vertex-cuts is also prohibitively expensive on large-scale real-world graphs, [6] proposed several simple *data-parallel* heuristics for edge-partitioning. The simplest strategy is to use a hash function to randomly assign edges to machines. Through a simple analysis it can be shown that for the power-law degree distributions found in real-world graphs, random vertex-cuts can be orders of magnitude more efficient than random edge-cuts. By cleverly constructing the hash function $h(i \to j)$ for each edge we can guarantee that each vertex spans at most $2\sqrt{M}$ of the the machines in a cluster of size M. This can be achieved by extending 2D partitioning [4] with hashing:

$$h(i \to j) = \sqrt{M} \times (h(i) \bmod \sqrt{M}) + (h(j) \bmod \sqrt{M}) \qquad (1)$$

where the number of machines is a perfect square $\sqrt{M} \in \mathbb{N}$ and $h(i)$ is a uniform hash function on the vertex ids.



A	1
A	2
A	3
B	1
C	1
D	1
E	2
E	3
F	3

Figure 3: **GraphX Tabular Representation of a Vertex-Cut:** Here we partition the graph on the left across three virtual partitions using a vertex-cut. The edge table contains the edge data as well as the vertex ids for each edge and is partitioned by the virtual `pid` field associated with each record. The vertex table contains the vertex id and vertex data and is partitioned (keyed) by the vertex `id`. Finally, the vertex map contains tuples of `(vid,pid)` and encodes the mapping from vertex `id` to the edge table partitions which contain adjacent edges. The vertex map table is also partitioned and keyed by the vertex id.

3.1.2 Vertex-Cuts as Tables in GraphX

The GraphX resilient distributed graph (RDG) data-structure achieves a vertex-cut representation of a graph using three unordered horizontally partitioned tables implemented as Spark RDDs. Readers are encouraged to refer to Figure 3 as an example to illustrate the internal representation.

1. `EdgeTable(pid, src, dst, data)`: stores the adjacency structure and edge data. Each edge is represented as a tuple consisting of the source vertex id, destination vertex id, and user-defined data as well as a virtual partition identifier (`pid`). Note that the edge table contains only the vertex ids and not the vertex data. The edge table is partitioned by the `pid`.
2. `VertexDataTable(id, data)`: stores the vertex data, in the form of a vertex (id, data) pairs. The vertex data table is indexed and partitioned by the vertex id.
3. `VertexMap(id, pid)`: provides a mapping from the id of a vertex to the ids of the virtual partitions that contain adjacent edges. For example in Figure 3, because vertex A is associated with edges in all partitions, there are three tuples related to A in the vertex map table. The vertex map table is partitioned by the vertex id.

During graph computations, we often need to assemble an edge with the data associated on both vertices. GraphX uses a 3-way

```
def Pregel(graph: Graph[V,E],
           initialMsg: M
           vprogf: ((Id,V), M) => V,
           sendMsgf: Edge[V,E] => Option[M],
           combinef: (M,M) => M,
           numIter: Long): Graph[V,E] = {
  // Initialize the messages to all vertices
  var msgs: RDD[(Vid, A)] =
    graph.vertices.map(v => (v.id, initialMsg))

  // Loop while their are messages
  var i = 0
  while (msgs.count > 0 && i < maxIter) {
    // Receive the message sums on each vertex
    graph = graph.updateVertices(msgs, vprogf)

    // Compute and combine new messages
    msgs = graph.aggregateNeighbors(sendMsgf,
      combinef)
    i = i + 1
  }
}
```

Listing 2: **Pregel Runtime in GraphX:** The Pregel runtime applies user defined functions to transform the vertex attributes of a GraphX RDG. The Pregel method takes as arguments the RDG parameterized by `V` and edge attributes `E`, an initial message of type `M`, the `vprogf` function which takes a vertex and a message and returns a new attribute value for that vertex, a `sendMsgf` function which computes the new message along each edge, and a `combinef` function which is used to combine messages to the same vertex.

relational join to bring together the source vertex data, edge data, and target vertex data:

```
VertexDataTable v
JOIN
VertexMap vm
  ON (v.id=vm.id)
RIGHT OUTER JOIN
EdgeTable e
  ON (e.pid=vm.pid && (e.src=v.id OR e.dst=v.id))
WITH PARTITIONER edgeTable.partitioner ON pid
```

The joins are fairly straightforward with the exception of the partitioner. As discussed in the previous subsection, the edge table is often much larger than the vertex data table. The partitioner is a hint to Spark to ensure the join site would be local to the edge table. This allows GraphX to shuffle only the vertex data and avoid moving any of the edge data.

Note that conceptually the vertex data table and the vertex map table can be merged as a single table. However, we separate them into two tables due to their functional differences: the vertex data table contains states associated with the vertices that are changing in the course of graph computations, while the vertex map table remains static as long as the graph structure does not change. To minimize the communication, GraphX co-partitions the two tables so the first join can be done locally.

The resulting table from the 3-way join presents an edge-centric view of the graph, with each tuple containing the edge data, source vertex data, and the target vertex data. This table can be used to implement the basic transformations such as `aggregateNeighbors`:

```
SELECT dstVid, reduceFunc(*) FROM (
  SELECT dstVid, mapFunc(*) FROM edgeWithVertices)
GROUP BY dstVid
```

```
// Load and initialize the graph
val graph = Graph.load('hdfs://webgraph.tsv')
var prGraph = graph.updateV(graph.degrees(OutEdges),
  (v,deg) => (v.id,(deg, 1.0)) // Initial rank=1

// Execute PageRank
prGraph = Pregel(prGraph,
  1.0, // Initial message is 1.0
  vprogf = // Update Rank
    (v, msg) => (v.deg, 0.15 + 0.85 * msg),
  sendMsgf = // Compute Msg
    e => e.src.rank/e.src.deg,
  combinef = // Combine msg
    (m1, m2) => m1 + m2,
  10) // Run 10 iterations

// Display the maximum PageRank
print(prGraph.vertices.map(v=>v.rank).max)
```

Listing 3: **PageRank in Pregel:** The graph is loaded from HDFS, and the default vertex attributes are replaced with a tuple consisting of the out-degree and the initial PageRank. We then apply then define and apply the PageRank algorithm using only three tiny functions. Finally, we extract and print the maximum PageRank value from the resulting Graph.

4. GRAPH-PARALLEL COMPUTATION

While the basic GraphX RDG interface naturally expresses graph transformations, filtering operations, and queries, it does not directly provide an API for recursive graph-parallel algorithms (*e.g.,* PageRank). Instead, the GraphX interface was designed to enable the construction of new graph-parallel APIs. By composing operations in the RDG interface we are able to compactly express several of the most widely used graph-parallel abstractions and in this section we provide the actual code for PowerGraph [6] and Pregel [9] written in a few lines using the GraphX interface.

Existing **graph-parallel** abstractions like PowerGraph [6] and Pregel [9] adopt a vertex centric programming model in which the user implements a **vertex program** Q which is executed in *parallel* on each vertex $v \in V$ in the *sparse* graph $G = \{V, E\}$. Each instance $Q(v)$ of the vertex program transforms the vertex attributes by interacting, through shared state or messages, with neighboring instances $Q(u)$ where $(u, v) \in E$. Because most graph algorithms factor according to the graph structure, they can be naturally expressed in the form of vertex-programs. By restricting the scope of computation, automatically coordinating communication, and addressing the challenges of distributed execution, graph-parallel abstractions substantially simplify the implementation of scalable distributed graph algorithms. In this section we will use the PageRank [10] as an example of a canonical graph-parallel algorithm.

EXAMPLE 4.1 (PAGERANK). *The PageRank algorithm recursively defines the rank of a vertex v:*

$$\boldsymbol{Pr}(v) = 0.15 + 0.85 \sum_{u \text{ links to } v} w_{u,v} \times \boldsymbol{Pr}(u) \qquad (2)$$

in terms of the weighted $w_{u,v}$ ranks $\boldsymbol{Pr}(u)$ of the vertices u that link to v. The PageRank algorithm iterates Eq. (2) until the ranks of all vertices converge.

4.1 Pregel

Pregel [9] is a bulk synchronous *message passing* graph-parallel abstraction in which all vertex programs run concurrently in a sequence of super-steps. Within a **super-step** each program instance

```
def PowerGraph(graph: Graph[V,E],
  gatherf: Edge[V,E] => A,
  sumf: (A, A) => A,
  applyf: ((Id,V), A) => V,
  signalf: Edge[V,E]=> Bool,
  maxIter: Int): Graph[V,E] = {

  // Extend the vertex data to include isActive
  var glGraph =
    graph.mapVertices((Id,v) => (id, (true,v)))

  // Loop while there are active vertices
  var i = 0
  var nActive = g.numVertices
  while (i < maxIter && nActive > 0) {
    // Execute the gather phase
    val acc = glGraph.filterE(e => e.dst.isActive)
      .aggregateNeighbors(gatherf, sumf)

    // Execute the apply phase.
    glGraph = glGraph.updateVertices(acc, applyf)

    // Execute the Scatter Phase
    val active = glGraph.filter(e=>e.src.isActive)
      .aggregateNeighbors(scatterf, (a,b)=> a||b)

    // Update activity status of vertices
    glGraph = glGraph.updateVertices(active,
      ((id, (old,v)), active)=>(id,(active,v)))

    // Count the number of active vertices
    nActive = glGraph.vertices
      .map(v => v.isActive)
      .reduce((a,b) => a + b)
    i = i + 1
  }
  // Return the graph (without active flag)
  return glGraph.mapVertices(v => (v.id, v.data))
}
```

Listing 4: **PowerGraph Runtime in GraphX:** The PowerGraph program logic is similar to the Pregel program logic however we augment the vertex data to track active vertices and then restrict the `aggregateNeighbors` operations to run on the subset of edges adjacent to active vertices. During the gather phase we only consider edges inbound to active vertices and then during the scatter phase we only consider edges departing from active vertices. The program loops while their are still active vertices or the max number of iterations is achieved.

```
// Load and initialize the graph
val graph = Graph.load(’hdfs://webgraph.tsv’)

// Initialize the graph for dynamic PageRank by
// storing the degree and the old and new PageRank
var prGraph = graph.updateV(graph.degrees(OutEdges),
  (v,deg) => (v.id, (deg, 1.0, 1.0))

// Execute PageRank
prGraph = PowerGraph(prGraph,
  gatherf = e => e.src.rank / e.src.deg,
  sumf = (a,b) => a + b,
  applyf = // Update rank and save previous rank
    (v, a) => (v.deg, 0.15 + 0.85*a, v.rank)
  scatterf = // Activate neighbors on big change
    e => abs(e.src.rank - e.srd.oldRank) > eps,
  10) // Run 10 iterations

// Display the maximum PageRank
print(prGraph.vertices.map(v=>v.rank).max)
```

Listing 5: **Dynamic PageRank in PowerGraph:** The graph is loaded from HDFS, and the default vertex attributes are replaced with a tuple consisting of the out-degree and the initial PageRank and the old PageRank. In dynamic PageRank we keep track of how much the rank changes on each update and only trigger neighbors to recompute if the rank changes by more than some small constant epsilon.

$Q(v)$ receives the *sum* of all messages sent by neighbors in the previous super-step, computes a new value, and then sends messages to its out neighbors in the next super-step. A barrier is imposed between super-steps ensuring that all messages are received before entering the next super-step. The message sum is computed through a user-defined commutative associative binary **message combiner**.

In Listing 2 we provide the actual[1] code for a modified version of the Pregel runtime implemented in a few lines using GraphX. The program takes as input a graph, an initial message to broadcast to all vertices, the vertex program, a function that computes the message along each edge, the message combiner, and the maximum number of iterations.

We have adopted a more functional API than was described by [9] in which the vertex program is a mapping from the old vertex value and message to a new vertex value. Furthermore, unlike the original Pregel API in which the vertex program is passed the set of neighbors and returns a list of messages, our implementation learns from the observations made by [6] and lifts the message construction out of the vertex-program. Messages are computed using a message generating function which takes an edge containing the source and target attributes and returns a message or void indicating the absence of a message. By lifting the message construction out of the vertex-program, we are able to achieve a *more efficient* execution than the original Pregel framework and leverage the vertex-cut representation. Moreover, message construction for a single vertex can be distributed over the cluster, moved to the receiving machine, and executed in an efficient order.

When their are no remaining messages (*i.e.,* all message calculations return void) or the maximum number of iterations is achieved the Pregel execution terminates returning the new graph. In Listing 3 we use the GraphX Pregel API (defined in Listing 2) to load and prepare a graph, implement and apply the PageRank algorithm, and then finally compute the highest PageRank.

4.2 PowerGraph

In the PowerGraph abstraction vertex-programs interact by directly reading the state of neighboring vertices and edges. To efficiently achieve this shared-memory illusion in the distributed setting, [6] introduced the Gather-Apply-Scatter (**GAS**) decomposition which further decomposes a vertex-program into three functional phases. During the gather phase, a map-reduce job is run on the incoming-edges of each active vertex (this is precisely the `aggregateNeighbors` operation in the GraphX API). Then, during the apply phase, the output of the gather phase is consumed along with the old vertex attribute and a new vertex attribute is computed (this is precisely the `updateVertices` operation in the GraphX API).

Finally, during the scatter phase, a predicate is evaluated on all outgoing-edges of active vertices. If any of the incoming-edges of a vertex evaluate to true then it becomes active during the next super-step. If a vertex is not activated in the previous phase then it is skipped in the subsequent phase. The ability to modulate computation and *pull* information during the gather phase allows

[1] We have omitted some Scala syntax to keep lines short.

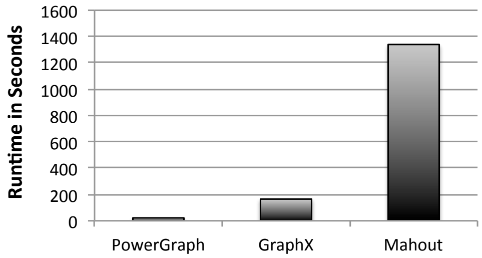


Figure 4: **PageRank Runtime Comparison** between GraphX, Mahout/Hadoop, and PowerGraph. The reported runtime includes the time to load the graph from HDFS and then run 10 iterations of PageRank.

us to easily express a much more efficient dynamic PageRank (see Listing 5).

In Listing 4 we provide the code for the PowerGraph runtime using the GraphX API. The program takes as input a graph, an initial message to broadcast to all vertices, the vertex program, a function that computes the message along each edge, the message combiner, and the maximum number of iterations. The program logic is relatively similar to that of Pregel except that we filter the graph during the gather and scatter phase dropping edges adjacent to inactive vertices.

5. PRELIMINARY EXPERIMENTS

We conducted a preliminary performance evaluation of the GraphX system by comparing against two popular data-parallel and graph-parallel platforms:

1. **Apache Mahout:** version 0.6 on Hadoop 0.20.205.
2. **PowerGraph:** the latest release as part of GraphLab 2.1.

We evaluated each platform by timing the execution of the PageRank algorithm on the `LiveJournal`[8] graph with approximately 4.8 million vertices and 69 million edges. All experiments were conducted on Amazon EC2 using 16 `m2.4xlarge` nodes. Each node has 8 virtual cores, 68GB of memory, and runs 64-bit Linux 3.2.28.

As illustrated in Figure 4, our results show that GraphX is 8X faster than a general data-parallel platform (Mahout/Hadoop), but 7X slower than PowerGraph, a heavily optimized graph-parallel execution engine and one of the fastest open-source graph-parallel frameworks. We emphasize that it is not our intention to beat PowerGraph in performance. GraphX is designed to be a general engine that supports a wide range of operations including loading, construction, transformation, and computation, while leveraging the performance advances developed in the context of dedicated graph computation systems like PowerGraph. We believe that the loss in performance may, in many cases, be ameliorated by the gains in productivity achieved by the GraphX system.

However, we have already identified a number of candidates for performance improvement in our prototype. It is our belief that we can shorten the gap in the near future, while providing a highly usable *interactive system* for graph data mining and computation.

6. CONCLUSIONS AND FUTURE WORK

We have presented GraphX, an interactive graph computation engine that combines the advantages of graph-parallel systems and data-parallel systems. It provides a programming abstraction called Resident Distributed Graphs (RDGs) that significantly simplifies graph loading, construction, transformation, and computations. Based on RDGs, we implement Pregel and PowerGraph APIs in 20 lines of code. GraphX's internal data representation uses a vertex-cut partitioning scheme that minimizes the movement of data during graph computation. GraphX will be open sourced as part of the Berkeley Data Analytics Stack.

We have identified a number of possible future research directions. First, we will continue to work on improving the performance of GraphX. Second, our tabular representation of the vertex-cut partitioning is a natural fit for relational databases and we would like to implement the GraphX abstraction on top of a distributed relational database. A thorough study of performance characteristics between relational databases, Spark, and PowerGraph will help us understand the trade-off in more general systems. Last but not least, we plan to investigate more declarative interfaces that can be compiled down into GraphX programs to further simplify graph computation.

7. ACKNOWLEDGMENTS

We thank Haijie Gu and other members of the GraphLab team for discussions on the GraphX prototype. We also thank Gene Pang and the anonymous reviewers for useful feedback on earlier drafts of this paper. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

8. REFERENCES

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, 2006. 3.1.1

[2] R. Albert, H. Jeong, and A. L. Barabási. Error and attack tolerance of complex networks. In *Nature*, volume 406, pages 378—482, 2000. 3.1.1

[3] A. Buluç and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011. 1

[4] U. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, 2010. 3.1.1

[5] R. Cheng et al. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, 2012. 1

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*. 1, 2, 3.1.1, 4, 4.1, 4.2

[7] K. Lang. Finding good nearly balanced cuts in power law graphs. Technical Report YRL-2004-036, Yahoo! Research Labs, Pasadena, CA, Nov. 2004. 3.1.1

[8] J. Leskovec et al. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2008. 3.1.1, 5

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010. 1, 4, 4.1

[10] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999. 4

[11] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *ISWC*, 2010. 1

[12] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012. 1