Chapter 1

GraphX and Spark

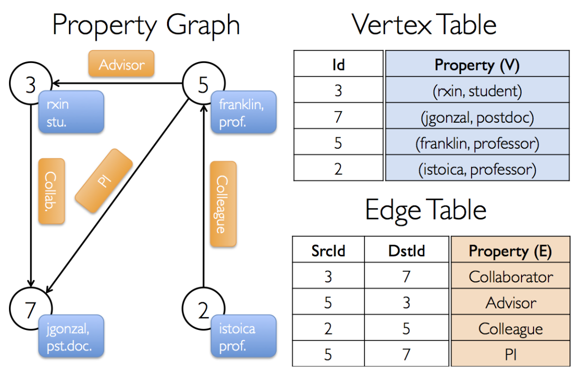
Contributed by Mark Horeni

1.1 Background

GraphX along with Apache Spark were made from University of California Berkely [4]. The purpose of Spark was to have the function of map reduce, but still have everything in memory so that operations can run faster [4]. GraphX is a library that was written on top Spark using Java and Scala to take advantage of this in memory form of map reduce with specifically graphs in mind [2].

1.2 Expressing Graphs

The main datatype in GraphX and Spark is the Resilient Distributed Database (RDD). This data type is a read only collection of information. Since an RDD is read only, there is no way to transform the graph once it has been created, only by creating an entirely new RDD can changes in a graph occur. Although this is true, RDD's have their advantages as they are usually faster because of their ability to run backup copies of nodes, and can be programmed with locality in mind. RDD's are also very fault tolerant, and when faults do occur, reconstruction of data can be parallelized so performance impact is minimal [3]. The general abstraction in GraphX is a *Property Graph*, a directed graph with edge metadata stored inside the graph. Edges and nodes are saved in their own respective tables along with their metadata. An example of this can be shown in Figure 1.1.



Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Figure 1.1: Example of a Property Graph [1]

1.3 Syntax

The main language used by GraphX and Spark is Scala, though Java can be used in place of Scala. Since a graph is just another datatype that consists of edges and verticies expressed as an RDD, it can be coded like an RDD [4][2][3]. An example of the code in Figure 1.1 is expressed in Scala is given bellow [1]

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

1.4 Key Graph Primitives

By default, GraphX has PageRank, connected components, k-core, triangle counting, LDA, SVD++, and a few other primitives [2]. Some of the graph primitives such as PageRank, connected components, and K-core are written over the GAS Pregel API, but are possible to be written in native Spark [2]. Any message passing based graph primitive can be easily written in as a function, and as an example, the code for connected components can be seen bellow [2]

```
def ConnectedComp(g: Graph[V, E]) = {
    g = g.mapV(v => v.id) //    Initialize vertices
    def vProg(v: Id, m: Id): Id = {
        if (v == m) voteToHalt(v)
        return min(v, m)
    }
    def sendMsg(t: Triplet): Id =
        if (t.src.cc < t.dst.cc) t.src.cc
        else None // No message required
    def gatherMsg(a: Id, b: Id): Id = min(a, b)
    return Pregel(g, vProg, sendMsg, gatherMsg)
}
```

1.5 Execution Model

The basis of the GraphX is the underlying engine of Spark. The general execution model of spark can be seen in Figure 1.3. The spark context is an object in a driver program that controls the cluster manager, and the cluster manager manages individual worker nodes who do map reduce

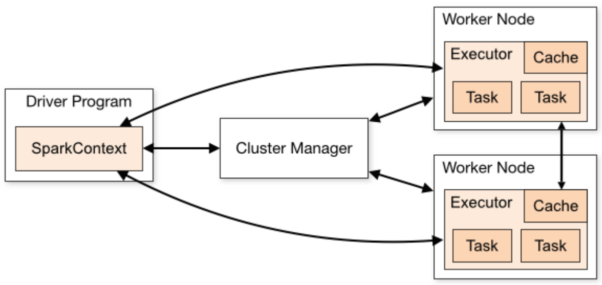


Figure 1.2: A view of the execution model of Spark [1]

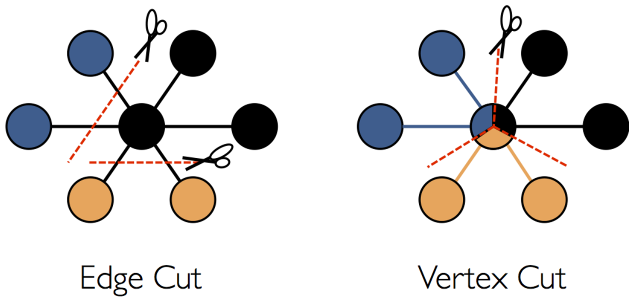


Figure 1.3: An diagram of vertex cutting vs edge cutting [1]

tasks, and the work nodes then return the results of those tasks back to the driver program [4]. This is done completely in memory ensure fast execution times [4].

Inside GraphX, to achieve parallelization, the library uses a technique called edge cutting as illustrated in Figure 1.3. This is done either at randomm (by default) or using a user defined cutting heuristic [2]. This allows for operations to be done on subgraphs, and messages passed through a routing table [2]. An example of this can be seen in Figure 1.4.

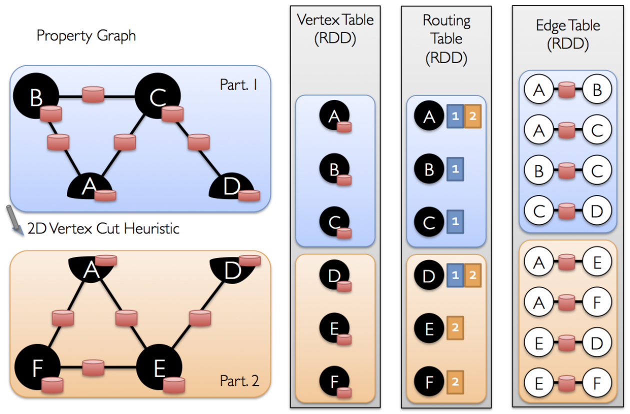
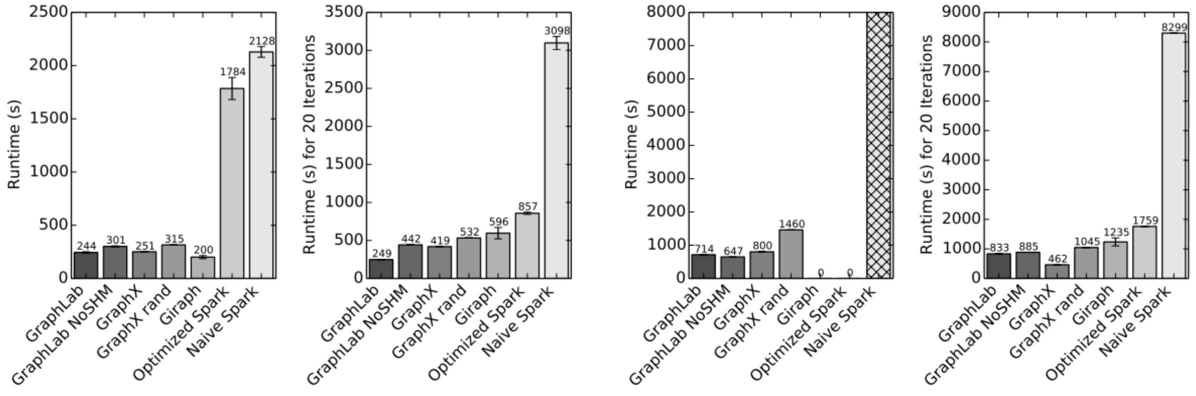


Figure 1.4: A diagram of vertex routing [1]



(a) Conn. Comp. Twitter (b) PageRank Twitter (c) Conn. Comp. uk-2007-05* (d) PageRank uk-2007-05

Figure 1.5: Runtimes of various primitives

1.6 Examples

As mentioned before, because GraphX runs completely in memory, fast computation times are achieved [2]. An example of runtimes of various primitives compared to other paradigms can be seen in Figure 1.5. For a primitive that is already in GraphX, running it is as simple as defining it as a map function the RDD as shown below.

```
import org.apache.spark.graphx.{GraphLoader, PartitionStrategy}

// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

1.7 Conclusion

GraphX and Spark are a simple in memory way of using map reduce on a graph to achieve faster computation time than other graph paradigms.

Bibliography

[1] Spark 2.3.0 docs.

[2] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework.

[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[4] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. 2016.