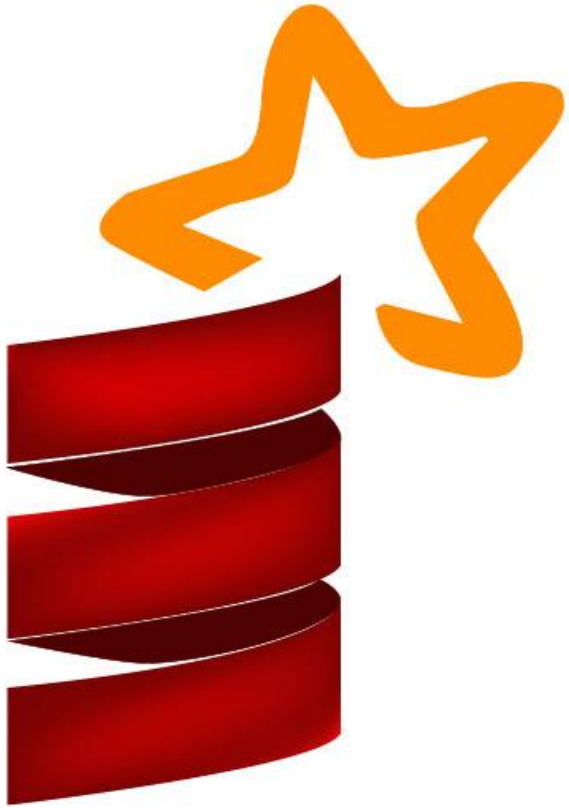
skillspeed

for the serious learner



Apache Spark and Scala

Module 8: Advanced Spark Concepts

Course Topics

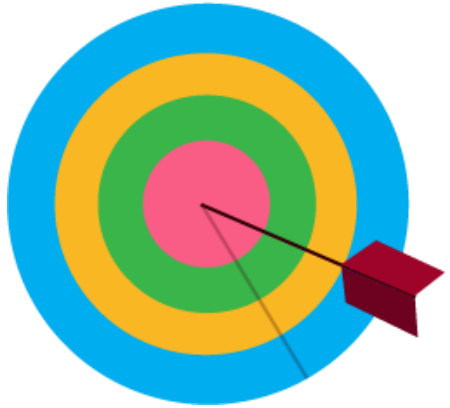
skillspeed
for the serious learner

Module 1 Getting Started / Introduction to Scala	**Module 2** Scala – Essentials and Deep Dive	**Module 3** Introducing Traits and OOPS in Scala	**Module 4** Functional Programming in Scala
Module 5 Spark and Big Data	**Module 6** Understanding RDDs	**Module 7** Shark, SparkSQL and Project Discussion	**Module 8** Advanced Spark Concepts

Session Objectives

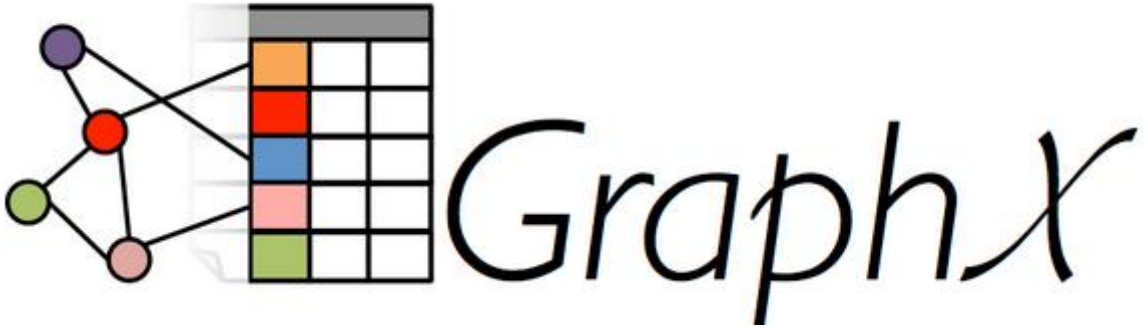
In this session, you will learn about

- Web Graphs
- Triangle Counting
- Collaborative Filtering
- The Graph – Parallel Pattern
- Complex Pipelines
- Property graphs, Creating a Graph, Built-in Algorithms, The triplets view, The subgraph transformation, Graph Coarsening, Sorting graphs as tables, Implementing Triplets



GraphX

- GraphX is Apache Spark's API for graphs and graph-parallel computation
- GraphX extends the distributed fault-tolerant collections API and interactive console of Spark with a new graph API which leverages recent advances in graph systems (e.g., GraphLab) to enable users to easily and interactively build, transform, and reason about graph structured data at scale



Characteristics of Graph X

- Flexible: It can work seam lessely with both graphs and collections. GraphX unites exploratory analysis, iterative graph computation and ETL easily and effectively within a single system
- RDD's can be transformed and joined easily in GraphX
- Fast: GraphX is faster as compared to any other graph, without comprising on flexibility and fault tolerance
- Rich set of Algorithms: GraphX comes with lots of Built-in Algorithms

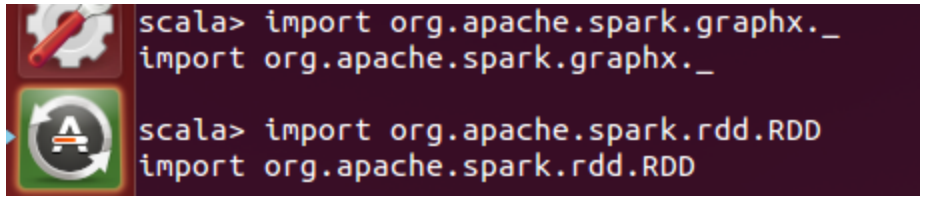
Introduction to the GraphX API

- In order to start, you first need to import GraphX
- Start the Spark-Shell

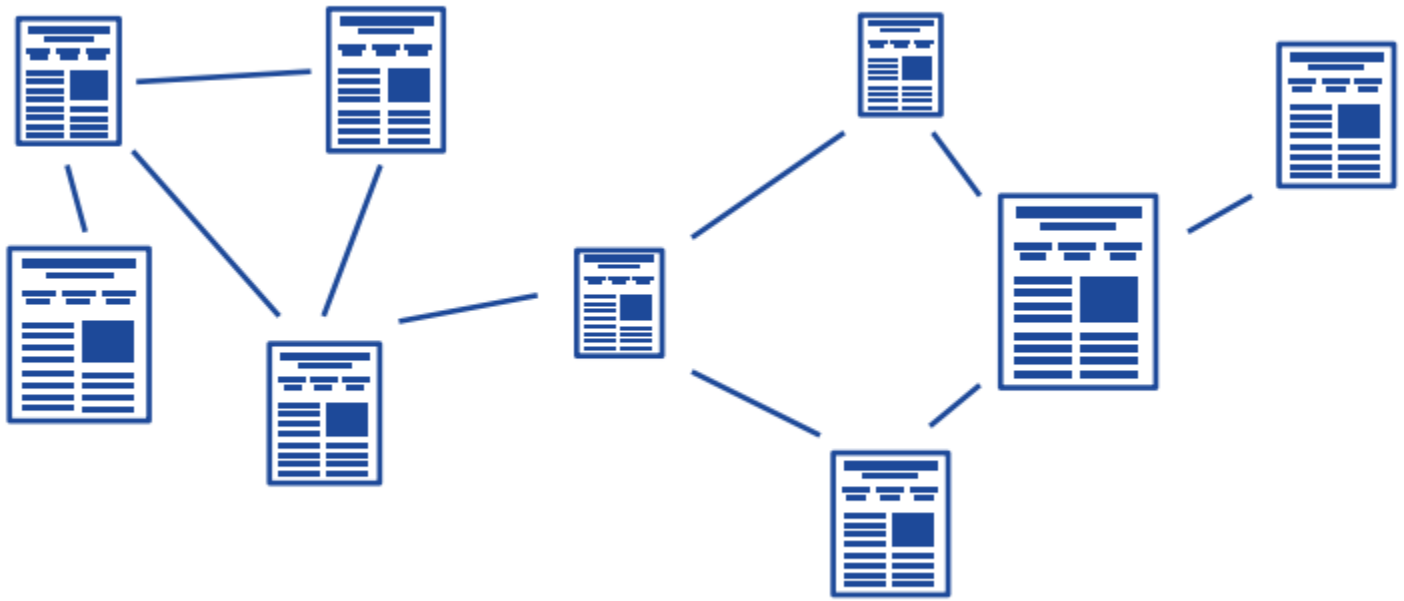
```
skillspeed@ubuntuvms:~/spark-1.1.0/bin$ ./spark-shell
```

And the paste the following code in your Spark shell:

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```



Web Graph

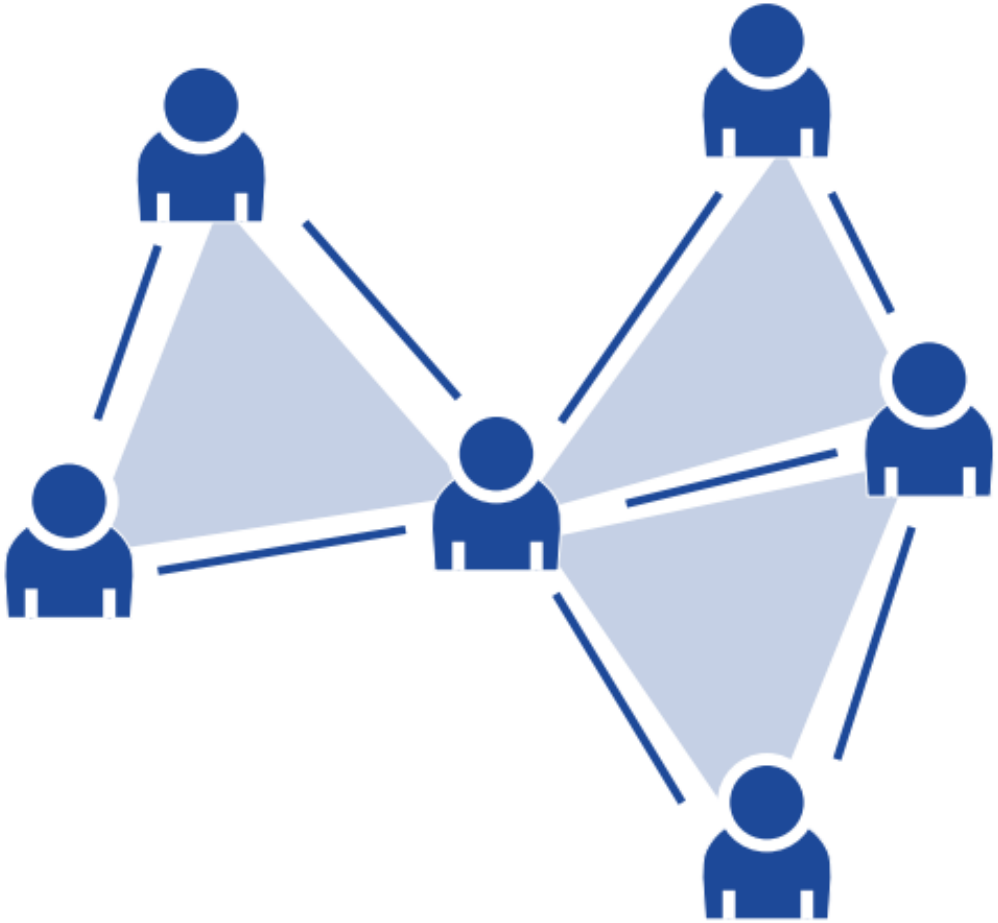


Triangle Counting

- Triangle Count is very useful in social network analysis.
- The triangle is a three-node small graph, where every two nodes are connected. For Example,
  - Suppose you're followed by two schoolmates in Facebook, and those two schoolmates are followed by each other, you three make up a triangle.
  - Likewise, the social network which owns more triangles usually has more tight connections

- TriangleCount is defined in [[lib/TriangleCount.scala]]. It counts the triangles passing through each vertex using a straightforward algorithm:
  - Compute the set of neighbors for each vertex
  - For each edge compute the intersection of the sets and send the count to both vertices
  - Compute the sum at each vertex and divide by two since each triangle is counted twice

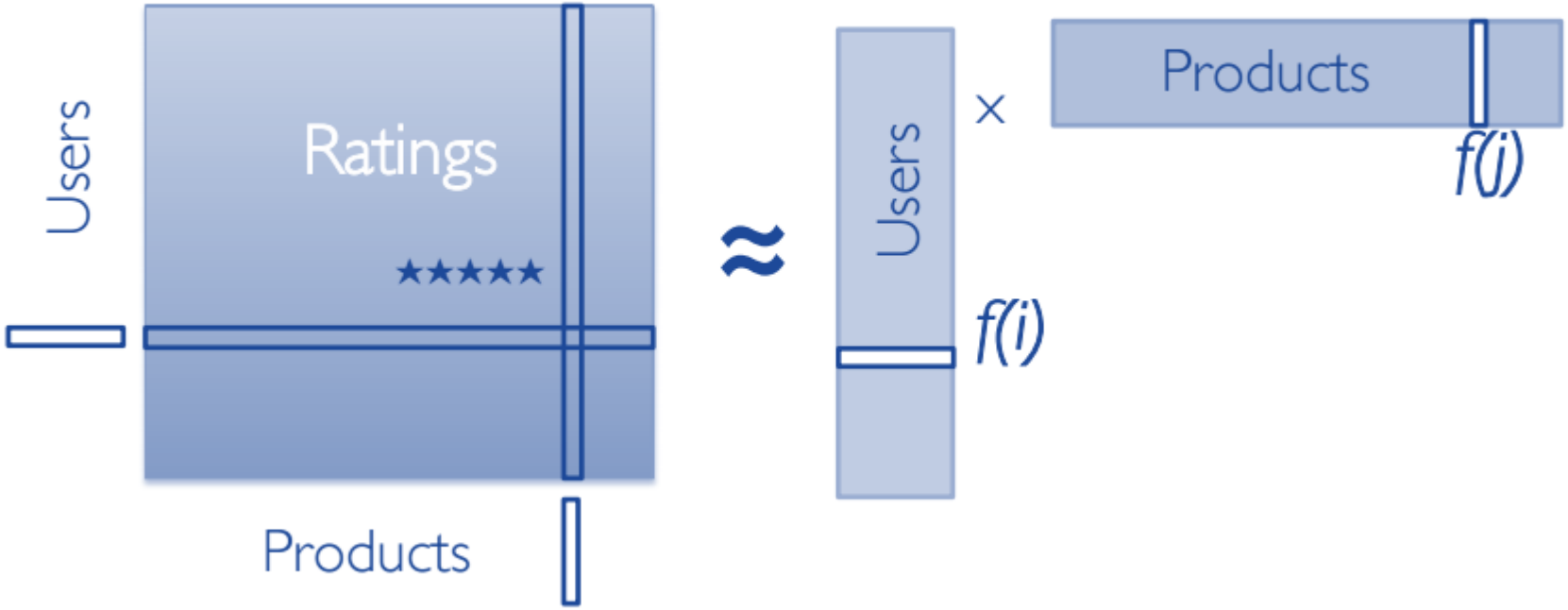
Triangle Counting (Cont'd)



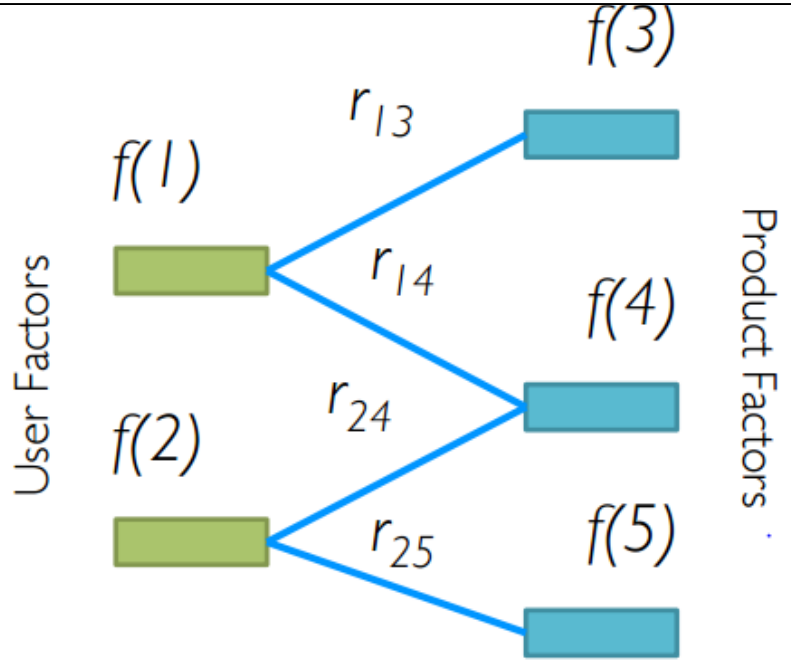
Collaborative Filtering

- Collaborative filtering is commonly used for Recommender systems
- These techniques aim to fill in the missing entries of a user-item association matrix
- MLlib currently supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries
- MLlib uses the alternating least squares (ALS) algorithm to learn these latent factors. The implementation in MLlib has the following parameters:
  - numBlocks is the number of blocks used to parallelize computation
  - rank is the number of latent factors in the model
  - iterations is the number of iterations to run
  - lambda specifies the regularization parameter in ALS
  - implicitPrefs specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data
  - alpha is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations

Collaborative Filtering (Cont'd)

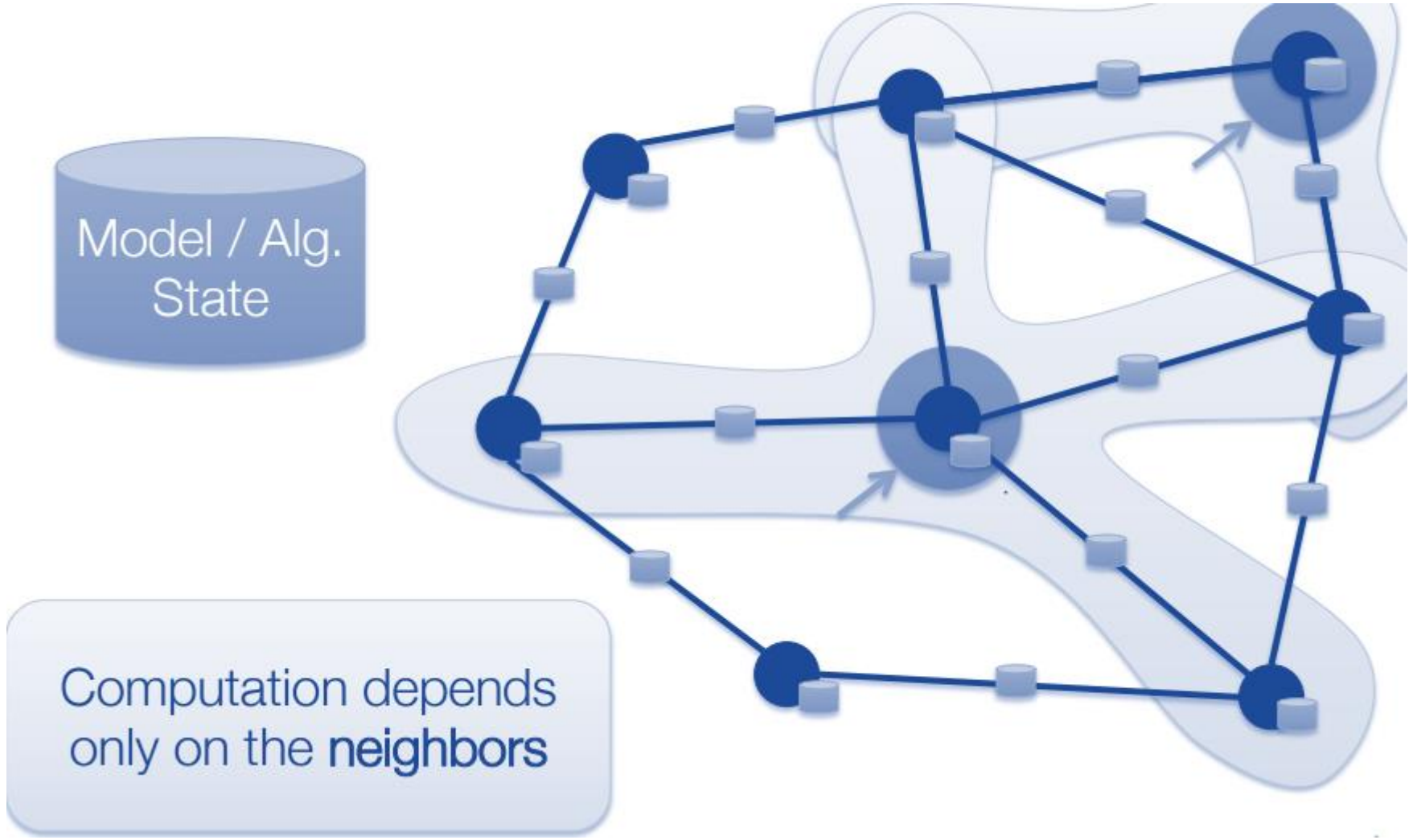


Collaborative Filtering (Cont'd)



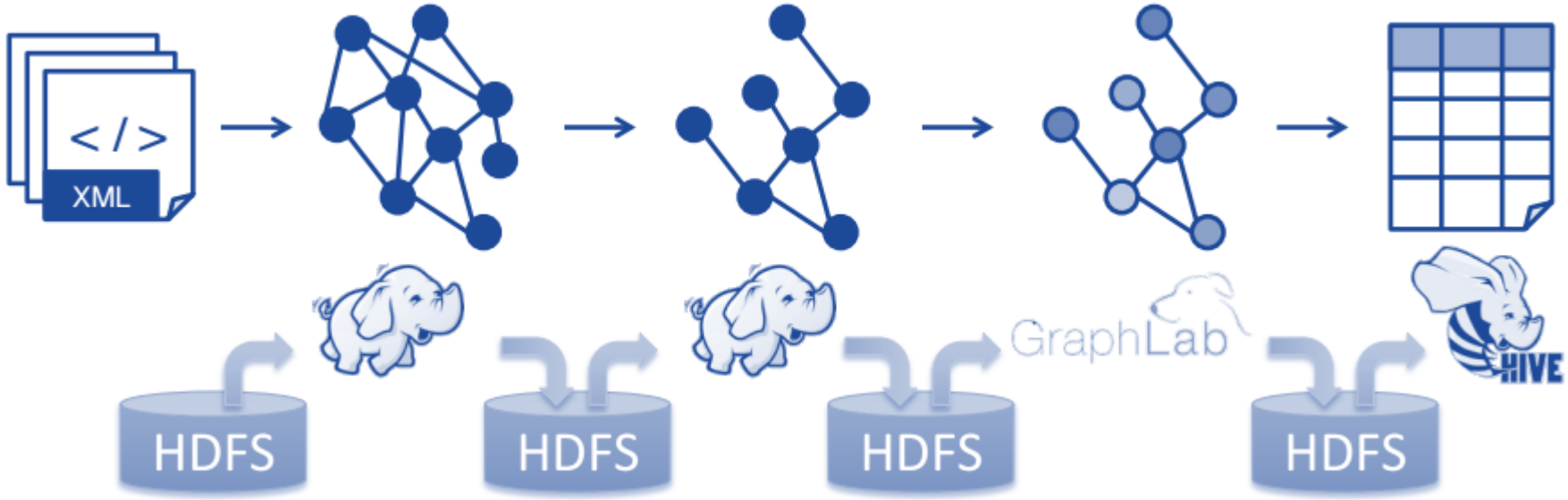
$$f[i] = \arg\min_{w \in \mathbb{R}^d} \sum_{j \in \text{Nbrs}(i)} \left(r_{ij} - w^T f[j]\right)^2 + \lambda ||w||_2^2$$

The Graph X – Parallel Pattern



Inefficient

Extensive data movement and duplication across the network and file system

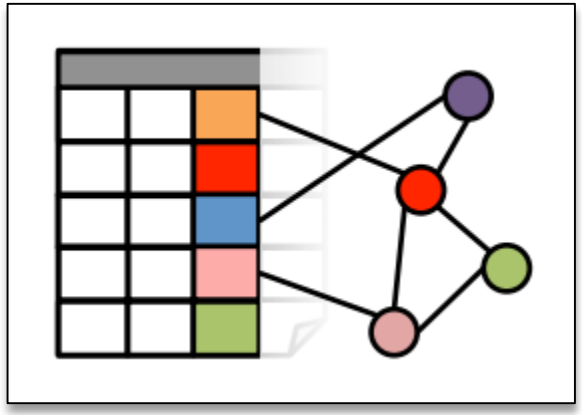


Limited reuse internal data-structures across stages

Solution: The GraphX Unified Approach

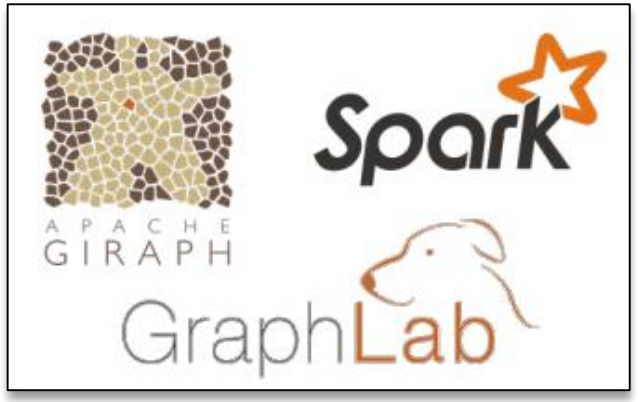
New API

Blurs the distinction between Tables and Graphs



New System

Combines Data-Parallel Graph-Parallel Systems



Enabling users to easily and efficiently express the entire graph analytics pipeline

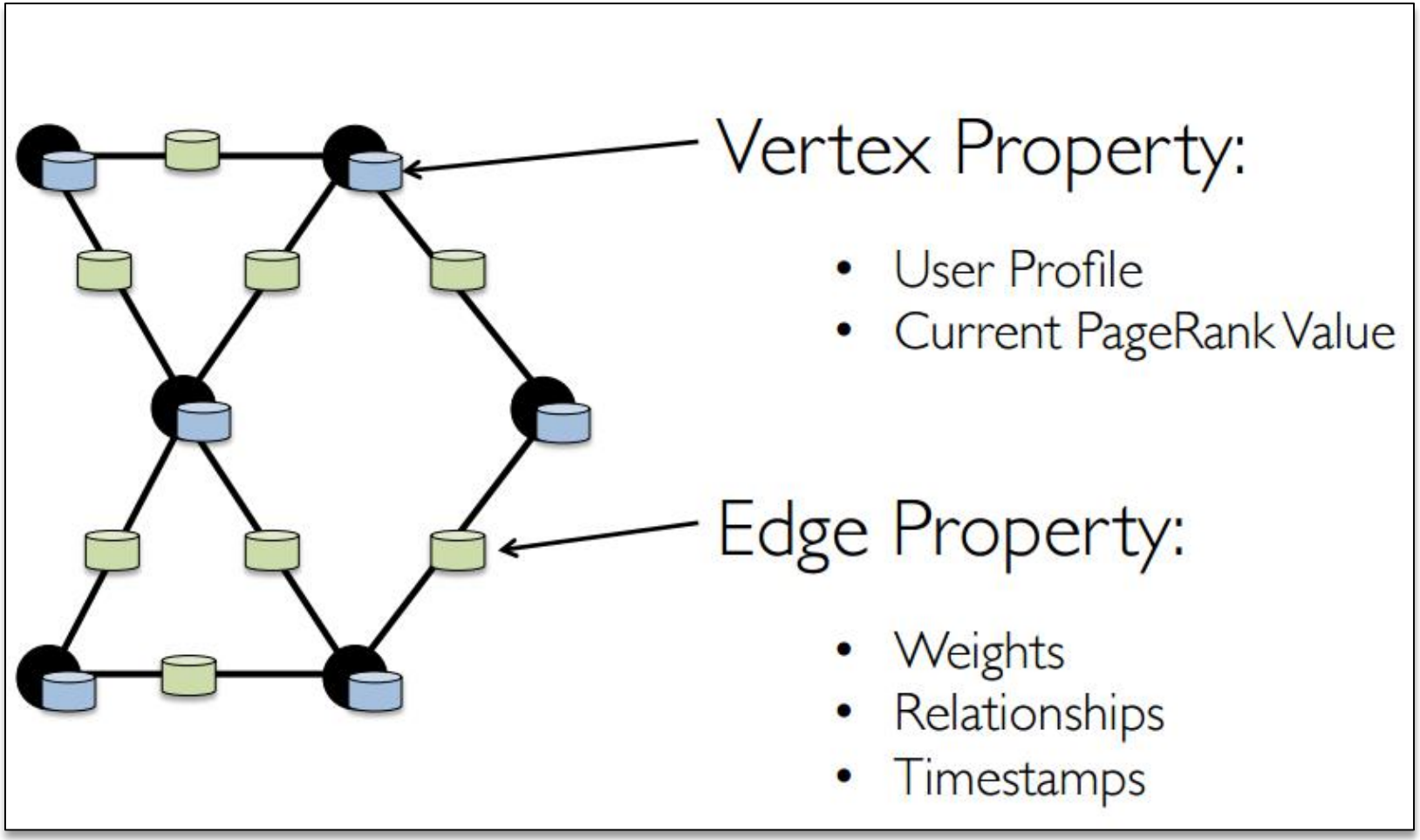
Property Graph

- The property graph is a directed multigraph (a directed graph with potentially multiple parallel edges sharing the same source and destination vertex) with properties attached to each vertex and edge
- Each vertex is keyed by a unique 64-bit long identifier (VertexID). Similarly, edges have corresponding source and destination vertex identifiers. The properties are stored as Scala/Java objects with each edge and vertex in the graph

Example:

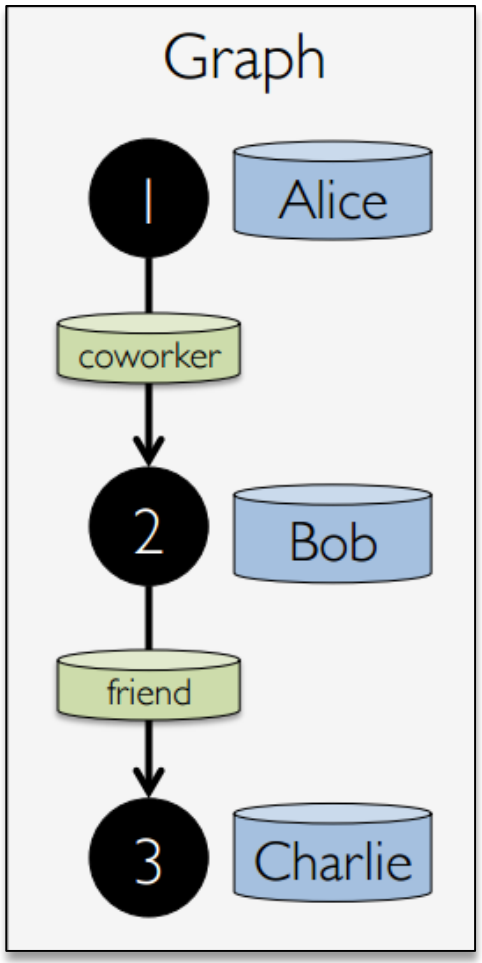
In this example we have a small social network with users and their ages modeled as vertices and likes modeled as directed edges

Property Graph (Cont'd)



Creating a Graph (Scala)

```
type VertexId = Long
val vertices : RDD[(VertexId,
String)]=
sc.parallelize(List(
    (1L,"Alice"),
    (2L,"Bob"),
    (3L,"charlie")))

class Edge[ED](
  val srcId: VertexId,
  val dstId: VertexId,
  val attr: ED)
val edges : RDD(Edge[String]] =
sc.parallelize(List(
   Edge(1L,2L,"coworker"),
   Edge(2L,3L, "friend"))
val graph = Graph(Vertices, edges)
```

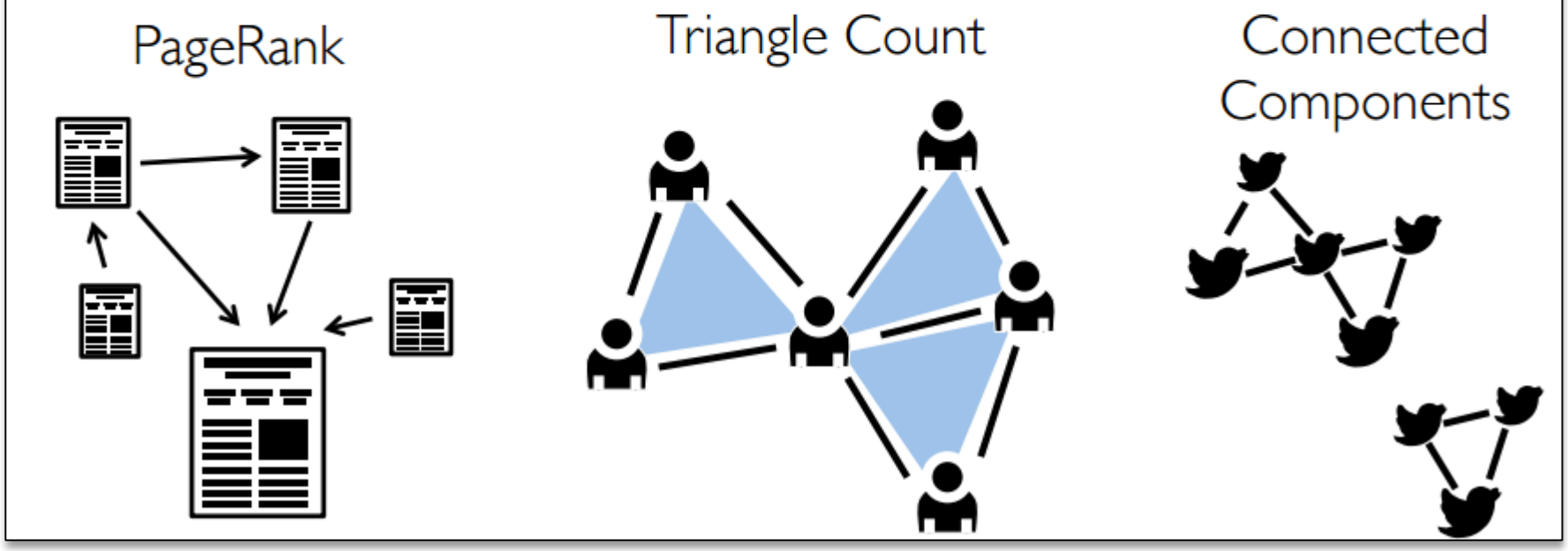


Graph Operations (Scala)

```scala
class Graph[VD, ED] {
    // Table Views -----------------------
    def vertices: RDD[(VertexId, VD)]
    def edges: RDD[Edge[ED]]
    def triplets: RDD[EdgeTriplet[VD, ED]]
    // Transformations ------------------------------------
    def mapVertices[VD2](f: (VertexId, VD) => VD2): Graph[VD2, ED]
    def mapEdges[ED2](f: Edge[ED] => ED2): Graph[VD2, ED]
    def reverse: Graph[VD, ED]
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
    // Joins ----------------------------------------
    def outerJoinVertices[U, VD2]
        (tbl: RDD[(VertexId, U)])
        (f: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
    // Computation ----------------------------------
    def mapReduceTriplets[A](
        sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
        mergeMsg: (A, A) => A): RDD[(VertexId, A)]
```

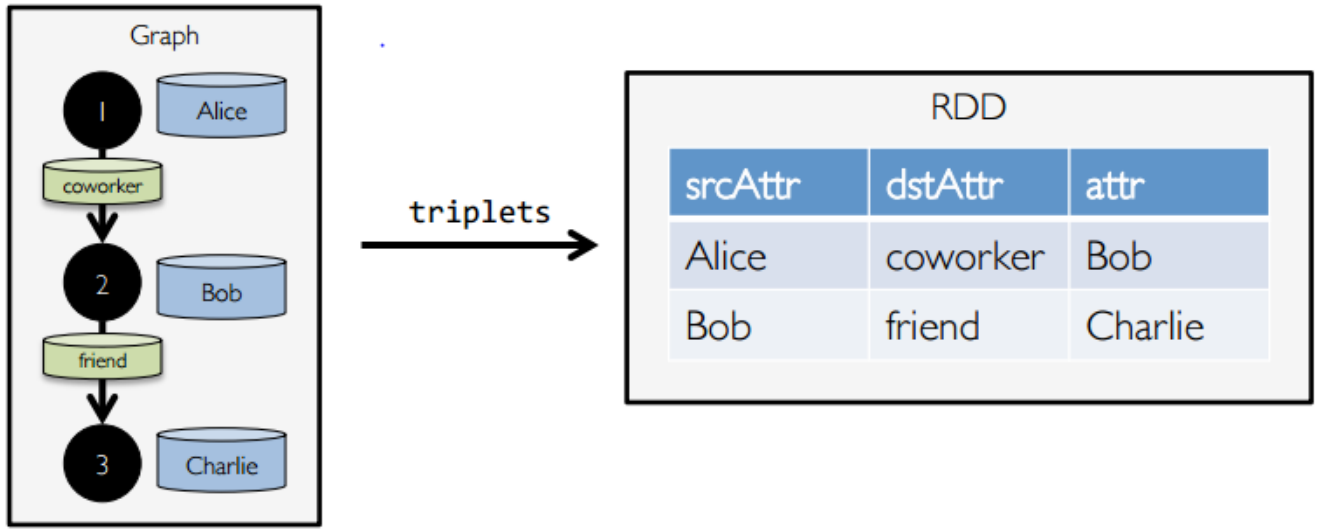
Build – in Algorithm

```
    def pageRank(tol: Double): Graph[Double, Double]
    def triangleCount(): Graph[Int, ED]
    def connectedComponents(): Graph[VertexId, ED]
    // ...and more: org.apache.spark.graphx.lib
}
```



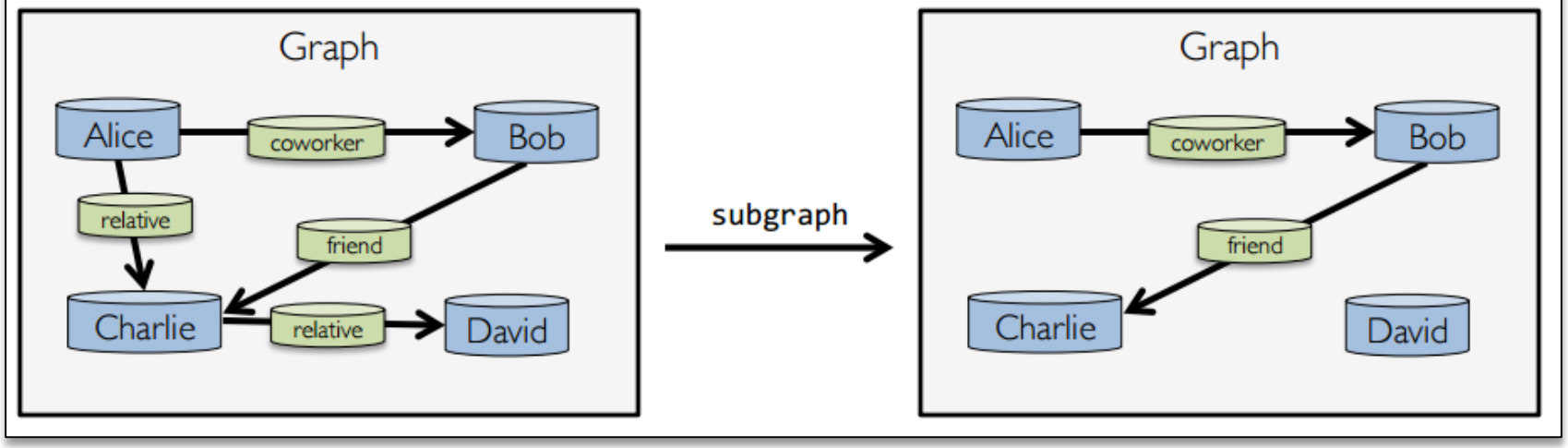
The Triplets View

```
class Graph[VD, ED] {
    def triplets: RDD[EdgeTriplet[VD, ED]]
}

class EdgeTriplet[VD, ED](
  val srcId: VertexId, val dstId: VertexId, val attr: ED,
  val srcAttr: VD, val dstAttr: VD)
```

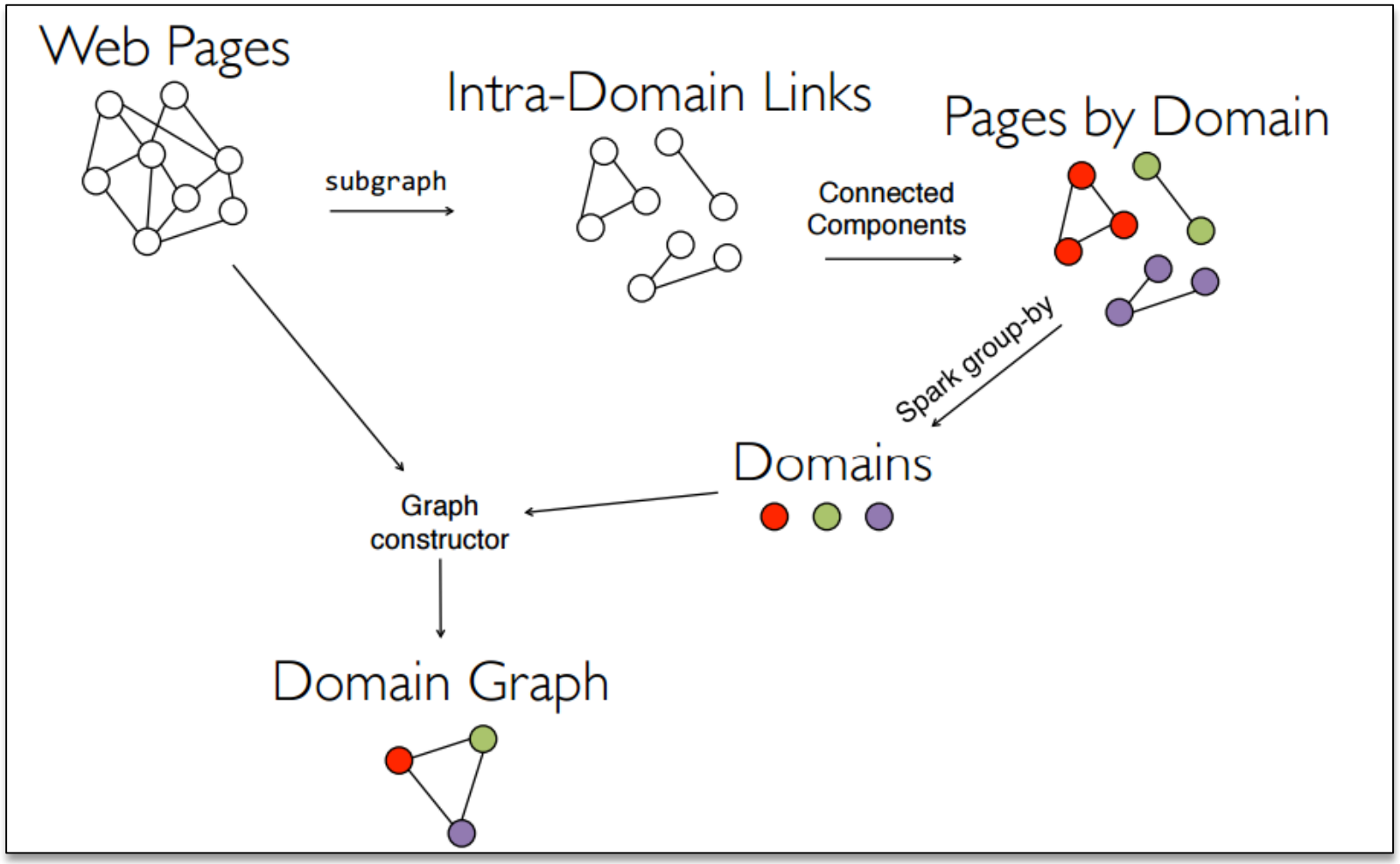


The Subgraph Transformation

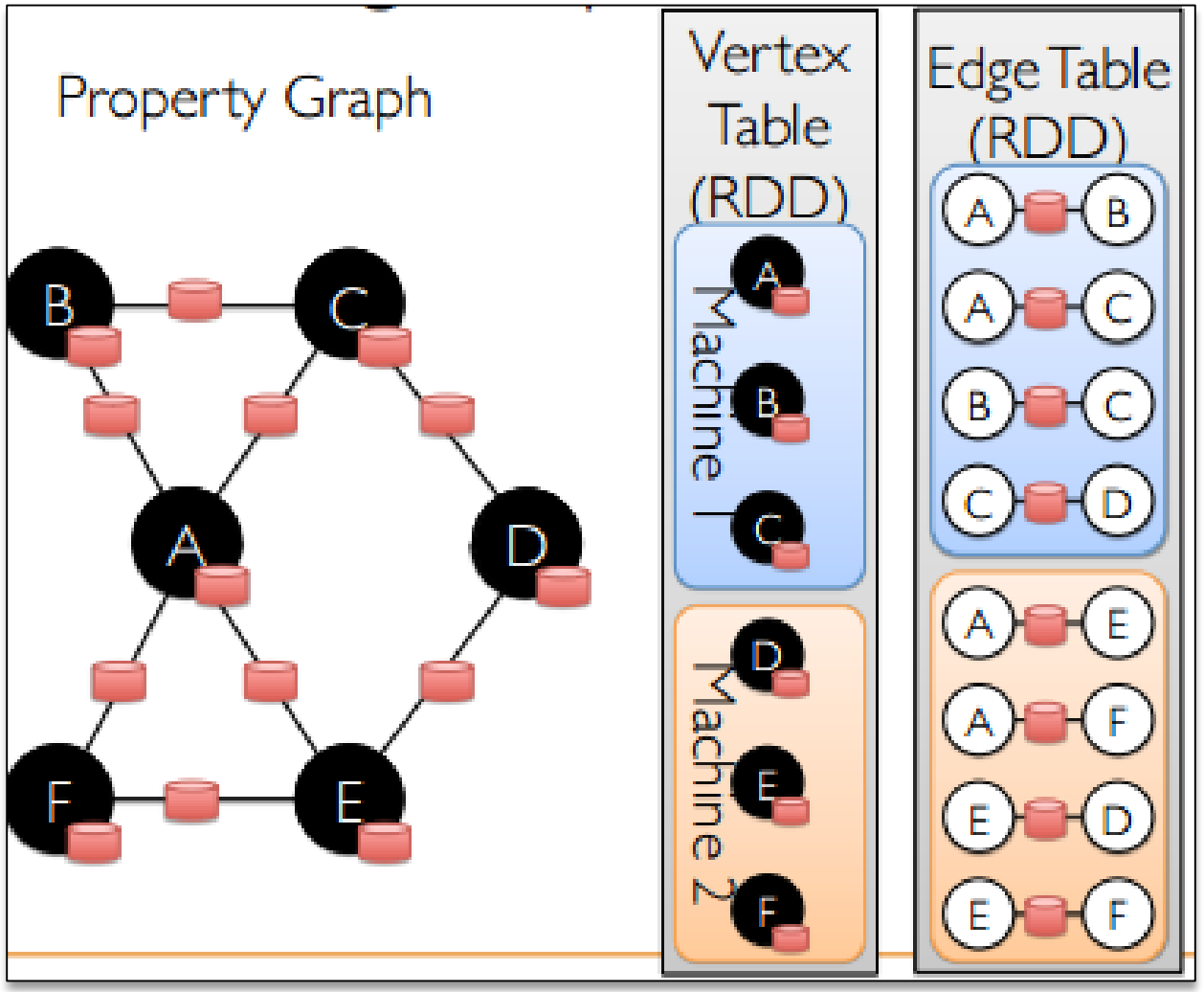
```
class Graph[VD, ED] {
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
}

graph.subgraph(epred = (edge) => edge.attr != "relative")
```



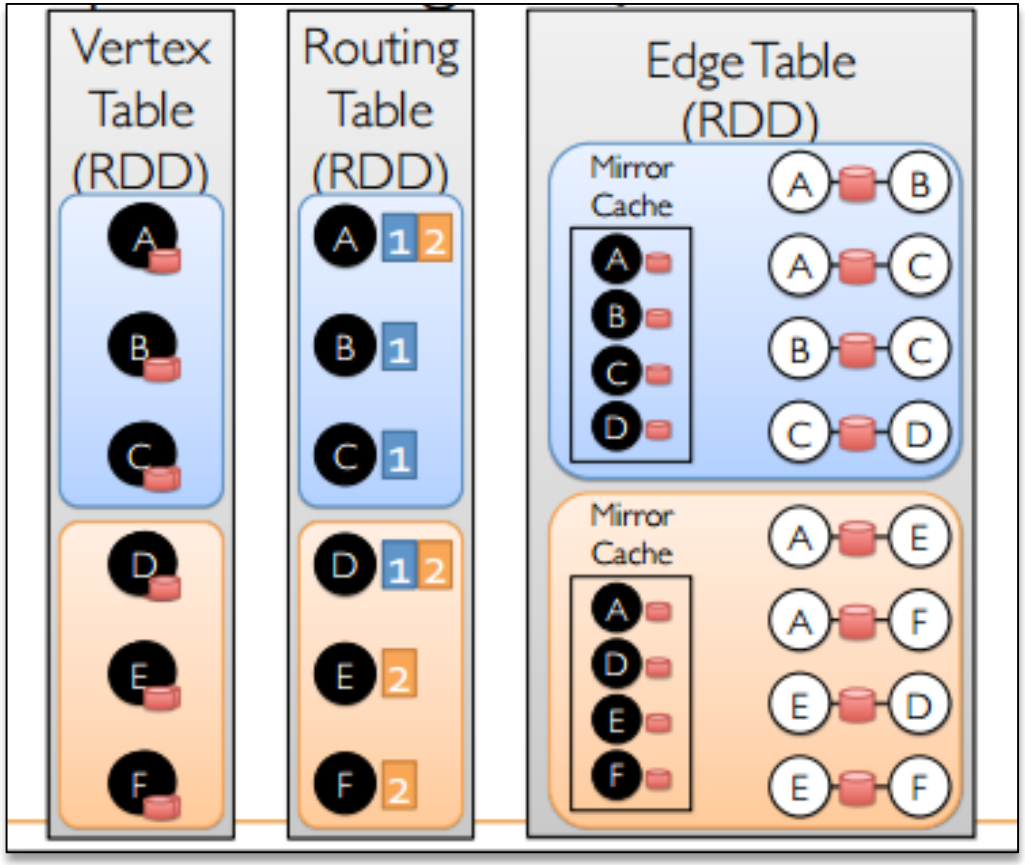
Example: Graph Coarsening



Sorting Graph as Tables



Implementing Triplets



References

Please refer this – in case you need some extensive approach… https://amplab.cs.berkeley.edu/wp-content/uploads/2014/02/graphx@strata2014_final.pdf

Questions

skillspeed
for the serious learner