**What this program does (overview)**

You have a tiny **web graph** with four “pages” A, B, C, D and directed links between them (A→B,C; B→C; C→A; D→C). The program:

1. Starts Spark (RDD API, OK on Spark 1.6).
2. Stores that graph as an RDD.
3. Initializes every page’s rank to 1.0.
4. Repeats the **PageRank update** ~10 times: each page **splits** its current rank equally among its outlinks; the destination pages **sum** all incoming contributions; then we apply **damping**: new = 0.15 + 0.85 \* (sum of contributions).
5. Prints the final ranks.

It’s a minimal, readable version of the algorithm—perfect for teaching.

**Line-by-line explanation**

# task13\_pagerank.py

# Simplified PageRank with PySpark RDD (works on Spark 1.6)

These comments describe the file and that it uses the classic RDD API (good for Spark 1.6).

from pyspark import SparkContext

Imports the entry point for RDD-based Spark. SparkContext lets your Python driver talk to the Spark engine and create RDDs/transformations.

# Step 1: Start Spark

sc = SparkContext(appName="Task13\_PageRank")

Creates a SparkContext named “Task13\_PageRank”.

* If you run via spark-submit, Spark picks up the cluster/master settings from your environment.
* From here on, sc is how you create RDDs (sc.parallelize, sc.textFile, etc.) and trigger jobs.

# Step 2: Define links between pages (graph edges)

# Each tuple: (page, [list of pages it links to])

links = sc.parallelize([

("A", ["B", "C"]),

("B", ["C"]),

("C", ["A"]),

("D", ["C"])

])

Builds the **graph** as an RDD.

* sc.parallelize([...]) takes a local Python list and distributes it across Spark executors as an RDD.
* Each record is (source\_page, [dest1, dest2, ...]).
  + A links to B and C
  + B links to C
  + C links to A
  + D links to C
* Type-wise: links is RDD[(str, list[str])].

# Step 3: Initialize all page ranks to 1.0

ranks = links.mapValues(lambda \_: 1.0)

Creates an initial ranks RDD that pairs **each page** (every key in links) with rank **1.0**.

* mapValues keeps the **keys** (A, B, C, D) and transforms the **values** only.
* We ignore the current value (the neighbor list) using the placeholder \_ and return 1.0.
* Result: ranks = { ("A",1.0), ("B",1.0), ("C",1.0), ("D",1.0) } (as an RDD).

# Step 4: Run PageRank iterations

for i in range(10): # 10 rounds

We’ll update the ranks **ten times**. More rounds → closer to convergence. Ten is fine for a demo.

# Each page gives its rank divided by number of outlinks to neighbors

contribs = links.join(ranks).flatMap(

lambda x: [(dest, x[1][1] / len(x[1][0])) for dest in x[1][0]]

)

This is the **core** of PageRank: compute **contributions** each page sends to its outlinks.

Break it down:

1. links.join(ranks)
   * Both links and ranks are keyed by page name.
   * A join yields: (page, (neighbor\_list, current\_rank)).
   * Examples after init:
     + ("A", (["B","C"], 1.0))
     + ("B", (["C"], 1.0)), etc.
   * This join is a **wide** transformation (it shuffles data by key).
2. .flatMap(lambda x: …)
   * For each joined record x:
     + x[0] = page (e.g., "A")
     + x[1][0] = neighbor list (e.g., ["B","C"])
     + x[1][1] = that page’s current rank (e.g., 1.0)
   * The list comprehension:  
     [(dest, x[1][1] / len(x[1][0])) for dest in x[1][0]]
     + For each dest in the neighbor list, emit one pair (dest, contribution)
     + contribution = current\_rank / outdegree  
       (equal split among all outlinks)
   * flatMap flattens all those small lists into one RDD stream:
     + From A (rank 1.0, 2 outlinks) → ("B", 0.5), ("C", 0.5)
     + From B (rank 1.0, 1 outlink) → ("C", 1.0)
     + From C (rank 1.0, 1 outlink) → ("A", 1.0)
     + From D (rank 1.0, 1 outlink) → ("C", 1.0)

So after this line, contribs is an RDD of **destination-keyed** contributions: RDD[(dest\_page, partial\_rank\_float)].

# New rank = sum of contributions (with damping factor 0.85)

ranks = contribs.reduceByKey(lambda a, b: a + b) \

.mapValues(lambda rank: 0.15 + 0.85 \* rank)

Two steps:

1. reduceByKey(lambda a, b: a + b)
   * Group contributions by **destination page** and sum them.
   * Example (first iteration from the contributions above):
     + A receives: 1.0 (from C) → sum = 1.0
     + B receives: 0.5 (from A) → sum = 0.5
     + C receives: 0.5 (from A) + 1.0 (from B) + 1.0 (from D) → sum = 2.5
     + D receives: nothing → **not present** in this RDD (important caveat below)
2. .mapValues(lambda rank: 0.15 + 0.85 \* rank)
   * Apply **damping**:
     + The constant **0.15** is the “teleportation” (random jump) part.
     + The factor **0.85** scales the summed contributions (the link-following part).
   * Using the example sums:
     + A: 0.15 + 0.85\*1.0 = 1.00
     + B: 0.15 + 0.85\*0.5 = 0.575
     + C: 0.15 + 0.85\*2.5 = 2.275
     + D: not present → **drops out** here (see note next).

**Important note (teachable limitation):**  
This minimal code **only** assigns a new rank to pages that **received at least one contribution** in that iteration. A page with **no inbound links** in a given round (like **D** in this graph) won’t appear in contribs, so it disappears from ranks. A production implementation preserves all nodes (and handles “dangling” pages that have no outlinks). For this tiny demo it’s fine, but if you want to keep D visible, see the “Keeping all pages each round” note at the end.

# Step 5: Collect and print final ranks

print("=== Final Page Ranks ===")

for page, rank in ranks.collect():

print(page, "has rank", round(rank, 3))

* ranks.collect() materializes the small result to the driver as a Python list of (page, rank) pairs.
* We loop and print each page’s rank, rounded to 3 decimals.
* In this graph you’ll see **C** largest (it gets votes from A, B, D), then A, then B; D will typically vanish with this minimal approach because it never receives contributions.

sc.stop()

Politely shuts down the SparkContext and frees cluster resources.

**One full iteration (worked example)**

Initial ranks: A=1.0, B=1.0, C=1.0, D=1.0.

* A (2 outlinks) → B:+0.5, C:+0.5
* B (1 outlink) → C:+1.0
* C (1 outlink) → A:+1.0
* D (1 outlink) → C:+1.0

Sum incoming:

* A: 1.0
* B: 0.5
* C: 0.5 + 1.0 + 1.0 = 2.5
* D: 0.0 (no one points to D)

Apply damping: new = 0.15 + 0.85 \* incoming

* A: 1.00
* B: 0.575
* C: 2.275
* D: (not present in contribs → disappears in this minimal code)

After several rounds, ranks stabilize with **C** dominant.

**Extra notes (optional, for completeness)**

**Why C dominates:** many pages funnel rank to C (A, B, D). B is a pure feeder to C; D also feeds C. C then gives all its rank to A, so A gets some boost too.

**Damping (0.85/0.15):** prevents “rank sink” problems and ensures every page gets some base traffic (the random-surfer teleports 15% of the time).

**Keeping all pages each round (if you want D to stay visible):**  
You can union with all pages and fill missing contributions with zero, then apply damping to **every page**:

all\_pages = links.keys().distinct() # A,B,C,D

contrib\_sums = contribs.reduceByKey(lambda a, b: a + b)

ranks = all\_pages.leftOuterJoin(contrib\_sums) \

.mapValues(lambda x: 0.15 + 0.85 \* (x[1] if x[1] is not None else 0.0))

This way, even pages with **no inbound links** (like D) still get the base 0.15 each iteration.