UNIVERSITÀ DEGLI STUDI DI MILANO

Ranking System Based on the PageRank

Fatemeh Amirian
34015A

Algorithms for Massive Data
Professor Malchiodi

Spring 2025

Abstract

This paper proposes a PageRank-based book ranking system on Amazon book review data. The dataset was preprocessed by a sequence of nonstandard choices, such as aggressive normalization of titles and custom mappings to normalize identifier inconsistencies. Books were connected in a graph based on shared reviewers, with no treatment for dangling nodes, since their structure rendered them useless. We have three PageRank implementations: a GraphFrames baseline, a pure Python version and one based on Spark RDD. A topic-sensitive version was also implemented using genre metadata. Results were comparable across methods, and the Spark implemention was tested for scalability.

Contents

1 Introduction 4

2 Dataset Description 4

3 Data Preprocessing 4

4 Graph Construction 6

5 Algorithm Implementation 7

5.1 Benchmark PageRank (Built-in) 7

5.2 Pure Python PageRank 8

5.3 Spark RDD Implementation 8

5.4 Comparison of Classic PageRank Implementations 9

5.5 Topic-Sensitive PageRank 11

5.5.1 Topic-Sensitive PageRank – Python Version 12

5.5.2 Topic-Sensitive PageRank – Spark RDD Version 13

5.6 Topic-Sensitive PageRank Results 13

6 Scalability Analysis 14

7 Results and Discussion 16

8 Limitations and Future Work 16

1 Introduction

A fundamental issue in large-scale data analysis is how to rank items in a graph. PageRank, which was created by Brin and Page to rank web sites, is still a fundamental solution because of its straightforward yet effective recursive principle: a node is significant if it is connected to other significant nodes. The concept was first used with the web graph, but it easily applies to other graph-structured domains as well.

Let P be the column-stochastic transition matrix of a directed graph with N nodes. The PageRank vector $\pi \in R^N$ is defined as the stationary distribution of a modified Markov chain that incorporates random teleportation. It satisfies the equation:

$$\pi = \alpha P\pi + (1 - \alpha)d$$

where $\alpha \in (0, 1)$ is the damping factor, and d is the teleportation distribution, typically uniform. The vector π is computed iteratively as:

$$\pi^{(k+1)} = \alpha P\pi^{(k)} + (1 - \alpha)d$$

We use this model on a network taken from the dataset of Amazon book reviews in this project. Our graph links books with a threshold amount of common reviewers instead of using links. In this reviewer co-occurrence network, the goal is to rank books according to their structural centrality using PageRank.

This report's remaining sections are laid out as follows: we start by outlining the data pretreatment procedures, which include a number of unconventional cleaning choices. After describing how the book graph was constructed, we provide a summary of the PageRank-based techniques that were used, including standard, Python-based, Spark-based, and topic-sensitive approaches. An assessment of the findings, scalability observations, and a discussion of the research's limits and future directions wrap up the paper.

2 Dataset Description

The dataset used in this project was obtained from Kaggle[1] and consists of two CSV files. About three million user reviews of books, including user ID, book title, and rating score, are included in the first. The books' own metadata, such as titles and related genre classifications, are included in the second file.

We created a graph of co-reviewed books using the review data, which served as the foundation for most of our study. Only book categories were mapped using the metadata file, which was then used as input for the topic-sensitive PageRank implementation.

3 Data Preprocessing

The raw review dataset was first loaded into Spark, and initial filtering steps were applied. All rows with missing `user_id` values were dropped. Missing `Title` values were replaced with the `"unknown"` to preserve structural. Since a key assumption in our model is that each user contributes at most one review per book, we removed duplications from all (`user_id`, `book_id`) pairs, keeping only a single review when multiple were found.

[1]`https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews`

After that, we used this cleaned version to run the complete pipeline and calculated a preliminary PageRank on the created graph. A crucial issue was revealed by the sorted list of books that emerged from the process: the top 20 results included books that were semantically identical but had slightly different titles or book IDs. For example:

- "The Catcher in the Rye [Audiobook] [Cd] [Unabridge" vs. "THE CATCHER IN THE RYE"
- "Fahrenheit 451" vs. "Fahrenheit 451 (Cascades S.)"
- "To Kill a Mocking Bird" vs. "To Kill a Mockingbird"

In other instances, the same book has appeared more than once with different book IDs but the same title. The graph's structure was warped by this duplication, which also made well-known titles seem more significant. It was unclear if this discrepancy resulted from upstream processing problems or the original dataset. A graph that did not represent our desired semantics a user's relationship with a book, not with its format or edition, was the eventual outcome, regardless of the cause. To address this, we introduced a custom normalization procedure for titles:

- All titles were converted to lowercase to eliminate case-based distinctions.
- Bracketed and parenthetical substrings—often used to denote editions, formats, or series labels—were stripped.
- Known problematic characters, typographic anomalies, and irregular whitespace were standardized.

However, we encountered a second-order challenge: some titles legitimately contained meaningful parentheticals (e.g., to denote authors of poems or specific versions that should remain distinct). To handle this, we created a list of exceptions titles that should not be normalized further. For example:

- `poems (Dante Alighieri)`
- `poems (Sylvia Plath)`

These were explicitly excluded from transformations, as collapsing them would undermine semantic distinctions relevant to genre-based ranking.

Once titles were normalized, many-to-one relationships emerged between original titles and normalized versions. To ensure consistent treatment across the graph, we remapped all book titles and user identifiers to unique integer IDs. This improved performance and ensured that each unique (normalized) book was represented only once. Duplicate rows arising from normalization were again removed after remapping.

At the end of preprocessing, the final dataset consisted of:

- 198,919 unique books
- 1,008,972 unique users
- An average of 1.91 reviews per user

The distribution of review counts per user is summarized below:

Min	**Q1**	**Median**	**Q3**	**Max**
1	1.0	1.0	1.0	5243

These decisions were not strictly standard. They reflect a design choice to prioritize semantic identity over literal data fidelity, which we considered critical for meaningful graph-based ranking. We include below the original top 20 books obtained from the raw data using implementations. This table illustrates the problem clearly, showing redundant popular titles:

Table 1: Top 20 Books – PageRank

#	**Title**
1	The Catcher in the Rye [Audiobook] [Cd] [Unabridged]
2	The Great Gatsby (Leading English literature library)
3	Harry Potter and The Sorcerer's Stone
4	Pride and Prejudice
5	1984
6	To Kill a Mocking Bird
7	1st to Die: A Novel
8	Pride and Prejudice
9	Manhattan Stories From the Heart of a Great City
10	The Catcher in the Rye
11	Fahrenheit 451 (Cascades S.)
12	The Great Gatsby
13	To Kill a Mockingbird
14	To Kill a Mockingbird
15	Jane Eyre (Signet classics)
16	Fahrenheit 451
17	The Hobbit
18	1984
19	The Hobbit
20	Wuthering Heights (Signet classics)

4 Graph Construction

The aim of graph construction in the project is the modeling of book relationships from a user behavior perspective. That is, two books are connected when they have at least some number of reviewers in common. This same-reviewer threshold constitutes a content-independent, behavioral concept of similarity.

PageRank does expect a directed graph, though. Our data was not directionally inherent (i.e., one book citing another), so we intentionally created an undirected graph and transformed it into bidirectional representation by copying each edge in both directions. This allows PageRank compatibility while still maintaining the symmetric quality of the reviewer-based relationship.

Construction is done using Apache Spark and GraphFrames and consists of the following primary steps:

1. **User–Book Pairs:** Extract all unique (`user_id`, `book_id`) combinations.
2. **Book–Book Pairs:** Perform a self-join on user ID to identify all pairs of books reviewed by the same user. Retain only book pairs where the first book ID is less than the second to prevent redundancy.
3. **Edge Filtering:** Group the resulting book pairs and count how many users have reviewed both. Keep only those pairs that meet a minimum threshold (e.g., 2 shared users).
4. **Bidirectionality:** Duplicate each edge in reverse to simulate undirected behavior within a directed graph framework.
5. **Vertex Set:** Construct the set of all books involved in any edge by unifying all sources and destinations.
6. **Purpose-Specific Output:** Return graph representations tailored to each PageRank implementation:
    - A `GraphFrame` for the built-in version
    - An RDD for the Spark-based version
    - Local Python lists for the pure Python version

The final graph contains:

- **Number of vertices:** 47,488
- **Number of directed edges:** 4,084,494
- **Unique book–book connections:** 2,042,247

5 Algorithm Implementation

5.1 Benchmark PageRank (Built-in)

Establishing a reliable reference point was crucial for validating our results, even though the main goal of this project was to implement PageRank from scratch. Because of this, we benchmarked against the GraphFrames library's built-in PageRank function.

The GraphFrames implementation accepts a GraphFrame object as input and provides two execution modes:

- A fixed number of iterations, where the algorithm updates node scores a specified number of times regardless of convergence.
- A convergence-based mode, where iterations continue until the change in PageRank values falls below a specified threshold.

Setting a maximum number of iterations was the first strategy we employed in our experiment. This guaranteed consistent comparison with our manual implementations and provided deterministic runtimes. In accordance with accepted literature, the damping factor, also referred to as the teleportation or reset probability, was set at 0.15.

Each vertex in the graph was given a `pagerank` value as part of the algorithm's output. In order to determine whether our Python and Spark RDD implementations met expectations, these scores were subsequently used as ground truth.[2]

5.2 Pure Python PageRank

As a controlled experiment, we implemented a PageRank algorithm in pure Python, despite the fact that large-scale graphs typically require distributed computing to be processed efficiently. The objective was to compare the output to the built-in benchmark, validate behaviour on an actual dataset, and enhance our comprehension of the algorithmic mechanics. Even though Python has built-in limitations when it comes to managing large graphs, this implementation worked surprisingly well on our dataset when run locally.

The procedure follows the standard iterative approach:

Algorithm 1 PageRank via Power Iteration (Python Implementation)

```
1: **Input:** Set of pages $P$, list of links $L$, damping factor $\beta$, max iterations $M$, tolerance $\varepsilon$
2: **Output:** Final rank scores $R$, number of iterations
3: Initialize $R[p] \leftarrow \frac{1}{|P|}$ for every page $p \in P$
4: Build an adjacency list from the edge list $L$
5: **for** each iteration from 1 to $M$ **do**
6:     Set new ranks $R_{\text{new}}[p] \leftarrow \frac{1-\beta}{|P|}$ for all $p \in P$
7:     **for** each page $p$ with outgoing links **do**
8:         Divide $R[p]$ equally among all neighbors
9:         Add $\beta \cdot \frac{R[p]}{\#\text{neighbors}}$ to each neighbor's $R_{\text{new}}$
10:    **end for**
11:    Compute diff $= R_{\text{new}} - R_2$
12:    **if** diff $< \varepsilon$ **then**
13:        Break the loop
14:    **end if**
15:    Update $R \leftarrow R_{\text{new}}$
16: **end for**
17: **return** Sorted ranks and iteration count
```

When the L2 distance between consecutive PageRank vectors is less than the designated tolerance, this method converges. Numerical stability was confirmed when we noticed that the total of all final rank values was nearly 1. Due to the graph's modest size following preprocessing (roughly 47,000 nodes and 2 million unique edges), this implementation was computationally tractable on the project's graph, despite not being appropriate for very large datasets.

5.3 Spark RDD Implementation

Real-world datasets can grow significantly larger than what local memory and single-threaded execution can manage, even though our Python implementation of PageRank

[2]Documentation source:`https://graphframes.io/docs/_site/user-guide.html#pagerank`

was helpful for development and small-scale testing. We used Spark's Resilient Distributed Dataset (RDD) API to implement a distributed version of the PageRank algorithm in order to account for possible scalability.

RDDs distribute computation and data throughout a cluster, enabling the parallel processing of large datasets. Because of this, Spark is a natural fit for iterative graph algorithms like PageRank, which call for repeated message-passing and aggregation processes.

The RDD-based algorithm's logic is very similar to the original formulation, but some changes were required to effectively support distributed execution. These consist of broadcasting the current rank vector, calculating contributions using Spark transformations, and managing rank vector normalisation by hand at every stage. Consequently, this implementation adheres to the same mathematical principles but is not exactly the same as the Python version. The following algorithm outlines the procedure used:

Algorithm 2 PageRank with Spark RDD

```
 1: **Input:** Edge list $L$, number of nodes $N$, damping factor $\beta$, max iterations $M$, tolerance $\varepsilon$
 2: **Output:** Final rank scores and iteration count
 3: Initialize rank $R[p] \leftarrow \frac{1}{N}$ for all $p$
 4: Construct adjacency list as $A[p] = \{\text{neighbors of } p\}$
 5: **for** each iteration from 1 to $M$ **do**
 6:     Broadcast current ranks $R$
 7:     For each $p$, compute contributions to neighbors $\frac{R[p]}{\text{out-degree}}$
 8:     Use reduceByKey to sum incoming contributions
 9:     Update each rank $R[p] \leftarrow \frac{1-\beta}{N} + \beta \cdot \text{incoming}$
10:     Compute L2 norm between new and old rank vectors
11:     **if** change $< \varepsilon$ **then break**
12:
13:         **return** Sorted rank list and number of iterations
```

5.4 Comparison of Classic PageRank Implementations

We evaluated the accuracy of our implementations by analysing the top 20 books produced by three PageRank algorithms: the Spark RDD version, our custom Python version, and the built-in GraphFrames method (used as benchmark). Every method was executed using the same graph and the same initialisation, convergence threshold, and damping factor.

A proper implementation should yield rankings that are very similar given the deterministic inputs. The ranked books, particularly the top results, should actually match in both order and score distribution if everything is done correctly (with tolerance because of scale normalisation).

Below we include the top 20 ranked books as generated by each method:

Table 2: Top 20 Books – PageRank

Title	**Genre**	**Avg Rating**	**PageRank**
harry potter and the sorcerer's stone	Juvenile Fiction	4.69	56.752445
blink: the power of thinking without ...	Business & Economics	3.70	41.615154
the catcher in the rye	Young Adult Fiction	3.92	41.596783
five people you meet in heaven	Fiction	4.16	39.573220
john adams	Electronic books	4.68	38.860889
night	Juvenile Fiction	4.55	35.996820
the great gatsby	Fiction	4.15	35.926736
guns, germs, and steel: the fates of ...	History	4.01	35.807039
the tipping point: how little things ...	Reference	4.12	35.586899
great gatsby	American Dream	4.16	35.580328
manhattan stories from the heart of a...	Manhattan (New York, N.Y.)	4.16	34.914868
the hobbit there and back again	Adventure stories	4.68	34.517521
the hobbit	Juvenile Fiction	4.68	34.517521
the hobbit or there and back again	Juvenile Fiction	4.68	34.491207
the hobbitt, or there and back again;...	Fiction	4.68	34.371543
fahrenheit 451	Comics & Graphic Novels	4.23	33.929131
who moved my cheese? an-amazing way t...	Business & Economics	3.41	33.608993
to kill a mocking bird	Drama	4.60	33.179958
harper lee's to kill a mockingbird	Juvenile Nonfiction	4.61	33.179958
to kill a mockingbird	Performing Arts	4.59	33.179958

Table 3: Top 20 Books – Custom PageRank (35 Iterations)

Title	**Genre**	**Avg Rating**	**PageRank**
harry potter and the sorcerer's stone	Juvenile Fiction	4.69	0.001195
blink: the power of thinking without ...	Business & Economics	3.70	0.000876
the catcher in the rye	Young Adult Fiction	3.92	0.000876
five people you meet in heaven	Fiction	4.16	0.000833
john adams	Electronic books	4.68	0.000818
night	Juvenile Fiction	4.55	0.000758
the great gatsby	Fiction	4.15	0.000757
guns, germs, and steel: the fates of ...	History	4.01	0.000754
the tipping point: how little things ...	Reference	4.12	0.000749
great gatsby	American Dream	4.16	0.000749
manhattan stories from the heart of a...	Manhattan (New York, N.Y.)	4.16	0.000735
the hobbit	Juvenile Fiction	4.68	0.000727
the hobbit there and back again	Adventure stories	4.68	0.000727
the hobbit or there and back again	Juvenile Fiction	4.68	0.000726
the hobbitt, or there and back again;...	Fiction	4.68	0.000724
fahrenheit 451	Comics & Graphic Novels	4.23	0.000715
who moved my cheese? an-amazing way t...	Business & Economics	3.41	0.000708
harper lee's to kill a mockingbird	Juvenile Nonfiction	4.61	0.000699
to kill a mockingbird	Performing Arts	4.59	0.000699
to kill a mocking bird	Drama	4.60	0.000699

Table 4: Top 20 Books – RDD PageRank (35 Iterations)

Title	**Genre**	**Avg Rating**	**PageRank**
harry potter and the sorcerer's stone	Juvenile Fiction	4.69	0.001195
blink: the power of thinking without ...	Business & Economics	3.70	0.000876
the catcher in the rye	Young Adult Fiction	3.92	0.000876
five people you meet in heaven	Fiction	4.16	0.000833
john adams	Electronic books	4.68	0.000818
night	Juvenile Fiction	4.55	0.000758
the great gatsby	Fiction	4.15	0.000757
guns, germs, and steel: the fates of ...	History	4.01	0.000754
the tipping point: how little things ...	Reference	4.12	0.000749
great gatsby	American Dream	4.16	0.000749
manhattan stories from the heart of a...	Manhattan (New York, N.Y.)	4.16	0.000735
the hobbit	Juvenile Fiction	4.68	0.000727
the hobbit there and back again	Adventure stories	4.68	0.000727
the hobbit or there and back again	Juvenile Fiction	4.68	0.000726
the hobbitt, or there and back again;...	Fiction	4.68	0.000724
fahrenheit 451	Comics & Graphic Novels	4.23	0.000715
who moved my cheese? an-amazing way t...	Business & Economics	3.41	0.000708
to kill a mocking bird	Drama	4.60	0.000699
harper lee's to kill a mockingbird	Juvenile Nonfiction	4.61	0.000699
to kill a mockingbird	Performing Arts	4.59	0.000699

As we can see:

- All three outputs share the same 20 books.
- The top five rankings are identical across methods.
- Differences are minimal and only appear in very low-rank reordering.

This confirms the correctness of our custom implementations. Both the Python and RDD versions match the structure, behavior, and output of the built-in benchmark, indicating that they are mathematically consistent and correctly implemented.

5.5 Topic-Sensitive PageRank

When modelling teleportation, the standard PageRank algorithm treats all nodes equally, giving each page an equal chance of being visited at random. *Topic-Sensitive PageRank* (TSPR), on the other hand, introduces bias by favouring nodes that are pertinent to a specific topic or category.

In this instance, we defined topic relevance using book *genre* information. We find books labelled with a list of target genres (such as "Fiction" or "History") and focus the teleportation probability on those books. As a result, the random surfer is more likely to pick up a book in the designated genre.

In order to accommodate topic-sensitive teleportation, we modified the teleportation distribution at initialisation and updated it appropriately at each iteration in both our Spark RDD and pure Python implementations.

5.5.1 Topic-Sensitive PageRank – Python Version

Algorithm 3 Topic-Sensitive PageRank in Python

```
 1: Input: Pages $P$, Links $L$, Genre Map $G$, Target Genres $T$, damping $\beta$, max iterations
    $M$, tolerance $\varepsilon$
 2: Initialize $R[p] \leftarrow 1/|P|$ for all $p \in P$
 3: Build adjacency list from $L$
 4: if no target genres $T$ then
 5:     Use uniform teleportation
 6: else
 7:     Identify topic-relevant pages $P_T \subset P$
 8:     Assign teleport weight $\frac{1-\beta}{|P_T|}$ to $p \in P_T$; 0 otherwise
 9: end if
10: for iteration = 1 to $M$ do
11:     Set $R_{\text{new}}[p] \leftarrow \text{teleport}[p]$ for all $p$
12:     for each page $p$ with out-links do
13:         Distribute $\frac{R[p]}{\#\text{neighbors}}$ to each neighbor
14:         Add $\beta$ share to $R_{\text{new}}$
15:     end for
16:     Compute L2 norm between $R$ and $R_{\text{new}}$
17:     if difference $< \varepsilon$ then break
18:         Update $R \leftarrow R_{\text{new}}$
19:
20:         return Sorted ranks and iteration count
```

5.5.2 Topic-Sensitive PageRank – Spark RDD Version

Algorithm 4 Topic-Sensitive PageRank with Spark RDD

```
1: Input: Edge list $L$, Node count $N$, Genre Map $G$, Target Genres $T$, damping $\beta$,
   max iterations $M$, tolerance $\varepsilon$
2: Extract all distinct nodes
3: Initialize rank $R[n] \leftarrow 1/N$
4: Build adjacency list as $A[n] = \{\text{neighbors of } n\}$
5: if no target genres then
6:     Assign uniform teleport probability $\frac{1-\beta}{N}$
7: else
8:     Identify topic nodes $N_T \subset N$
9:     Assign $\frac{1-\beta}{|N_T|}$ to nodes in $N_T$; 0 elsewhere
10: end if
11: for iteration = 1 to $M$ do
12:     Broadcast $R$
13:     For each node $n$, distribute contribution to neighbors
14:     Aggregate contributions with reduceByKey
15:     Update each rank $R[n] = \text{teleport}[n] + \beta \cdot \text{contribution}$
16:     Compute L2 norm between old and new ranks
17:     if difference $< \varepsilon$ then break
18:         Update $R \leftarrow R_{\text{new}}$
19:
20:         return Sorted ranks and iteration count
```

5.6 Topic-Sensitive PageRank Results

Topic-sensitive PageRank variants lack a standard built-in benchmark for direct comparison, in contrast to classic PageRank. In order to assess our implementations, we applied the topic-sensitive PageRank versions in Spark RDD and Python to a particular category, in this case `Fiction`.

The genre-focused teleportation vector produced a ranked list that was more heavily dominated by fiction books, as was to be expected. The majority of the top-ranked items are either clearly marked as fiction or fall under narrative-driven genres, which usually share semantic similarities with fiction.

The top 20 outcomes from our two topic-sensitive algorithms, each of which was run independently, are shown below. Both converged after 40 iterations with identical parameters.

Table 5: Top 20 Fiction Books – Topic-Sensitive PageRank (Python)

Title	**Genre**	**Avg Rating**	**PageRank**
harry potter and the sorcerer's stone	Juvenile Fiction	4.69	0.001366
five people you meet in heaven	Fiction	4.16	0.000905
the catcher in the rye	Young Adult Fiction	3.92	0.000893
the hobbit	Juvenile Fiction	4.68	0.000892
the hobbit or there and back again	Juvenile Fiction	4.68	0.000891
the hobbitt, or there and back again;...	Fiction	4.68	0.000887
the hobbit there and back again	Adventure stories	4.68	0.000880
prey	Predation (Biology)	3.50	0.000873
the hobbit; or, there and back again	Unknown	4.69	0.000827
one for the money	Fiction	4.39	0.000778
harper lee's to kill a mockingbird	Juvenile Nonfiction	4.61	0.000774
to kill a mockingbird	Performing Arts	4.59	0.000774
to kill a mocking bird	Drama	4.60	0.000774
the great gatsby	Fiction	4.15	0.000766
great gatsby	American Dream	4.16	0.000746
harry potter & the prisoner of azkaban	Literary Criticism	4.77	0.000744
manhattan stories from the heart of a...	Manhattan (New York, N.Y.)	4.16	0.000737
shutter island	Fiction	4.11	0.000728
harry potter and the chamber of secrets	Juvenile Fiction	4.67	0.000724
fahrenheit 451	Comics & Graphic Novels	4.23	0.000722

Table 6: Top 20 Fiction Books – Topic-Sensitive PageRank (RDD)

Title	**Genre**	**Avg Rating**	**PageRank**
harry potter and the sorcerer's stone	Juvenile Fiction	4.69	0.001366
five people you meet in heaven	Fiction	4.16	0.000905
the catcher in the rye	Young Adult Fiction	3.92	0.000893
the hobbit	Juvenile Fiction	4.68	0.000892
the hobbit or there and back again	Juvenile Fiction	4.68	0.000891
the hobbitt, or there and back again;...	Fiction	4.68	0.000887
the hobbit there and back again	Adventure stories	4.68	0.000880
prey	Predation (Biology)	3.50	0.000873
the hobbit; or, there and back again	Unknown	4.69	0.000827
one for the money	Fiction	4.39	0.000778
harper lee's to kill a mockingbird	Juvenile Nonfiction	4.61	0.000774
to kill a mocking bird	Drama	4.60	0.000774
to kill a mockingbird	Performing Arts	4.59	0.000774
the great gatsby	Fiction	4.15	0.000766
great gatsby	American Dream	4.16	0.000746
harry potter & the prisoner of azkaban	Literary Criticism	4.77	0.000744
manhattan stories from the heart of a...	Manhattan (New York, N.Y.)	4.16	0.000737
shutter island	Fiction	4.11	0.000728
harry potter and the chamber of secrets	Juvenile Fiction	4.67	0.000724
fahrenheit 451	Comics & Graphic Novels	4.23	0.000722

6 Scalability Analysis

The cost of each PageRank iteration should ideally increase approximately linearly with the graph's edge count. Assuming constant parallelism and minimal overhead, a perfect implementation should display per-iteration runtimes that increase proportionately with graph size because the algorithm's primary computation involves propagating rank values across edges.

We created a testing script that samples several independent subsets of the review dataset, each selected using a different random seed, in order to assess how close we are to this behaviour. Each of these subsets, which vary from 10 to 100 percent of the complete data, is processed in a separate Spark session to prevent execution artefacts, memory reuse, and residual caching. From each sample, the script builds a user-book co-review graph, calculates PageRank using our RDD-based implementation, and logs important metrics like runtime, edge count, and iteration count. A CSV file containing all of the results was saved for later review.

Importantly, Spark's local[*] mode was used to simulate parallel execution across available CPU cores, and the entire experiment was conducted on a local machine. To promote parallelism, the RDD operations—such as the creation of the co-review graph and the PageRank calculation—were specifically divided into eight partitions. However, considering the local execution context, it's unclear how much this affected the outcomes.

The findings did not demonstrate a clear linear correlation between per-iteration time and graph size. Instead, even though the number of edges increased from a few thousand to over four million, the time per iteration stayed relatively constant throughout the range, rising only from about 4.9 to 6.1 seconds. This could be caused by a number of factors:

Spark's fixed overhead (task scheduling, shuffle planning, DAG execution, etc.) may take up most of the runtime at small scales, leaving little time for the PageRank computation itself to appear in the metrics.

Some samples may result in more sparse or fragmented graphs due to their structural independence, which could have an impact on workload distribution or convergence behaviour during execution.

Scalability constraints or bottlenecks that would appear in a distributed environment might not be fully revealed by local mode execution on a single machine.

In conclusion, the time-per-iteration behaviour is flatter than anticipated, even though the algorithm operates reliably and remains stable with scale. These findings imply that Spark overhead and local execution effects might mask the algorithm's actual scaling properties in this particular configuration. To more definitively evaluate real-world scalability, more testing would be required, particularly on a distributed cluster.

Table 7: PageRank Scalability Results

Fraction	Sample Reviews	Nodes	Edges	Iterations	Total Time (s)	Time/Iter (s)	Converged	Rank Sum
0.1	192 665	1514	3608	55	273.453	4.971 879	TRUE	1
0.2	385 982	5426	31 684	47	229.445	4.881 805	TRUE	1
0.3	579 435	10 373	112 376	45	230.104	5.113 412	TRUE	1
0.4	771 544	15 768	284 874	42	234.881	5.592 416	TRUE	1
0.6	1 159 115	26 563	959 848	39	218.890	5.612 566	TRUE	1
0.8	1 545 417	37 383	2 203 660	37	209.039	5.649 702	TRUE	1
1.0	1 931 845	47 488	4 084 494	35	212.488	6.071 093	TRUE	1

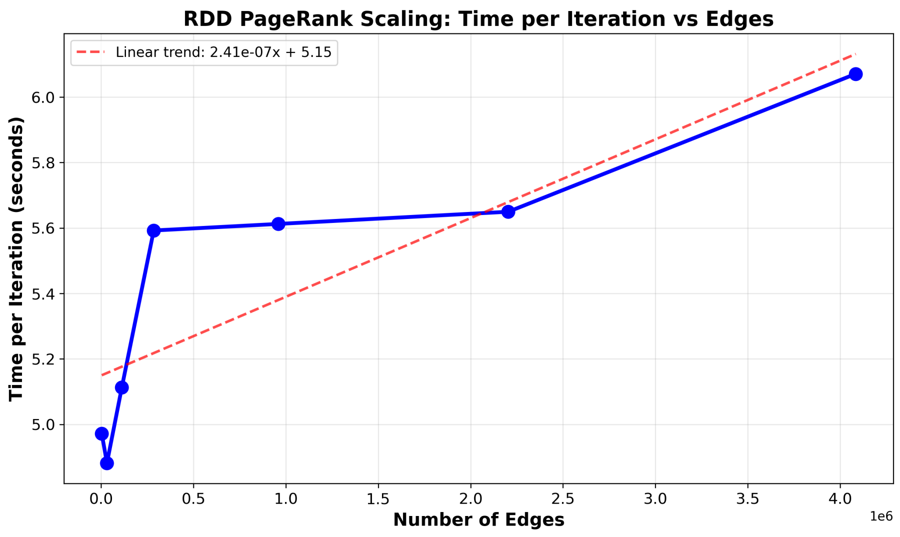

Figure 1: RDD PageRank Scaling: Time per Iteration vs. Number of Edges

7 Results and Discussion

Classic literature regularly ranked at the top of the PageRank algorithm's list of highly regarded books, which was to be expected since these works typically receive the most user interactions and co-reviews. This demonstrates how well PageRank represents popularity and influence in the context of book review networks. Regarding performance, the RDD-based implementation worked well and managed big graphs with respectable efficiency, though complete scalability is still a work in progress. The outcomes show the validity of the ranking model as well as the system's scalability with improved engineering.

8 Limitations and Future Work

This project ran into a number of real-world limitations, particularly with regard to computing power. We were forced to work on a subset of the dataset because of the limitations of using Apache Spark on Google Colab. Even though Spark was run in local mode, this mode is only appropriate for prototyping and relatively small datasets by nature. Only in a distributed cluster environment—which was outside the scope of our setup do Spark's true scalability and efficiency become apparent.

Building a complete PageRank implementation with DataFrames rather than RDDs or Python lists was one of our initial objectives. Although a partial prototype was completed, the performance was inadequate, most likely as a result of PageRank's iterative nature and the overhead of DataFrame operations. This avenue requires more attention given more time and improved exploration. We also tried a hybrid strategy that combined Spark RDDs with pure Python logic. For consistency and improved interpretability, we ultimately chose to stick with pure RDD and Python implementations, even though they were technically feasible.

We investigated substituting other distance metrics, such as Manhattan and Chebyshev distances, for the L2 norm in order to achieve faster convergence. Motivated by the findings in [4], these alternatives produced comparable outcomes with only slight speed gains.

Our pipeline was designed to be highly modular. Every algorithm is encapsulated in a callable function, allowing easy experimentation with parameters such as damping factor, convergence threshold, and the edge threshold used during graph construction.

There are still a number of interesting methods for further research. Using the timestamps in our dataset, we first would like to investigate time-aware PageRank variations by emphasising more recent reviews by adding an exponential decay factor. Second, we envision a weighted graph model where the weights of the edges are determined by the strength or similarity of user reviews. For example, we would give more weight to books that have 5-star reviews in common than to books that don't.

We also acknowledge that we could greatly enhance our preprocessing, especially the standardisation of book titles. A more dependable graph might result from adding book authors or other metadata from the books file, as title matching between records was not perfect. Creating alternative graph formulations to rank entities other than books, like authors or genres, would be another long-term objective.

Lastly, even though our RDD-based implementation has been thoroughly debugged and optimised, it is still feasible and desirable to achieve even greater efficiency and scalability.

References

[1] David F. Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015. `https://arxiv.org/abs/1407.5107`.

[2] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2nd edition, 2014. `http://www.mmds.org`.

[3] Jing Ma et al. A comparison of distance metrics in pagerank variants, 2021. `https://arxiv.org/abs/2108.02997`.

[4] Subhajit Sahu, Kishore Kothapalli, and Dip Sankar Banerjee. Adjusting pagerank parameters and comparing results. *arXiv preprint arXiv:2108.02997*, 2021.