Part 1: Word Count and Data Analysis with PySpark:

1. Basic Word Count Implementation:

- I have taken the book **"PRIDE AND PREJUDICE"** from Project Gutenberg website.
- Below is the list of words with their occurrence count in descending order.

word	count
the	4811
to	4383
of	3955
and	3740
her	2258
a	2079
in	2038
94	1899
was	1870
i	1794
she	1710
that	1561
it	1542
not	1508
you	1323
he	1318
be	1279
his	1279
as	1223
had	1179

I chose "**Pride and Prejudice**" for the following reasons :

1. **Popularity**: It's consistently one of the most downloaded books on Project Gutenberg, ensuring a rich dataset for analysis1.
2. **Literary Significance**: As a classic of English literature, analyzing its word usage can provide insights into 19th-century writing styles and themes.
3. **Accessibility**: The book is widely available and familiar to many readers, making the analysis results more relatable and interesting.
4. **Familiarity:** I have read the book multiple times before.

Analysis of Word Count Results

Top Words- The most common words are typical English stopwords, such as "the" (4811 occurrences), "to" (4383), "of" (3955), "and" (3740), and so on. These words are used frequently in any English text but do not carry significant meaning by themselves.

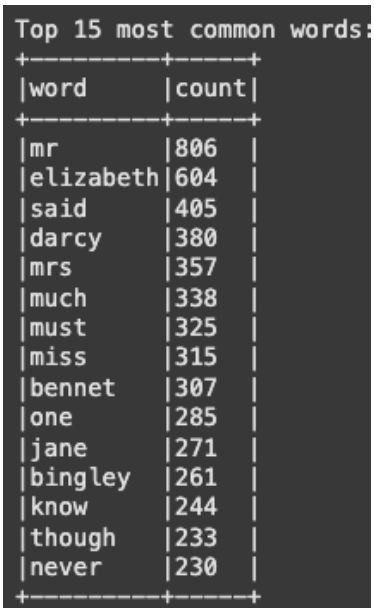
Pronouns like "her" (2258), "she" (1710), "his" (1279) also appear frequently, indicating a character's actions or relationships, which is characteristic of novels.

The number **94**(1899) appearing so many times in the results is likely due to a formatting issue or specific number used in the text.

Insights : This result is typical for most novels unless stopwords are explicitly filtered out. These common words dominate the word frequencies and can overshadow more meaningful terms unless preprocessing steps like stopword removal are applied.

2. Extended Word Count:

After removing all the stop words, punctuations and converting the text to lower case, the top 15 most used words are :

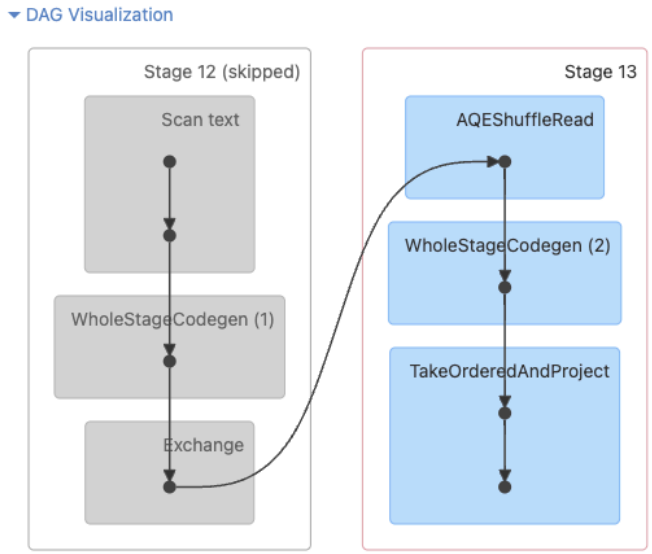


Top 15 most common words:

word	count
mr	806
elizabeth	604
said	405
darcy	380
mrs	357
much	338
must	325
miss	315
bennet	307
one	285
jane	271
bingley	261
know	244
though	233
never	230

3. Data Flow Analysis

For Basic Word Count Implementation:



Explanation of DAG

The DAG (Directed Acyclic Graph) shows how Spark breaks down the execution of the word count job into different stages. The job is broken into two main stages:

Stage 12 : Reads and processes the text file, applying transformations such as splitting into words and grouping by word. It is skipped in this case because Spark likely cached the data from a previous run, so it didn't need to re-scan the text.

Scan Text: This stage reads the text file from disk into an RDD or DataFrame.

WholeStageCodegen: This optimization combines multiple operations into a single stage to improve performance by generating optimized bytecode for the transformations.

Exchange: This represents a shuffle operation where data is redistributed across different nodes based on key values (in this case, words). Shuffling is necessary for grouping words together for counting.

Stage 13 : Performs a shuffle read to aggregate word counts and sorts the results in descending order.

AQEShuffleRead: This indicates that Adaptive Query Execution (AQE) is being used to optimize the shuffle read step. AQE dynamically adjusts query plans based on runtime statistics to improve performance.

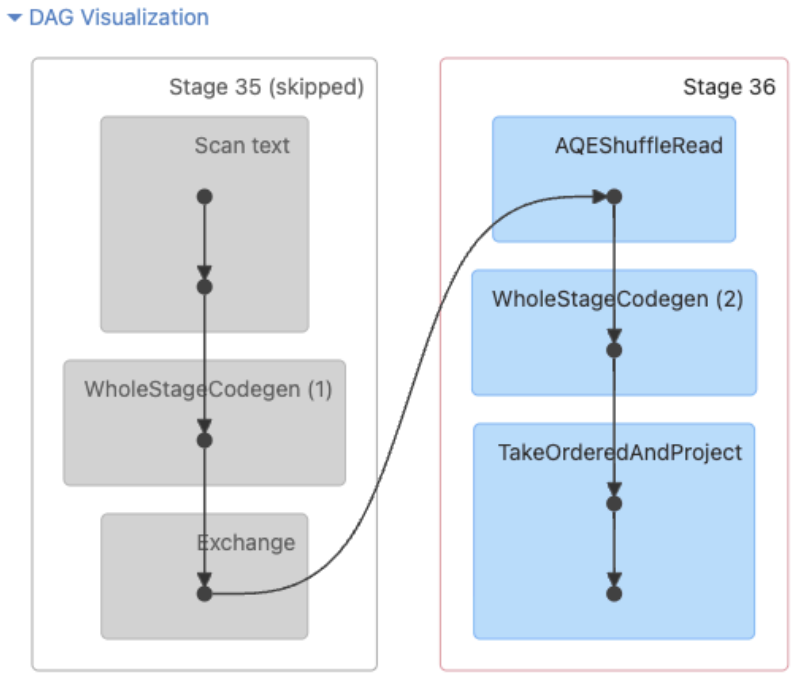
WholeStageCodegen: Another instance of WholeStageCodegen optimization for efficient execution of transformations.

TakeOrderedAndProject: This step corresponds to sorting the word counts in descending order and projecting the top results. It retrieves and displays the most frequent words.

Here's a simplified version of the **RDD transformation lineage graph based on the DAG**:

Read Text File -> Split Text into Words -> Convert to Lowercase -> Filter Empty Strings -> Group by Word-> Count Word Occurrences -> Shuffle Data for Aggregation -> Sort by Count (Descending) -> Show Results

For Extended Word Count Implementation:



Explanation of DAG

The DAG (Directed Acyclic Graph) shows how Spark breaks down the execution of the extended word count job into different stages. The job is broken into two main stages:

Stage 35: Handles initial data loading and transformation preparation but was skipped due to caching.

Scan Text: Reads the text file into an RDD or DataFrame, bringing the data into Spark's memory.

WholeStageCodegen (1): Spark uses this optimization to generate efficient bytecode, combining multiple operations into one stage.

Exchange: In this context, "Exchange" refers to a shuffle operation, which redistributes data across nodes based on the keys (in this case, words). This shuffling is essential for grouping similar data (like identical words) together to enable counting.

Stage 36: Conducts the main computation, counting and sorting words efficiently, then produces the final output

AQEShuffleRead: AQE reads the shuffled data from the previous stage (or cached data). By analyzing runtime metrics, AQE can adjust the query execution plan for optimal performance.

WholeStageCodegen (2): Another instance of bytecode generation that combines transformations for efficient execution, ensuring the transformations occur in a streamlined, single stage.

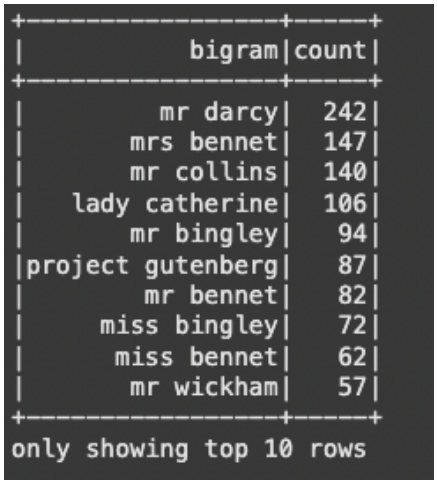
TakeOrderedAndProject: Sorts the word counts in descending order and retrieves the top results. This operation extracts the words with the highest counts for the final output.

Here's a simplified version of the **RDD transformation lineage graph based on the DAG**:

Read Text File -> Remove Punctuation -> Convert to Lowercase -> Split into Words -> Filter Empty Arrays -> Remove Stop Words -> Explode to Rows -> Filter Empty Words -> Group by Word -> Count Occurrences -> Sort by Count (Descending) -> Show Top 15 Words

Part 2: Word Co-Occurrence with PySpark

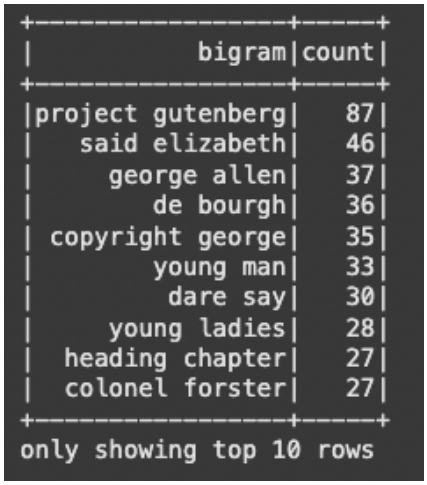
1. Word Co-Occurrence Implementation:



bigram	count
mr darcy	242
mrs bennet	147
mr collins	140
lady catherine	106
mr bingley	94
project gutenberg	87
mr bennet	82
miss bingley	72
miss bennet	62
mr wickham	57

only showing top 10 rows

As we can see, this just has names of people with their titles etc. Upon adding mr, mrs, lady etc to stop words in the program, we get :



bigram	count
project gutenberg	87
said elizabeth	46
george allen	37
de bourgh	36
copyright george	35
young man	33
dare say	30
young ladies	28
heading chapter	27
colonel forster	27

only showing top 10 rows

2. Analysis of Co-Occurrence

Analyzing bigrams is valuable in text processing for several reasons:

- **Understanding Word Relationships:** Bigrams capture the relationship between two adjacent words, which can help understand context.
- **Contextual Analysis:** In many languages, meaning is not always clear from individual words alone. The combination of two words, like “happy birthday,” conveys meaning more accurately than just “happy” or “birthday” alone.
- **Improving Text Classification Models:** By considering bigrams, you capture more granular language patterns. Many classifiers use bigrams to improve the accuracy of text classification tasks, as it can discern relationships between concepts (e.g., “data mining” or “machine learning”) that individual words might miss.

- **Collocation Detection:** Bigram analysis helps identify word collocations—pairs of words that commonly appear together. For example, “data science” is a common collocation that might not be recognized if only unigrams are considered.

Comparing the bigram analysis to the results from the word count:

The unigram analysis looks at individual words and shows that common words like **“mr”**, **“mrs”**, and **“said”** appear a lot, along with names like **“elizabeth”** and **“darcy”**. On the other hand, bigram analysis looks at pairs of words, and after removing titles, it shows meaningful word combinations like **“said elizabeth”** and **“young man”**, which tell us more about how the characters in the novel interact and the themes of the story. While unigram analysis shows us the most frequent words, bigrams help us understand how the characters speak to each other and the main ideas of the book. Removing titles makes the bigram analysis focus on the relationships between characters rather than just formal words. Together, both analyses give us a full picture of the text, with unigrams showing common words and bigrams revealing the connections and important themes.

Part 3: Data Bias and Review:

1. Bias Identification:

- **Fitness app scenario:**

Type of bias: **Accessibility bias**

This fitness app excludes users without internet access, which can disproportionately affect lower-income individuals or those in rural areas with limited connectivity. This creates a biased dataset that doesn't represent all users.

Potential solution:

Develop offline functionality that allows the app to track activity without an internet connection. The app could store data locally on the device and sync it when an internet connection becomes available. This would make the app more inclusive and provide a more representative dataset of all users, regardless of their internet access.

- **Recruitment algorithm scenario:**

Type of bias: **Educational bias**

This algorithm shows a bias towards candidates from top universities, potentially overlooking qualified candidates from other educational backgrounds. This can perpetuate existing inequalities and limit diversity in hiring.

Potential solution:

Redesign the algorithm to focus on skills, experience, and potential rather than just educational background. Include a wider range of factors in the initial screening process, such as relevant work experience, specific skills, and achievements. The algorithm could also be trained on a more diverse dataset of successful employees from various educational backgrounds. Additionally, implement regular audits of the algorithm's output to check for any unintended biases.

2. Review of Streaming Data Paper

"*Discretized Streams: Fault-Tolerant Streaming Computation at Scale*"

The paper "Discretized Streams: Fault-Tolerant Streaming Computation at Scale" introduces a new way to process big data in real-time called D-Streams. This method is really important for Spark Streaming, as it's the foundation of how it works.

The main idea of D-Streams is to break down streaming data into small batches based on time intervals. Instead of having long-running tasks that keep changing, D-Streams treat each batch as a separate, unchangeable piece of data. This is different from how other systems used to do it, and it solves some big problems.

One of the coolest things about D-Streams is how it handles failures and slow computers (stragglers). In the old systems, if a computer failed, it was hard to recover the lost data. But with D-Streams, each batch knows exactly how it was created. So if a computer fails, other computers can quickly rebuild the lost data by redoing the steps. This is called parallel recovery, and it's much faster than the old ways.

D-Streams also deal with stragglers by running the same task on multiple computers and using the result from whichever finishes first. This helps keep everything running smoothly even if some computers are being slow[1].

The paper shows that D-Streams can handle a lot of data really fast. They tested it on 100 computers and it could process over 60 million records per second, which is super impressive. It's also flexible and can do many different types of calculations on the streaming data.

For Spark Streaming, this paper is super important because it's the basis for how the whole system works. It allows Spark Streaming to process huge amounts of data in real-time, recover quickly from failures, and keep working even when some computers are slow. This makes Spark Streaming really reliable and efficient for big data processing.

In simple terms, D-Streams make streaming computations more like a series of small, manageable tasks instead of one big, ongoing process. This makes everything easier to handle, faster to fix when things go wrong, and able to work with really big amounts of data.