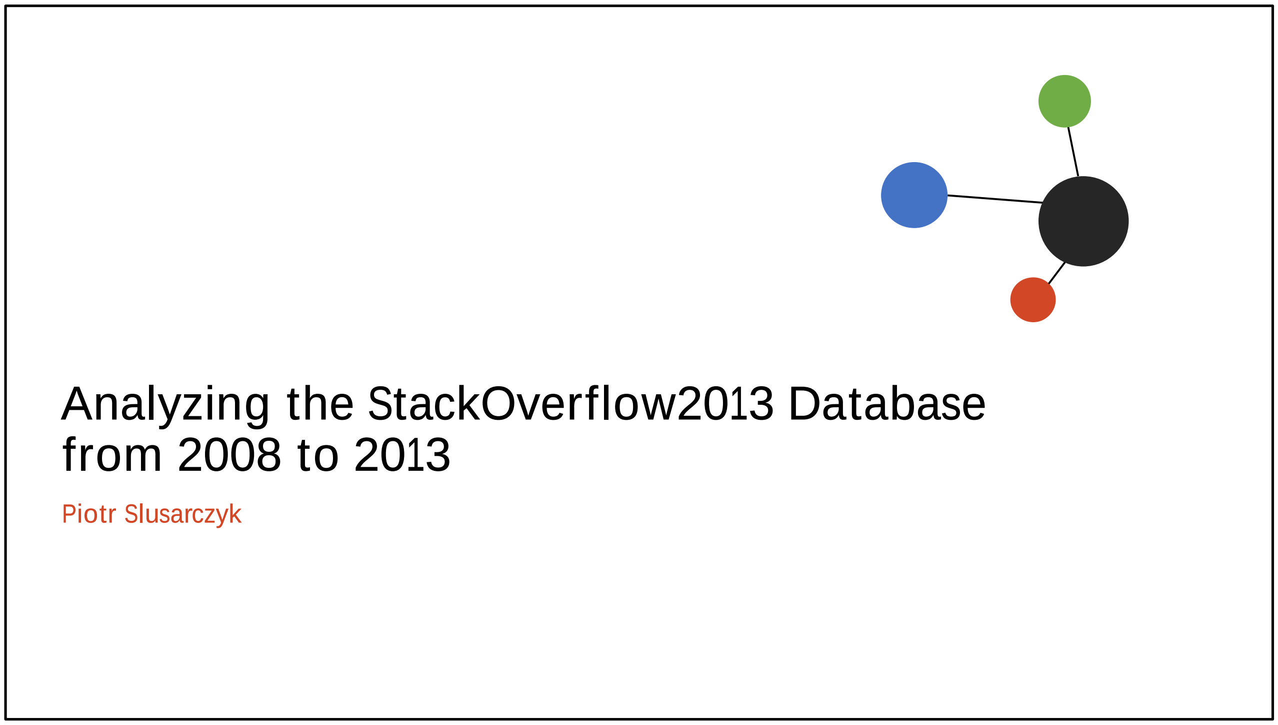
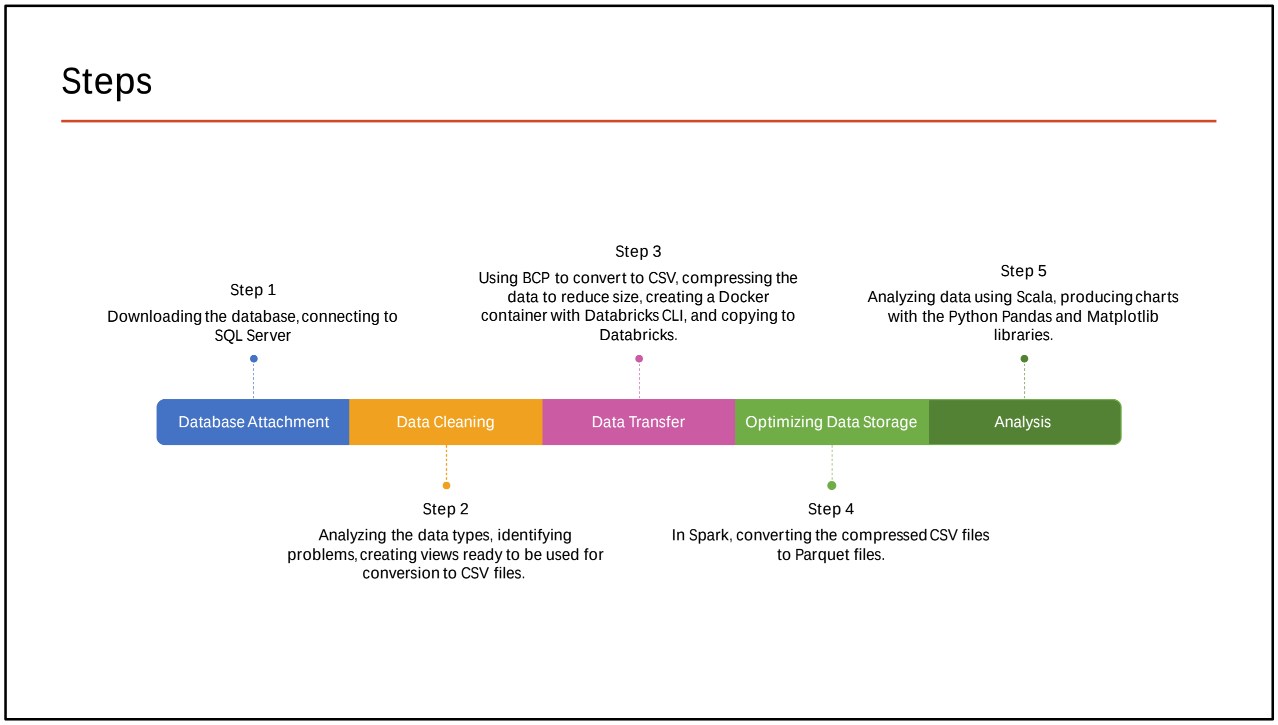


# Analyzing the StackOverflow2013 Database from 2008 to 2013

Piotr Slusarczyk

The dataset under examination is the 'StackOverflow2013' database, which spans from the years 2008 to 2013. This comprehensive dataset provides a rich opportunity for various analytical explorations and can offer valuable insights into the trends, patterns, and dynamics of the Stack Overflow community during its early years.

25’’



## Steps

Step 1
Downloading the database, connecting to SQL Server
Step 2
Analyzing the data types, identifying problems, creating views ready to be used for conversion to CSV files.
Step 3
Using BCP to convert to CSV, compressing the data to reduce size, creating a Docker container with Databricks CLI, and copying to Databricks.
Step 4
In Spark, converting the compressed CSV files to Parquet files.
Step 5
Analyzing data using Scala, producing charts with the Python Pandas and Matplotlib libraries.
Database Attachment
Data Cleaning
Data Transfer
Optimizing Data Storage
Analysis

The slide shows how the whole project looked like.

I began by downloading the database and establishing a connection to SQL Server, by attaching the downloaded database, laying the groundwork for the subsequent stages.

I then conducted data cleaning, ensuring the data types were correct and identifying any issues. This step was crucial for proper conversion to CSV.

Using BCP from command line, I converted the data into a CSV format.

I compressed the data into gzip format to make it smaller, which saves time when sending the data to Databricks.

To streamline the workflow, I set up a Docker container with Databricks CLI, simplifying the transfer of everything to Databricks.

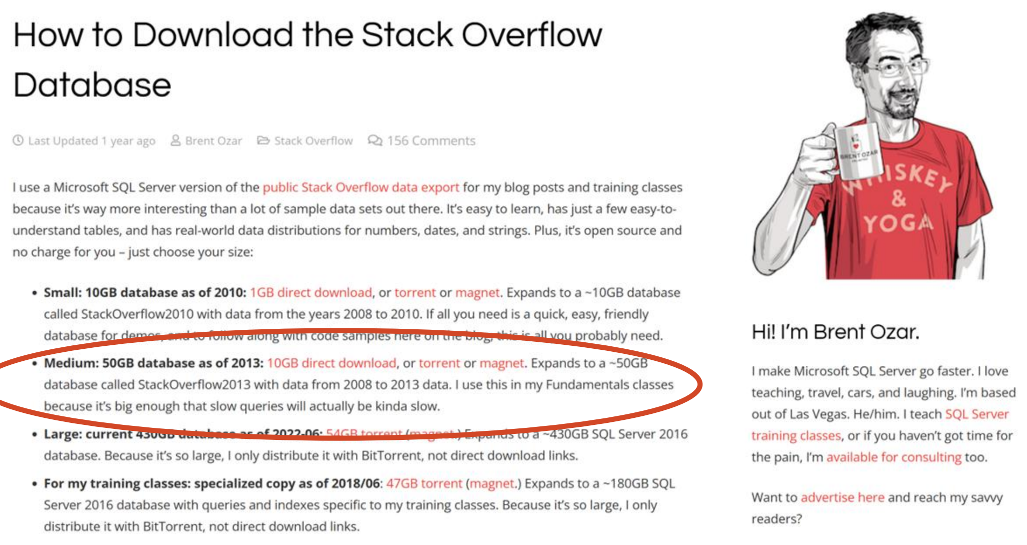
Moving forward, I used Spark to convert the compressed CSV files into the Parquet format.

Finally, using Scala, I analyzed the data, and with Python's Pandas and Matplotlib libraries, I created charts that clearly depict the insights I've gleaned.

Each step was pivotal in transforming the data to successfully complete the project.

80’’

## Data Source



How to Download the Stack Overflow Database

Last Updated 1 year ago   Brent Ozar   Stack Overflow   156 Comments

I use a Microsoft SQL Server version of the public Stack Overflow data export for my blog posts and training classes because it's way more interesting than a lot of sample data sets out there. It's easy to learn, has just a few easy-to-understand tables, and has real-world data distributions for numbers, dates, and strings. Plus, it's open source and no charge for you – just choose your size:

- **Small: 10GB database as of 2010:** 1GB direct download, or torrent or magnet. Expands to a ~10GB database called StackOverflow2010 with data from the years 2008 to 2010. If all you need is a quick, easy, friendly database for demos, and to follow along with code samples here on the blog, this is all you probably need.
- **Medium: 50GB database as of 2013:** 10GB direct download, or torrent or magnet. Expands to a ~50GB database called StackOverflow2013 with data from 2008 to 2013 data. I use this in my Fundamentals classes because it's big enough that slow queries will actually be kinda slow.
- **Large: current 430GB database as of 2022-06:** 54GB torrent (magnet.) Expands to a ~430GB SQL Server 2016 database. Because it's so large, I only distribute it with BitTorrent, not direct download links.
- **For my training classes: specialized copy as of 2018/06:** 47GB torrent (magnet.) Expands to a ~180GB SQL Server 2016 database with queries and indexes specific to my training classes. Because it's so large, I only distribute it with BitTorrent, not direct download links.

Hi! I'm Brent Ozar.

I make Microsoft SQL Server go faster. I love teaching, travel, cars, and laughing. I'm based out of Las Vegas. He/him. I teach SQL Server training classes, or if you haven't got time for the pain, I'm available for consulting too.

Want to advertise here and reach my savvy readers?

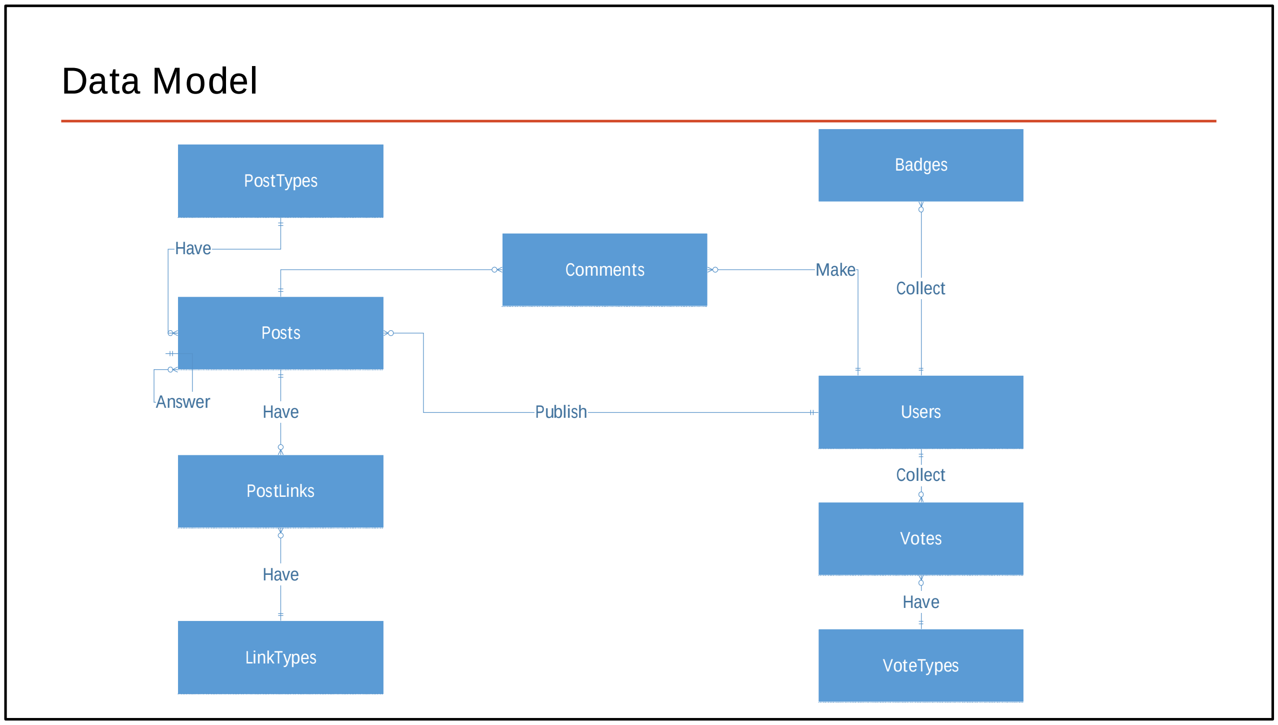
In this part of the presentation, I'm focusing on the data source that I used for the analysis. I have chosen the Stack Overflow database for a few key reasons:

Firstly, this database is derived from the public Stack Overflow data export, which is a rich dataset used widely in the tech community for analysis and training. It's not just a simple dataset; it reflects real-world distributions of numbers, dates, and strings, making it an excellent candidate for realistic data analysis scenarios.

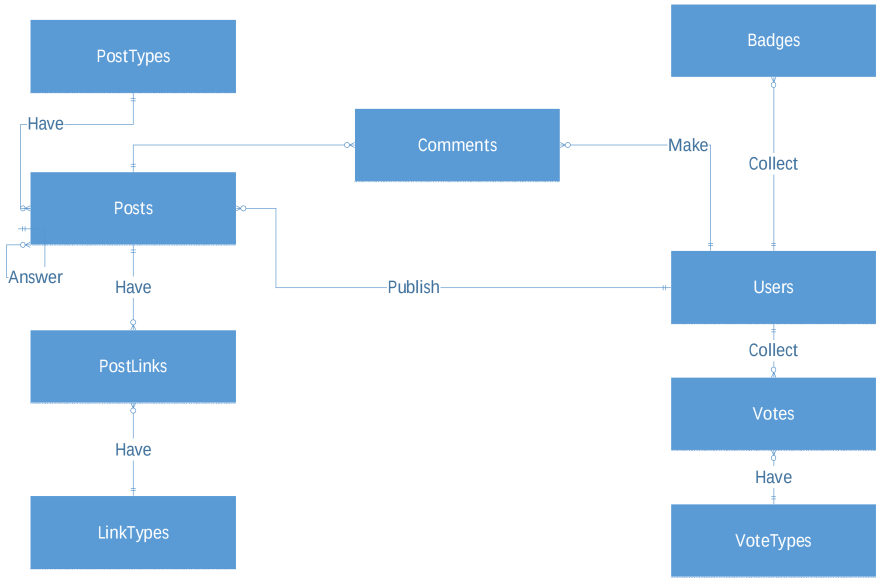
For my purposes, I've selected the medium option – the 50GB StackOverflow2013 database, which expands from a 10GB compressed file. This particular dataset contains data from 2008 to 2013.

The data comes in the form of .mdf files, which are the primary database file types used by SQL Server. This makes it convenient as they can be attached directly to SQL Server without any additional conversion, saving time and simplifying the setup process.

68’’



## Data Model



On this slide, you can see the visual representation of the StackOverflow2013 Database's data model. This model is composed of nine tables, which together capture the essence of the Stack Overflow community's interactions.

The 'Posts' table is central to this model, linked directly to 'PostTypes', which is composed of two main types of entries: Questions and Answers.

Users are the lifeblood of Stack Overflow, and here you can see their interactions with the system. A user can own a post, meaning they've contributed content to the community, or they can answer posts, providing solutions to questions posed by others.

The 'Comments' table allows for discourse on these posts, enabling users to seek clarification, discuss content, and offer brief insights.

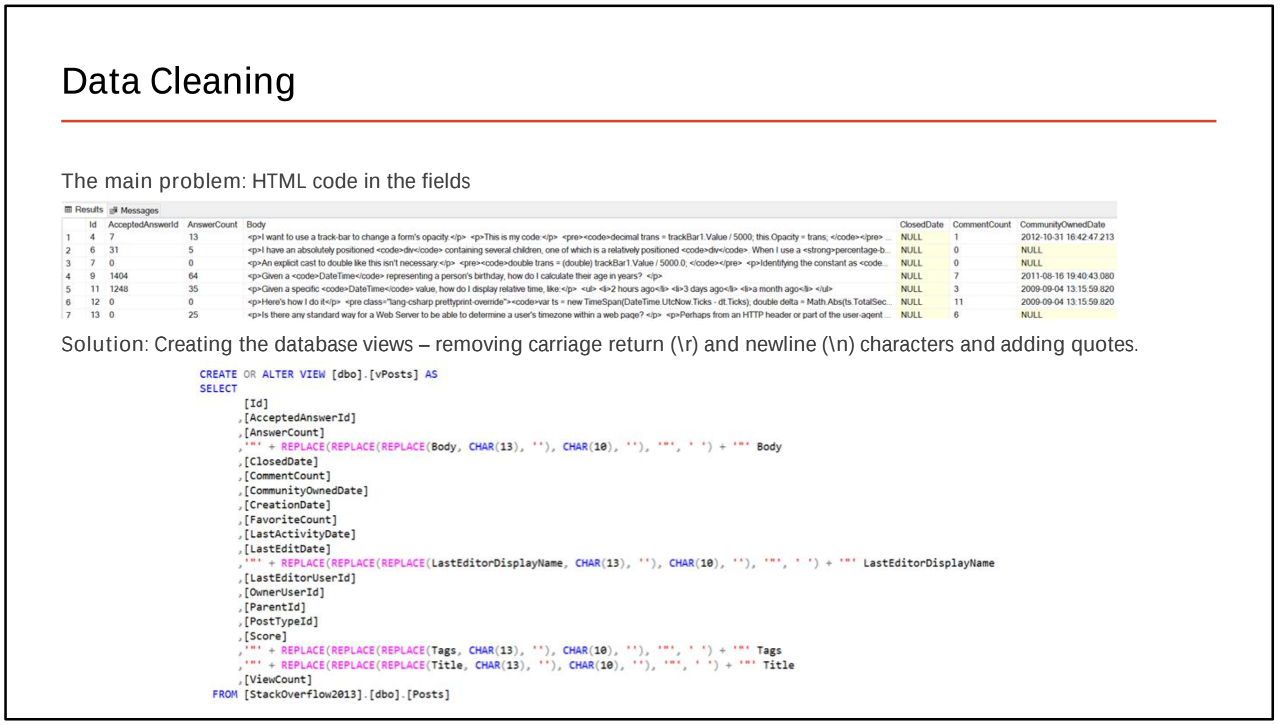
In the 'Votes' table, the community's feedback on posts is captured, where users can vote to signify the usefulness and accuracy of the content provided. It's connected to 'VoteTypes', which defines the nature of these votes.

The 'Users' table holds profiles, reflecting the personal and professional backgrounds of community members, as well as their contributions and reputation within the platform. This table also connects to 'Badges', which are earned by users as a form of recognition for their contributions.

User engagement is further documented through 'PostLinks' and 'LinkTypes', which trace the relationships between different posts, such as duplicates or related content.

Together, these tables not only store data but also tell the story of how knowledge is built, shared, and valued in one of the largest technical communities online.

98’’



## Data Cleaning

The main problem: HTML code in the fields

Results | Messages

| | Id | AcceptedAnswerId | AnswerCount | Body | ClosedDate | CommentCount | CommunityOwnedDate |
|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | 13 | <p>I want to use a track-bar to change a form's opacity.</p> <p>This is my code:</p> <pre><code>decimal trans = trackBar1.Value / 5000; this.Opacity = trans; </code></pre> ... | NULL | 1 | 2012-10-31 16:42:47.213 |
| 2 | 6 | 31 | 5 | <p>I have an absolutely positioned <code>div</code> containing several children, one of which is a relatively positioned <code>div</code>. When I use a <strong>percentage-b... | NULL | 0 | NULL |
| 3 | 7 | 0 | 0 | <p>An explicit cast to double like this isn't necessary:</p> <pre><code>double trans = (double) trackBar1.Value / 5000.0; </code></pre> <p>Identifying the constant as <code... | NULL | 0 | NULL |
| 4 | 9 | 1404 | 64 | <p>Given a <code>DateTime</code> representing a person's birthday, how do I calculate their age in years? </p> | NULL | 7 | 2011-08-16 19:40:43.080 |
| 5 | 11 | 1248 | 35 | <p>Given a specific <code>DateTime</code> value, how do I display relative time, like:</p> <ul> <li>2 hours ago</li> <li>3 days ago</li> <li>a month ago</li> </ul> | NULL | 3 | 2009-09-04 13:15:59.820 |
| 6 | 12 | 0 | 0 | <p>Here's how I do it</p> <pre class="lang-csharp prettyprint-override"><code>var ts = new TimeSpan(DateTime.UtcNow.Ticks - dt.Ticks); double delta = Math.Abs(ts.TotalSec... | NULL | 11 | 2009-09-04 13:15:59.820 |
| 7 | 13 | 0 | 25 | <p>Is there any standard way for a Web Server to be able to determine a user's timezone within a web page? </p> <p>Perhaps from an HTTP header or part of the user-agent ... | NULL | 6 | NULL |

Solution: Creating the database views – removing carriage return (\r) and newline (\n) characters and adding quotes.

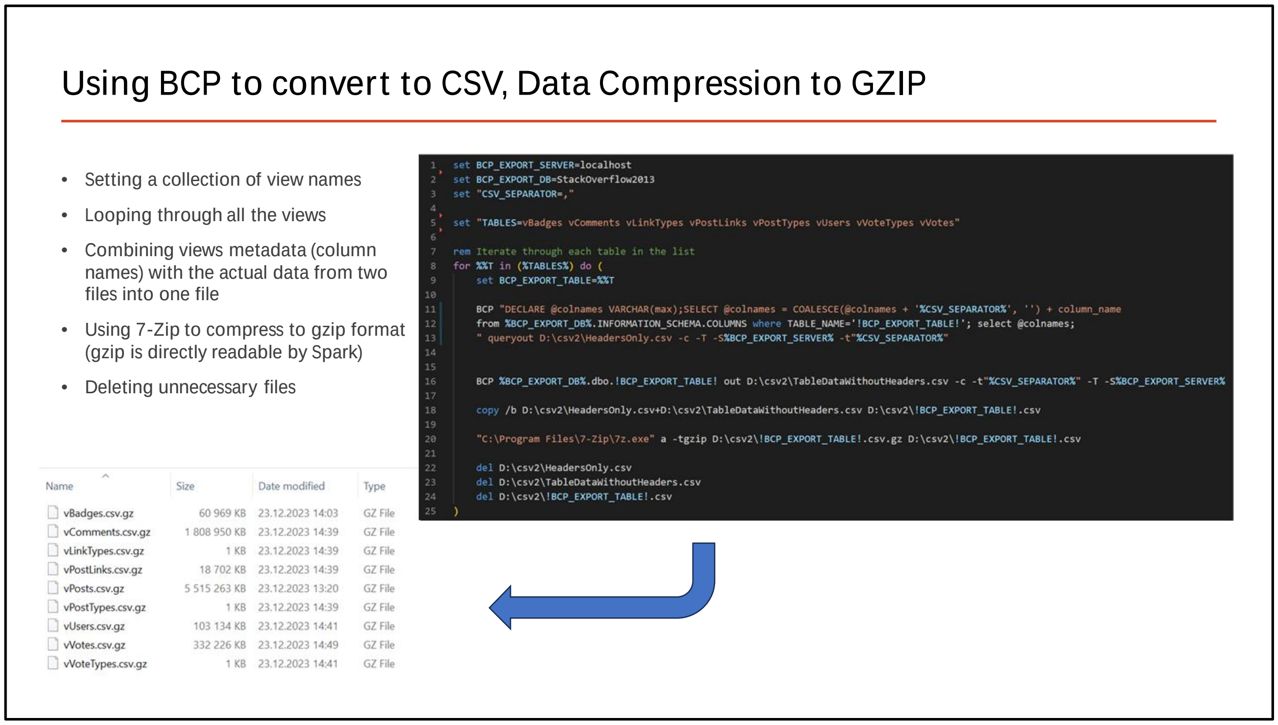
```sql
CREATE OR ALTER VIEW [dbo].[vPosts] AS
SELECT
       [Id]
      ,[AcceptedAnswerId]
      ,[AnswerCount]
      ,'"' + REPLACE(REPLACE(REPLACE(Body, CHAR(13), ''), CHAR(10), ''), '"', ' ') + '"' Body
      ,[ClosedDate]
      ,[CommentCount]
      ,[CommunityOwnedDate]
      ,[CreationDate]
      ,[FavoriteCount]
      ,[LastActivityDate]
      ,[LastEditDate]
      ,'"' + REPLACE(REPLACE(REPLACE(LastEditorDisplayName, CHAR(13), ''), CHAR(10), ''), '"', ' ') + '"' LastEditorDisplayName
      ,[LastEditorUserId]
      ,[OwnerUserId]
      ,[ParentId]
      ,[PostTypeId]
      ,[Score]
      ,'"' + REPLACE(REPLACE(REPLACE(Tags, CHAR(13), ''), CHAR(10), ''), '"', ' ') + '"' Tags
      ,'"' + REPLACE(REPLACE(REPLACE(Title, CHAR(13), ''), CHAR(10), ''), '"', ' ') + '"' Title
      ,[ViewCount]
  FROM [StackOverflow2013].[dbo].[Posts]
```

On this slide, I present the crucial step in data preparation: Data Cleaning. The main challenge I faced was the presence of HTML code within the text fields. This code caused problems in export procedures.

To address this, I implemented a data cleaning process through the creation of database views. These views serve a specific purpose: they remove unnecessary carriage return (\r) and newline (\n) characters from text fields. Additionally, to align with the objective of converting the data into CSV format, I needed to ensure that each text field is properly quoted.

The SQL script you see performs this transformation seamlessly. By wrapping the text fields with quotes and stripping out unwanted characters, I make the data CSV-ready. This step was not only necessary but also the most efficient route to prepare the text fields for the conversion process.

60’’



## Using BCP to convert to CSV, Data Compression to GZIP

- Setting a collection of view names
- Looping through all the views
- Combining views metadata (column names) with the actual data from two files into one file
- Using 7-Zip to compress to gzip format (gzip is directly readable by Spark)
- Deleting unnecessary files

```
set BCP_EXPORT_SERVER=localhost
set BCP_EXPORT_DB=StackOverflow2013
set "CSV_SEPARATOR=,"

set "TABLES=vBadges vComments vLinkTypes vPostLinks vPostTypes vUsers vVoteTypes vVotes"

rem Iterate through each table in the list
for %%T in (%TABLES%) do (
    set BCP_EXPORT_TABLE=%%T

    BCP "DECLARE @colnames VARCHAR(max);SELECT @colnames = COALESCE(@colnames + '%CSV_SEPARATOR%', '') + column_name
    from %BCP_EXPORT_DB%.INFORMATION_SCHEMA.COLUMNS where TABLE_NAME='!BCP_EXPORT_TABLE!'; select @colnames;
    " queryout D:\csv2\HeadersOnly.csv -c -T -S%BCP_EXPORT_SERVER% -t"%CSV_SEPARATOR%"


    BCP %BCP_EXPORT_DB%.dbo.!BCP_EXPORT_TABLE! out D:\csv2\TableDataWithoutHeaders.csv -c -t"%CSV_SEPARATOR%" -T -S%BCP_EXPORT_SERVER%

    copy /b D:\csv2\HeadersOnly.csv+D:\csv2\TableDataWithoutHeaders.csv D:\csv2\!BCP_EXPORT_TABLE!.csv

    "C:\Program Files\7-Zip\7z.exe" a -tgzip D:\csv2\!BCP_EXPORT_TABLE!.csv.gz D:\csv2\!BCP_EXPORT_TABLE!.csv

    del D:\csv2\HeadersOnly.csv
    del D:\csv2\TableDataWithoutHeaders.csv
    del D:\csv2\!BCP_EXPORT_TABLE!.csv
)
```

| Name | Size | Date modified | Type |
|---|---|---|---|
| vBadges.csv.gz | 60 969 KB | 23.12.2023 14:03 | GZ File |
| vComments.csv.gz | 1 808 950 KB | 23.12.2023 14:39 | GZ File |
| vLinkTypes.csv.gz | 1 KB | 23.12.2023 14:39 | GZ File |
| vPostLinks.csv.gz | 18 702 KB | 23.12.2023 14:39 | GZ File |
| vPosts.csv.gz | 5 515 263 KB | 23.12.2023 13:20 | GZ File |
| vPostTypes.csv.gz | 1 KB | 23.12.2023 14:39 | GZ File |
| vUsers.csv.gz | 103 134 KB | 23.12.2023 14:41 | GZ File |
| vVotes.csv.gz | 332 226 KB | 23.12.2023 14:49 | GZ File |
| vVoteTypes.csv.gz | 1 KB | 23.12.2023 14:41 | GZ File |

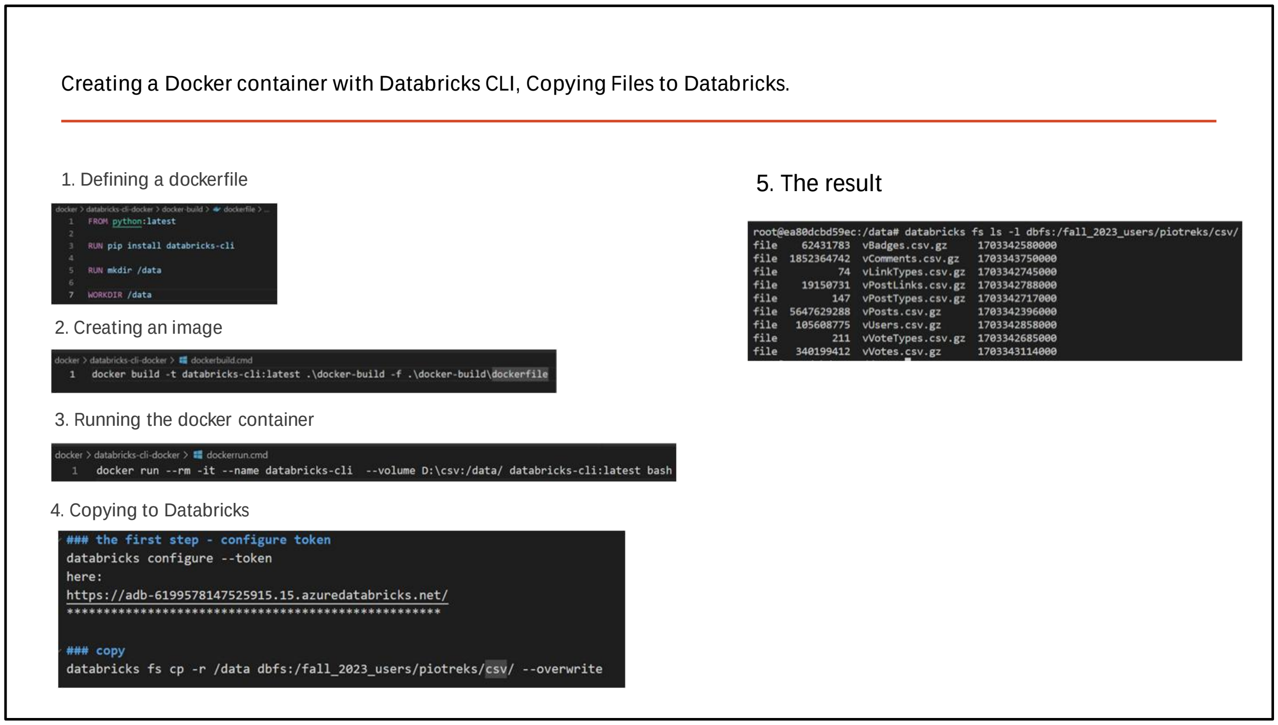
In this slide, I'm detailing the efficient process of exporting data from SQL Server to CSV and then compressing it for optimal transfer speed.

I start by using BCP, which stands for Bulk Copy Program, a tool provided by SQL Server. It's the fastest method available to perform large-scale data exports to CSV format, ensuring speed and efficiency. I specify the view names that form the collection of data I wish to export, and then I initiate a loop to process each view individually.

Once the CSV files are generated, I use 7-Zip to compress them into gzip format. One of the key advantages of gzip is its compatibility with Databricks' file system. It can read gzip files directly, which means there's no need to decompress them on the platform, saving valuable processing time and simplifying the data pipeline.

The bottom part of the slide shows the actual script that orchestrates this entire process: from setting environment variables for BCP to executing the compression commands and cleaning up interim files. This script is the backbone of the procedure, handling everything from data extraction to final file compression, ready for sending to Databricks.

67’’



Creating a Docker container with Databricks CLI, Copying Files to Databricks.

1. Defining a dockerfile

```
docker > databricks-cli-docker > docker-build > dockerfile > ...
FROM python:latest

RUN pip install databricks-cli

RUN mkdir /data

WORKDIR /data
```

2. Creating an image

```
docker > databricks-cli-docker > dockerbuild.cmd
docker build -t databricks-cli:latest .\docker-build -f .\docker-build\dockerfile
```

3. Running the docker container

```
docker > databricks-cli-docker > dockerrun.cmd
docker run --rm -it --name databricks-cli  --volume D:\csv:/data/ databricks-cli:latest bash
```

4. Copying to Databricks

```
### the first step - configure token
databricks configure --token
here:
https://adb-6199578147525915.15.azuredatabricks.net/
***************************************************


### copy
databricks fs cp -r /data dbfs:/fall_2023_users/piotreks/csv/ --overwrite
```

5. The result

```
root@ea80dcbd59ec:/data# databricks fs ls -l dbfs:/fall_2023_users/piotreks/csv/
file    62431783  vBadges.csv.gz      1703342580000
file  1852364742  vComments.csv.gz    1703343750000
file          74  vLinkTypes.csv.gz   1703342745000
file    19150731  vPostLinks.csv.gz   1703342788000
file         147  vPostTypes.csv.gz   1703342717000
file  5647629288  vPosts.csv.gz       1703342396000
file   105608775  vUsers.csv.gz       1703342858000
file         211  vVoteTypes.csv.gz   1703342685000
file   340199412  vVotes.csv.gz       1703343114000
```

In this section of the presentation, I'm going to walk you through the process of creating a Docker container equipped with Databricks CLI for the purpose of transferring files efficiently to Databricks.

I began by defining a Dockerfile. This file is a blueprint for Docker, describing the environment needed to run Databricks CLI. It's based on a Python image, installs the Databricks CLI, and sets a working directory.

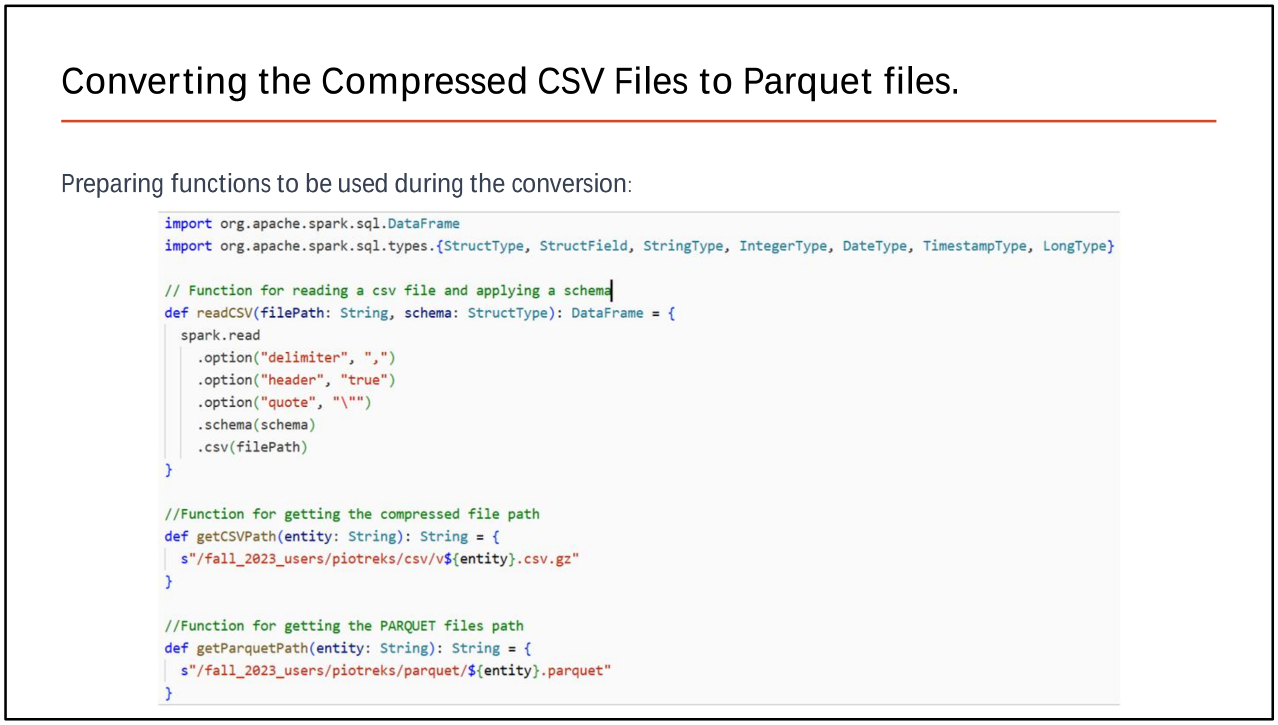
Next, I created an image from the Dockerfile. The command shown on the slide builds the Docker image, which contains all the necessary components to run the Databricks CLI.

Once the image was ready, I proceeded to run the Docker container. This step involves executing a Docker run command, which initiates a container instance where I can execute Databricks CLI commands.

The fourth step was to copy the CSV files to Databricks. To do this securely, I first configured access to Databricks using a token. This ensures that the connection to Databricks is secure and that only authorized commands are executed. Once the token was configured, I used the Databricks CLI to copy files from the local data directory to the Databricks file system, replacing any existing files with the same name.

The result, which you can see on the right, is the list of files successfully copied to Databricks. These are the gzipped CSV files now stored in the Databricks file system, ready to be used for further conversion and data analysis.

84’’



## Converting the Compressed CSV Files to Parquet files.

Preparing functions to be used during the conversion:

```scala
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, DateType, TimestampType, LongType}

// Function for reading a csv file and applying a schema
def readCSV(filePath: String, schema: StructType): DataFrame = {
  spark.read
    .option("delimiter", ",")
    .option("header", "true")
    .option("quote", "\"")
    .schema(schema)
    .csv(filePath)
}

//Function for getting the compressed file path
def getCSVPath(entity: String): String = {
  s"/fall_2023_users/piotreks/csv/v${entity}.csv.gz"
}

//Function for getting the PARQUET files path
def getParquetPath(entity: String): String = {
  s"/fall_2023_users/piotreks/parquet/${entity}.parquet"
}
```

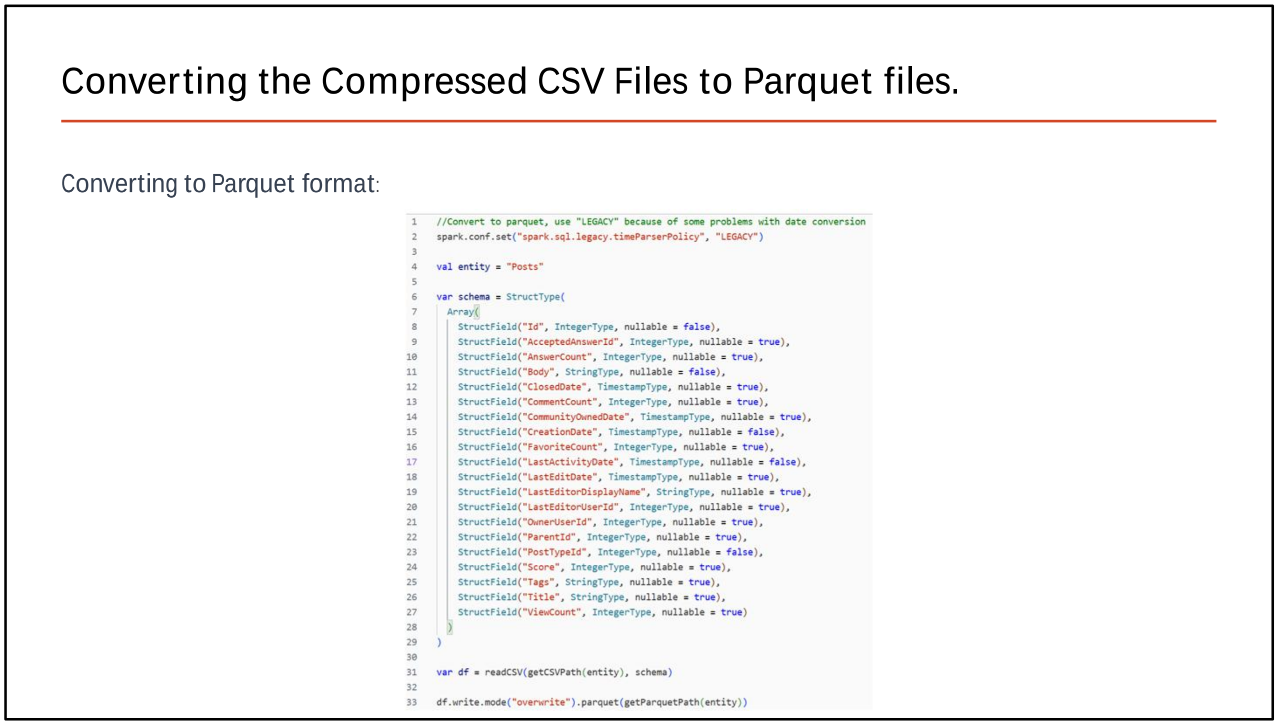
This slide illustrates the functions I've prepared for converting compressed CSV files to the Parquet format, optimized for processing with Apache Spark.

The first function, readCSV, is a Scala function that reads a CSV file into a Spark DataFrame, enforcing a predefined schema. It's essential for ensuring the data conforms to the expected format, which facilitates reliable analysis and processing.

To manage file paths easier, I use the getCSVPath and getParquetPath functions. They generate the storage paths for the CSV and Parquet files, respectively.

By applying these functions, I ensure that the data transformation pipeline is not only automated but also integrated, setting the stage for the subsequent analytics tasks in Spark without any manual intervention.

79’’



## Converting the Compressed CSV Files to Parquet files.

Converting to Parquet format:

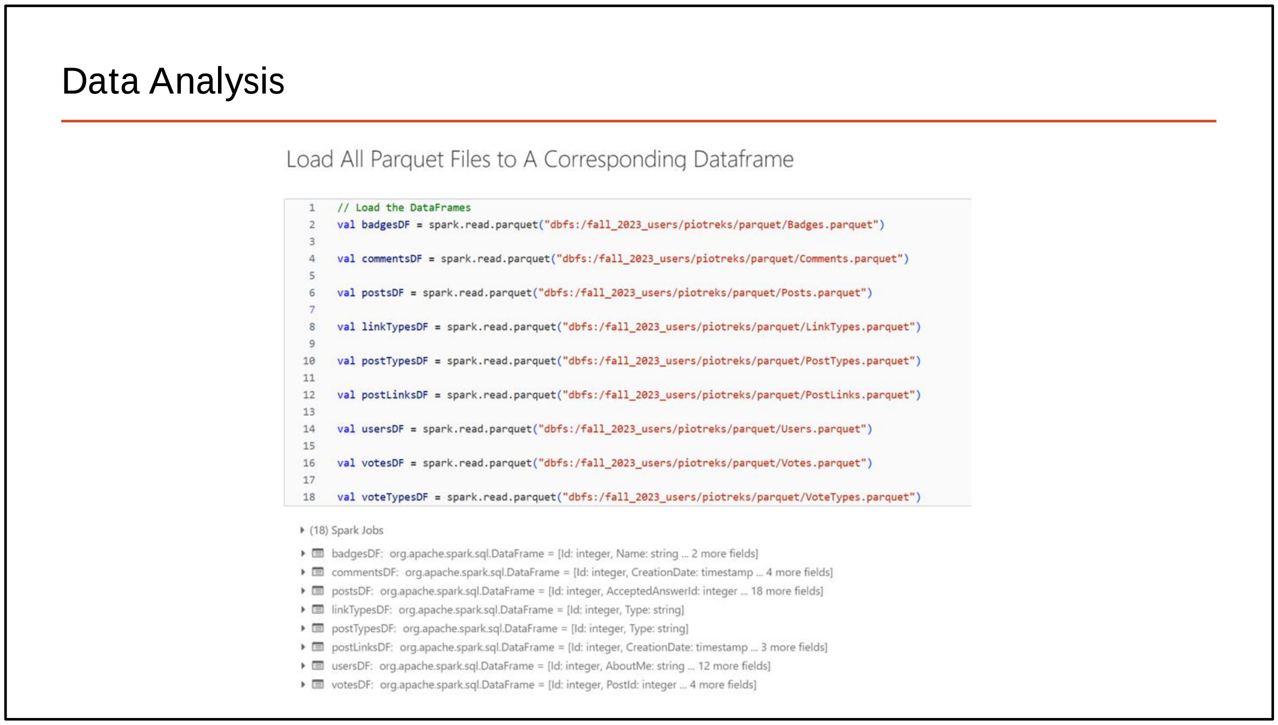
```scala
//Convert to parquet, use "LEGACY" because of some problems with date conversion
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEGACY")

val entity = "Posts"

var schema = StructType(
  Array(
    StructField("Id", IntegerType, nullable = false),
    StructField("AcceptedAnswerId", IntegerType, nullable = true),
    StructField("AnswerCount", IntegerType, nullable = true),
    StructField("Body", StringType, nullable = false),
    StructField("ClosedDate", TimestampType, nullable = true),
    StructField("CommentCount", IntegerType, nullable = true),
    StructField("CommunityOwnedDate", TimestampType, nullable = true),
    StructField("CreationDate", TimestampType, nullable = false),
    StructField("FavoriteCount", IntegerType, nullable = true),
    StructField("LastActivityDate", TimestampType, nullable = false),
    StructField("LastEditDate", TimestampType, nullable = true),
    StructField("LastEditorDisplayName", StringType, nullable = true),
    StructField("LastEditorUserId", IntegerType, nullable = true),
    StructField("OwnerUserId", IntegerType, nullable = true),
    StructField("ParentId", IntegerType, nullable = true),
    StructField("PostTypeId", IntegerType, nullable = false),
    StructField("Score", IntegerType, nullable = true),
    StructField("Tags", StringType, nullable = true),
    StructField("Title", StringType, nullable = true),
    StructField("ViewCount", IntegerType, nullable = true)
  )
)

var df = readCSV(getCSVPath(entity), schema)

df.write.mode("overwrite").parquet(getParquetPath(entity))
```

On this slide, I show the process I used to convert a compressed CSV file into the Parquet format using Spark. The schema defined here mirrors the exact structure in the data model shown earlier.

Here, you can see the code snippet for converting the 'Posts' file as an example. I've set Spark's SQL legacy time parser policy to avoid issues with date conversion, which is a common problem in data processing.

The readCSV function is used to read the CSV file into a DataFrame, applying the schema directly as it's defined. Then, with the write method, the DataFrame is saved in the Parquet format. While this slide shows the conversion for one file, I want to note that the same approach was used for the rest of the files. Each CSV file was converted in a similar manner, ensuring a uniform and efficient batch processing.

46''



## Data Analysis

Load All Parquet Files to A Corresponding Dataframe

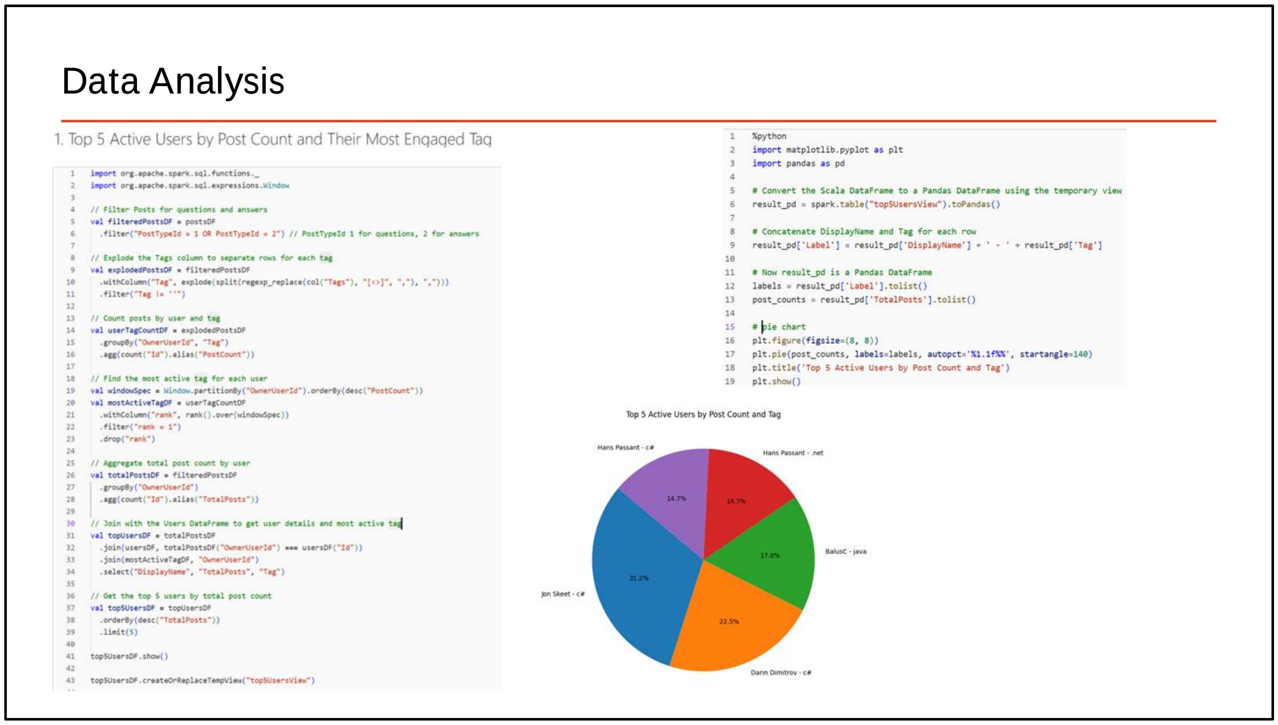
```scala
// Load the DataFrames
val badgesDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/Badges.parquet")

val commentsDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/Comments.parquet")

val postsDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/Posts.parquet")

val linkTypesDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/LinkTypes.parquet")

val postTypesDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/PostTypes.parquet")

val postLinksDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/PostLinks.parquet")

val usersDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/Users.parquet")

val votesDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/Votes.parquet")

val voteTypesDF = spark.read.parquet("dbfs:/fall_2023_users/piotreks/parquet/VoteTypes.parquet")
```

```
(18) Spark Jobs
badgesDF:  org.apache.spark.sql.DataFrame = [Id: integer, Name: string ... 2 more fields]
commentsDF:  org.apache.spark.sql.DataFrame = [Id: integer, CreationDate: timestamp ... 4 more fields]
postsDF:  org.apache.spark.sql.DataFrame = [Id: integer, AcceptedAnswerId: integer ... 18 more fields]
linkTypesDF:  org.apache.spark.sql.DataFrame = [Id: integer, Type: string]
postTypesDF:  org.apache.spark.sql.DataFrame = [Id: integer, Type: string]
postLinksDF:  org.apache.spark.sql.DataFrame = [Id: integer, CreationDate: timestamp ... 3 more fields]
usersDF:  org.apache.spark.sql.DataFrame = [Id: integer, AboutMe: string ... 12 more fields]
votesDF:  org.apache.spark.sql.DataFrame = [Id: integer, PostId: integer ... 4 more fields]
```

This slide shows the first step of my data analysis, where I have loaded all Parquet files into Spark DataFrames. Each DataFrame corresponds to a specific component of the Stack Overflow data captured in Parquet files, which are highly efficient for this kind of processing.

By loading these datasets into DataFrames, I've laid the foundation for further data exploration.

18''

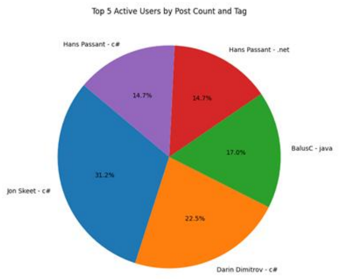


## Data Analysis

1. Top 5 Active Users by Post Count and Their Most Engaged Tag

```scala
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window

// Filter Posts for questions and answers
val filteredPostsDF = postsDF
  .filter("PostTypeId = 1 OR PostTypeId = 2") // PostTypeId 1 for questions, 2 for answers

// Explode the Tags column to separate rows for each tag
val explodedPostsDF = filteredPostsDF
  .withColumn("Tag", explode(split(regexp_replace(col("Tags"), "[<>]", ","), ",")))
  .filter("Tag != ''")

// Count posts by user and tag
val userTagCountDF = explodedPostsDF
  .groupBy("OwnerUserId", "Tag")
  .agg(count("Id").alias("PostCount"))

// Find the most active tag for each user
val windowSpec = Window.partitionBy("OwnerUserId").orderBy(desc("PostCount"))
val mostActiveTagDF = userTagCountDF
  .withColumn("rank", rank().over(windowSpec))
  .filter("rank = 1")
  .drop("rank")

// Aggregate total post count by user
val totalPostsDF = filteredPostsDF
  .groupBy("OwnerUserId")
  .agg(count("Id").alias("TotalPosts"))

// Join with the Users DataFrame to get user details and most active tag
val topUsersDF = totalPostsDF
  .join(usersDF, totalPostsDF("OwnerUserId") === usersDF("Id"))
  .join(mostActiveTagDF, "OwnerUserId")
  .select("DisplayName", "TotalPosts", "Tag")

// Get the top 5 users by total post count
val top5UsersDF = topUsersDF
  .orderBy(desc("TotalPosts"))
  .limit(5)

top5UsersDF.show()

top5UsersDF.createOrReplaceTempView("top5UsersView")
```

```python
%python
import matplotlib.pyplot as plt
import pandas as pd

# Convert the Scala DataFrame to a Pandas DataFrame using the temporary view
result_pd = spark.table("top5UsersView").toPandas()

# Concatenate DisplayName and Tag for each row
result_pd['Label'] = result_pd['DisplayName'] + ' - ' + result_pd['Tag']

# Now result_pd is a Pandas DataFrame
labels = result_pd['Label'].tolist()
post_counts = result_pd['TotalPosts'].tolist()

# pie chart
plt.figure(figsize=(8, 8))
plt.pie(post_counts, labels=labels, autopct='%1.1f%%', startangle=140)
plt.title('Top 5 Active Users by Post Count and Tag')
plt.show()
```

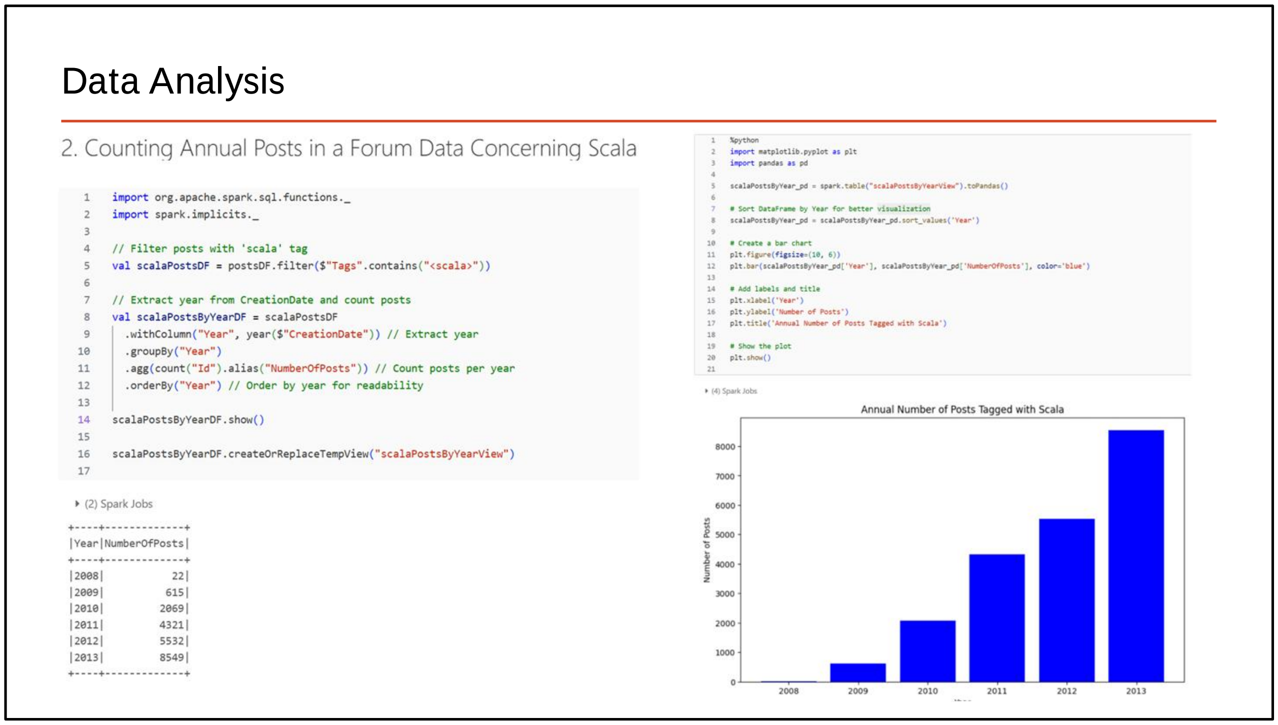


This slide presents an analysis aimed at identifying the top 5 active users on Stack Overflow by post count, along with their most engaged tags. I've used Spark to filter and process posts, and then to count and rank user activity. By exploding the tags column, I ensured each tag associated with a post was considered individually, which allowed me to determine the most common tags per user.

After aggregating the total post counts and the most active tags for each user, I joined this data with the users' details to provide a more comprehensive view. The result of this join operation gave me a table showcasing users' display names, their total post counts, and their primary tags.

Finally, I visualized this data using Python's matplotlib and pandas libraries to create the pie chart you see here. This chart  highlights the proportion of contributions from each of the top users.

53’’



## Data Analysis

2. Counting Annual Posts in a Forum Data Concerning Scala

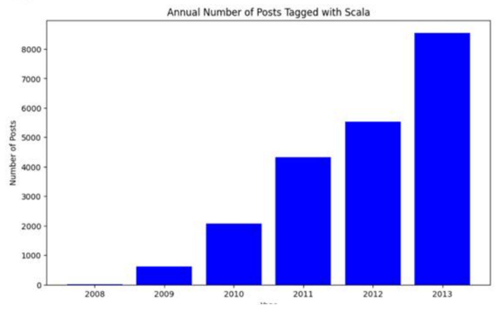
```scala
import org.apache.spark.sql.functions._
import spark.implicits._

// Filter posts with 'scala' tag
val scalaPostsDF = postsDF.filter($"Tags".contains("<scala>"))

// Extract year from CreationDate and count posts
val scalaPostsByYearDF = scalaPostsDF
  .withColumn("Year", year($"CreationDate")) // Extract year
  .groupBy("Year")
  .agg(count("Id").alias("NumberOfPosts")) // Count posts per year
  .orderBy("Year") // Order by year for readability

scalaPostsByYearDF.show()

scalaPostsByYearDF.createOrReplaceTempView("scalaPostsByYearView")
```

▸ (2) Spark Jobs

```
+----+-------------+
|Year|NumberOfPosts|
+----+-------------+
|2008|           22|
|2009|          615|
|2010|         2069|
|2011|         4321|
|2012|         5532|
|2013|         8549|
+----+-------------+
```

```python
%python
import matplotlib.pyplot as plt
import pandas as pd

scalaPostsByYear_pd = spark.table("scalaPostsByYearView").toPandas()

# Sort DataFrame by Year for better visualization
scalaPostsByYear_pd = scalaPostsByYear_pd.sort_values('Year')

# Create a bar chart
plt.figure(figsize=(10, 6))
plt.bar(scalaPostsByYear_pd['Year'], scalaPostsByYear_pd['NumberOfPosts'], color='blue')

# Add labels and title
plt.xlabel('Year')
plt.ylabel('Number of Posts')
plt.title('Annual Number of Posts Tagged with Scala')

# Show the plot
plt.show()
```

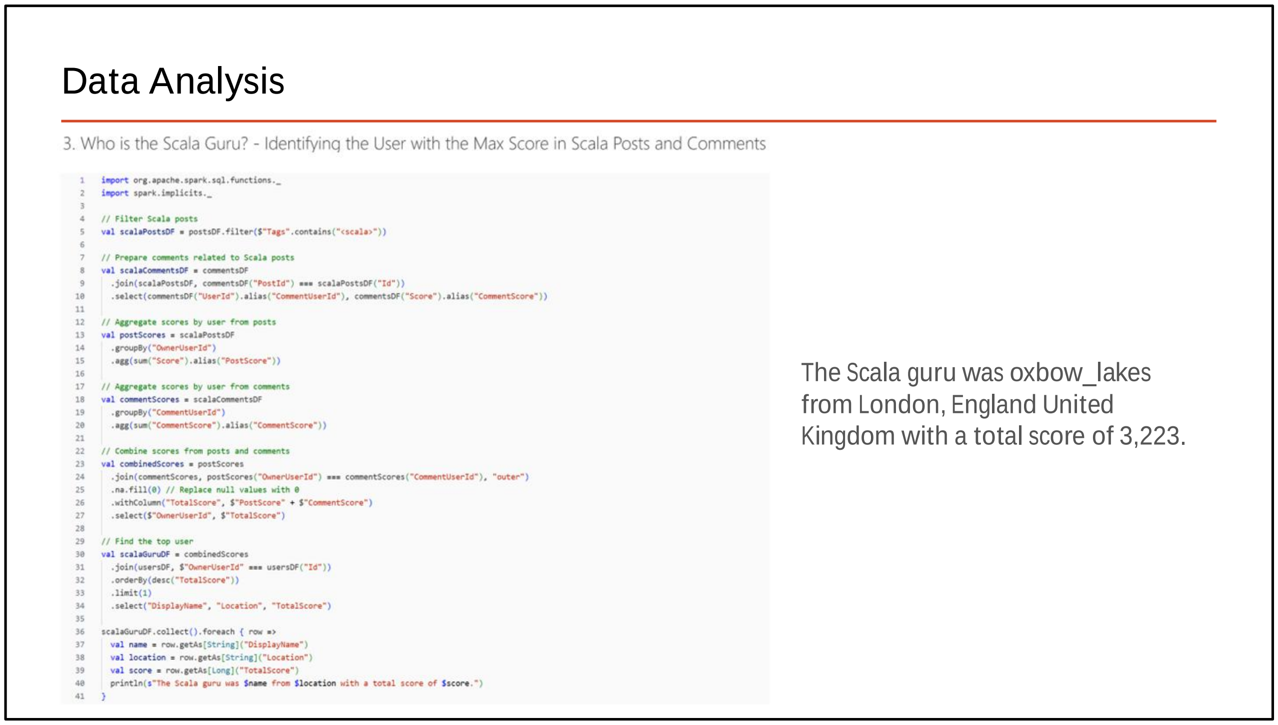
▸ (4) Spark Jobs



This slide demonstrates an approach to counting annual posts in a forum dataset, focusing on content related to the Scala programming language. Initially, I used Spark's DataFrame API to filter the dataset for posts tagged with 'Scala'. Next, I extracted the year from each post's creation date and aggregated the count of posts per year. This process involved sorting and grouping the data to facilitate a clearer analysis.

To visualize the trends, I employed Python's matplotlib and pandas libraries again, creating a bar chart.

25’’



## Data Analysis

3. Who is the Scala Guru? - Identifying the User with the Max Score in Scala Posts and Comments

```scala
import org.apache.spark.sql.functions._
import spark.implicits._

// Filter Scala posts
val scalaPostsDF = postsDF.filter($"Tags".contains("<scala>"))

// Prepare comments related to Scala posts
val scalaCommentsDF = commentsDF
  .join(scalaPostsDF, commentsDF("PostId") === scalaPostsDF("Id"))
  .select(commentsDF("UserId").alias("CommentUserId"), commentsDF("Score").alias("CommentScore"))

// Aggregate scores by user from posts
val postScores = scalaPostsDF
  .groupBy("OwnerUserId")
  .agg(sum("Score").alias("PostScore"))

// Aggregate scores by user from comments
val commentScores = scalaCommentsDF
  .groupBy("CommentUserId")
  .agg(sum("CommentScore").alias("CommentScore"))

// Combine scores from posts and comments
val combinedScores = postScores
  .join(commentScores, postScores("OwnerUserId") === commentScores("CommentUserId"), "outer")
  .na.fill(0) // Replace null values with 0
  .withColumn("TotalScore", $"PostScore" + $"CommentScore")
  .select($"OwnerUserId", $"TotalScore")

// Find the top user
val scalaGuruDF = combinedScores
  .join(usersDF, $"OwnerUserId" === usersDF("Id"))
  .orderBy(desc("TotalScore"))
  .limit(1)
  .select("DisplayName", "Location", "TotalScore")

scalaGuruDF.collect().foreach { row =>
  val name = row.getAs[String]("DisplayName")
  val location = row.getAs[String]("Location")
  val score = row.getAs[Long]("TotalScore")
  println(s"The Scala guru was $name from $location with a total score of $score.")
}
```

The Scala guru was oxbow_lakes from London, England United Kingdom with a total score of 3,223.

This slide focuses on finding the top participant regarding Scala topics. I began by filtering the posts for those tagged with 'Scala' and then prepared a dataset of comments related to these posts. By joining the two datasets and selecting relevant fields, I aggregated scores from both posts and comments for each user.

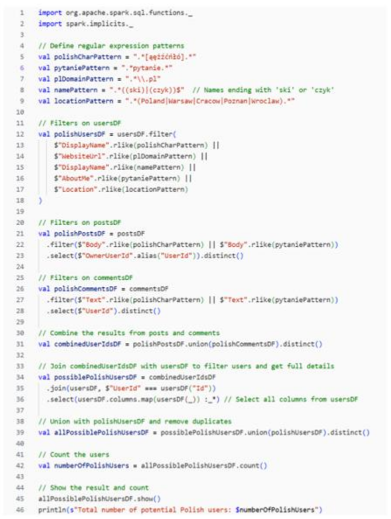
Next, I combined these scores to get a comprehensive view of user engagement. The final step was to identify the top contributor, which I achieved by ordering the combined scores and limiting the results to the single highest score.

The outcome of this analysis highlighted a user, showcased on the right of the slide, who emerged as the Scala guru with an impressive total score, reflecting his expertise and active participation in the Scala community.

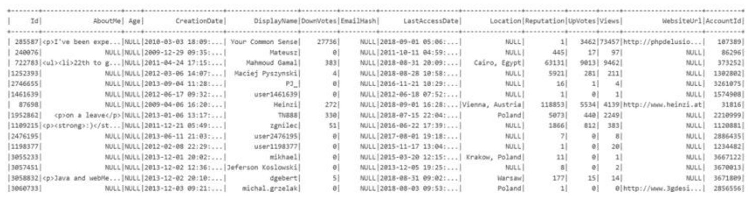
42’’

## Data Analysis

4. Identifying Potential Polish Users



```
import org.apache.spark.sql.functions._
import spark.implicits._

// Define regular expression patterns
val polishCharPattern = ".*[ąężźćńłó].*"
val pytaniePattern = ".*pytanie.*"
val plDomainPattern = ".*\\.pl"
val namePattern = ".*((ski)|(czyk))$"  // Names ending with 'ski' or 'czyk'
val locationPattern = ".*(Poland|Warsaw|Cracow|Poznan|Wroclaw).*"

// Filters on usersDF
val polishUsersDF = usersDF.filter(
    $"DisplayName".rlike(polishCharPattern) ||
    $"WebsiteUrl".rlike(plDomainPattern) ||
    $"DisplayName".rlike(namePattern) ||
    $"AboutMe".rlike(pytaniePattern) ||
    $"Location".rlike(locationPattern)
)

// Filters on postsDF
val polishPostsDF = postsDF
  .filter($"Body".rlike(polishCharPattern) || $"Body".rlike(pytaniePattern))
  .select($"OwnerUserId".alias("UserId")).distinct()

// Filters on commentsDF
val polishCommentsDF = commentsDF
  .filter($"Text".rlike(polishCharPattern) || $"Text".rlike(pytaniePattern))
  .select($"UserId").distinct()

// Combine the results from posts and comments
val combinedUserIdsDF = polishPostsDF.union(polishCommentsDF).distinct()

// Join combinedUserIdsDF with usersDF to filter users and get full details
val possiblePolishUsersDF = combinedUserIdsDF
  .join(usersDF, $"UserId" === usersDF("Id"))
  .select(usersDF.columns.map(usersDF(_)) :_*) // Select all columns from usersDF

// Union with polishUsersDF and remove duplicates
val allPossiblePolishUsersDF = possiblePolishUsersDF.union(polishUsersDF).distinct()

// Count the users
val numberOfPolishUsers = allPossiblePolishUsersDF.count()

// Show the result and count
allPossiblePolishUsersDF.show()
println(s"Total number of potential Polish users: $numberOfPolishUsers")
```



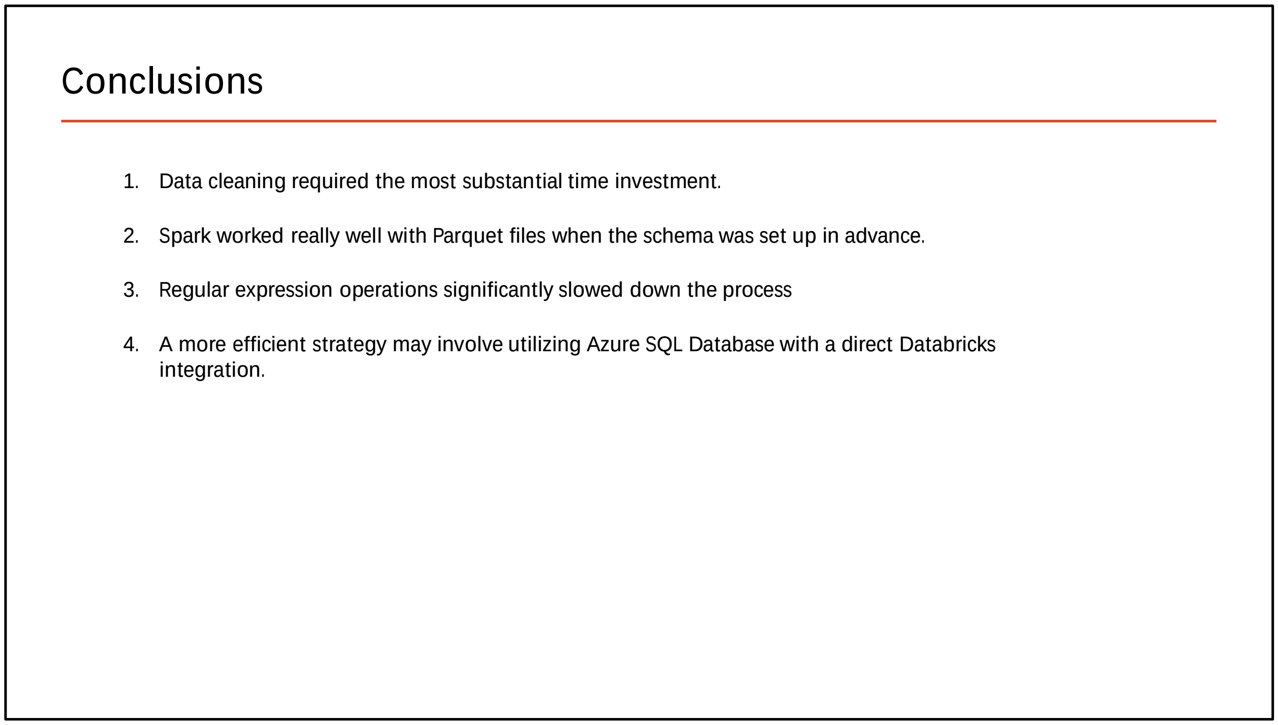
```
+-------+--------------------+----+--------------------+------------------+---------+---------+--------------------+---------------+----------+-------+-----+--------------------+---------+
|     Id|             AboutMe| Age|        CreationDate|       DisplayName|DownVotes|EmailHash|      LastAccessDate|       Location|Reputation|UpVotes|Views|          WebsiteUrl|AccountId|
+-------+--------------------+----+--------------------+------------------+---------+---------+--------------------+---------------+----------+-------+-----+--------------------+---------+
| 285587|<p>I've been expe...|NULL|2010-03-03 18:09:...| Your Common Sense|    27736|     NULL|2018-09-01 05:06:...|           NULL|         1|   3462|73457|http://phpdelusio...|   107389|
| 240076|                NULL|NULL|2009-12-29 09:35:...|           Mateusz|        0|     NULL|2011-10-11 04:59:...|           NULL|       445|     17|   97|                NULL|    86296|
| 722783|<ul><li>22th to g...|NULL|2011-04-24 17:15:...|     Mahmoud Gamal|      383|     NULL|2018-08-31 20:09:...|   Cairo, Egypt|     63131|   9013| 9462|                NULL|   373252|
|1252393|                NULL|NULL|2012-03-06 14:07:...|  Maciej Pyszynski|        4|     NULL|2018-08-28 10:58:...|           NULL|      5921|    281|  211|                NULL|  1302802|
|2746655|                NULL|NULL|2013-09-04 11:28:...|               PJ_|        0|     NULL|2016-11-21 10:29:...|           NULL|        16|      1|    4|                NULL|  3261075|
|1461639|                NULL|NULL|2012-06-17 09:32:...|       user1461639|        0|     NULL|2012-06-18 07:52:...|           NULL|         1|      0|    1|                NULL|  1574908|
|  87698|                NULL|NULL|2009-04-06 16:20:...|            Heinzi|      272|     NULL|2018-09-01 16:28:...|Vienna, Austria|    118853|   5534| 4139|http://www.heinzi.at|    31816|
|1952862|   <p>on a leave</p>|NULL|2013-01-06 13:17:...|             TN888|      330|     NULL|2018-07-15 22:04:...|         Poland|      5073|    440| 2249|                NULL|  2210999|
|1109215|<p><strong>:)</st...|NULL|2011-12-21 05:49:...|           zgnilec|       51|     NULL|2016-06-22 17:39:...|           NULL|      1866|    812|  383|                NULL|  1120881|
|2476195|                NULL|NULL|2013-06-11 21:03:...|       user2476195|        0|     NULL|2017-08-01 19:18:...|           NULL|         7|      0|    8|                NULL|  2886435|
|1198377|                NULL|NULL|2012-02-08 22:29:...|       user1198377|        0|     NULL|2015-11-17 13:04:...|           NULL|         1|      0|   20|                NULL|  1234482|
|3055233|                NULL|NULL|2013-12-01 20:02:...|           mikhael|        0|     NULL|2015-03-20 12:15:...| Krakow, Poland|        11|      0|    1|                NULL|  3667122|
|3057451|                NULL|NULL|2013-12-02 12:36:...|Jeferson Koslowski|        0|     NULL|2013-12-05 19:25:...|           NULL|         8|      0|    2|                NULL|  3670013|
|3058832|<p>Java and webMe...|NULL|2013-12-02 20:10:...|           dgebert|        5|     NULL|2018-08-31 09:02:...|         Warsaw|       177|     15|   14|                NULL|  3671809|
|3060733|                NULL|NULL|2013-12-03 09:21:...|    michal.grzelak|        0|     NULL|2018-08-03 09:53:...|         Poland|         1|      0|    0|http://www.3gdesi...|  2856556|
```

Total number of potential Polish users: 12,454

This slide presents a targeted analysis to pinpoint potential Polish users within a dataset. I began by establishing regex patterns to match Polish names and locations. Then, I applied filters on the user data to identify names or contain typical Polish characters, and also to find users from locations in Poland.

Subsequently, I filtered posts and comments for matches with the same patterns. By combining these filtered datasets, I could create a comprehensive list of users likely to be Polish. The final step was counting these users, which revealed a total of 12,454 potential Polish users.

38’’



## Conclusions

1. Data cleaning required the most substantial time investment.
2. Spark worked really well with Parquet files when the schema was set up in advance.
3. Regular expression operations significantly slowed down the process
4. A more efficient strategy may involve utilizing Azure SQL Database with a direct Databricks integration.

This slide summarizes the key findings from the analysis. The most significant observation is that the bulk of the time was spent on cleaning the data, which underscores the importance of having clean data for accurate analysis. I noted that Spark was highly efficient in handling Parquet files, though it's crucial to provide the correct schema upfront to ensure smooth processing.

Another point of interest is the observation that using regular expressions proved to be quite slow, which suggests the need for more efficient text processing methods in future analyses. Lastly, considering the overall data pipeline performance, deploying a database on Azure SQL Database and connecting it directly to Databricks could offer a more streamlined and faster approach.

42''