Question 1:

Map:

for row in R:

emit<key(b%α, 0→β, 0→γ), value=(a,b,"R")>

for row in S:

emit<key=(b%α, c%γ, 0→γ), value=(b,c,"S")>

for row in T:

emit<key=(0→α, c%β, d%γ), value=(c,d,"T")>

for row in U:

emit<key=(0→α, 0→β, d%γ), value=(d,e,"U")>

سوال ۱) در این مرحله باید یک Hash func طراحی کنیم که جوین
هر ۴ تای آن‌ها را عملی کند که باید گفت Key آن‌ها باید 3 تا مقدار
داشته باشد تا با هم قاطی نشوند حال اگر آن‌ها را به صورت
Mod(a,b) طراحی کنیم باید ۳ تا پارامتر α و β و γ را در نظر بگیریم.
اگر فرض کنیم حاصل ضرب آن‌ها $\alpha\beta\gamma=\theta$ شود داریم:

حال برای reduce باید همچنین مرج کنیم تا تمامی
متغیرهایی که a و e یکسانی دارند را پیدا کنیم.
حال اگر فرض بگیریم $r=s=t=u$ است پس
می‌توان تابع cost را با استفاده از α و β و γ
مینیمم کرد

$$Cost = r+s+t+u+r\beta\gamma+s\gamma+t\alpha+u\alpha\beta \xrightarrow{\text{تابع کل}} \nabla cost = 0 \longrightarrow \alpha=\gamma=\sqrt{\theta},\ \beta=1 \longrightarrow$$

$cost = 4r + 4r\sqrt{\theta} \longrightarrow r=s=t=u$ بازای

$$red\text{-}size = \beta\gamma r + \gamma s + \alpha t + \alpha\beta u = r\sqrt{\theta}$$

Question 2:

Feature	RDD	Dataframe
Level of Abstraction	Low level abstraction	High-level abstraction
Ease of Use	Requires more coding effort	Easier to use with SQL-like operations
Optimization	No optimization. transformations and actions are executed as is.	Highly optimized by Spark's Catalyst Optimizer.
Type Safety	Type-safe. The compiler checks types during compile time, reducing runtime errors.	Not type-safe in standard API. Column names are resolved at runtime, leading to potential runtime errors.
Performance	Slower for structured data because it lacks optimizations like predicate pushdown and code generation.	Faster due to its optimizations and ability to process structured data efficiently.

When to Use Each?

- **Use RDD**:
  1. When you need low-level transformations or actions.
  2. For unstructured data or custom processing logic.
  3. When type safety is critical, and you're not dealing with structured data.
- **Use DataFrame**:
  1. For structured or semi-structured data.
  2. When you prioritize performance and ease of use.
  3. When leveraging Spark SQL for querying data.

Which is Better for Structured Data?

DataFrame is better for structured data because:

1. It provides a tabular format that matches the structure of the data.
2. It is optimized for performance using Catalyst and Tungsten engines.
3. It is easier to write and maintain code with SQL-like syntax.

Question 3:

Methods To increase the efficiency of PySpark :

1. Optimize Resource Utilization

- **Adjust Parallelism**:
  - Use spark.default.parallelism for RDDs and spark.sql.shuffle.partitions for DataFrames.
  - Set these values close to the total number of cores in your cluster for better parallelization.

- **Executor and Core Configuration**:
  - Tune num-executors, executor-memory, and executor-cores to match your workload.
  - Ensure enough memory and avoid overloading each executor.

2. Optimize Data Serialization

- Use **Kryo Serialization** instead of Java serialization:
- spark.conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
- Register custom classes for Kryo if needed for better serialization performance.

3. Minimize Shuffles

- Avoid operations that trigger shuffles, such as groupByKey. Instead, use more efficient alternatives like reduceByKey or aggregateByKey.
- Optimize joins by using **broadcast joins** for smaller datasets:
- from pyspark.sql.functions import broadcast
- large_df.join(broadcast(small_df), "key")

4. Cache and Persist Data Wisely

- Cache intermediate results only when reused multiple times and uncache them when no longer needed:
- df.cache()
- df.unpersist()
- Use appropriate storage levels like MEMORY_AND_DISK.

5. Balance Partition Sizes

- Keep partitions large enough to avoid too many small tasks but not too large to avoid memory overflow.
- Target partition sizes of **128 MB**.

6. Use Broadcast Variables

- Broadcast read-only data to all nodes to avoid repeated transmission:
- broadcast_var = sc.broadcast(large_dict)
- rdd.map(lambda x: broadcast_var.value.get(x))

7. Enable Adaptive Query Execution (AQE)

- For Spark 3.0 and later, enable AQE for better join strategies and partition coalescing:
- spark.conf.set("spark.sql.adaptive.enabled", "true")

8. Monitor and Tune Spark Jobs

- Use the Spark UI to identify bottlenecks and optimize queries.
- Enable event logs to analyze and debug performance issues.

9. Reduce Driver Overhead

- Use mapPartitions() instead of map() to reduce the frequency of function calls.
- Minimize large data collections sent to the driver.

Practical

Question 1:

Part 1: In this question I have simply parsed the file.

```
# Part 1: Parse the JSON string
def parse_json(line):
    return json.loads(line)

parsed_rdd = arxiv_rdd.map(parse_json).filter(lambda x: x is not None)
```

Part 2: In this question I have simply got the fields of the file.

```
# Part 2: Extract and list all fields from the parsed RDD
def extract_fields(rdd):
    # Get a sample JSON object to understand its structure
    sample = rdd.take(1)
    fields = list(sample[0].keys())
    return fields

fields = extract_fields(parsed_rdd)
print("Extracted Fields:", fields)
```

✓ 0.8s

```
Extracted Fields: ['id', 'submitter', 'authors', 'title', 'comments', 'journal-ref', 'doi', 'report-no', 'categories', 'license', 'abstract', 'versions', 'update_date', 'authors_parsed']
```

Question 2:

Part 1: In this question I have simply removed Nan values from the fields.

```
# Part 1: Identify and remove or impute null values
def remove_nulls(record):
    # Check for null or empty fields in the record
    if all(record.values()):  # Ensure no field is None or empty
        return record
    else:
        return None  # Ignore records with null or empty fields

cleaned_rdd = parsed_rdd.filter(lambda x: remove_nulls(x) is not None)
```

Part 2: In this question I have simply removed stop words.

```python
# Define a set of stopwords for removal
stopwords = set(StopWordsRemover.loadDefaultStopWords("english"))

def remove_stopwords(record):
    for key, value in record.items():
        if isinstance(value, str):  # Process only text fields
            words = value.split()
            filtered_words = [word for word in words if word.lower() not in stopwords]
            record[key] = " ".join(filtered_words)
    return record

stopwords_removed_rdd = cleaned_rdd.map(remove_stopwords)
```

Part 3: In this question I have simply removed useless characters.

```python
def remove_useless_characters(record):
    for key, value in record.items():
        if isinstance(value, str):  # Process only text fields
            # Remove special characters, retaining only alphanumeric and spaces
            record[key] = re.sub(r"[^a-zA-Z0-9\s]", "", value)
    return record

final_cleaned_rdd = stopwords_removed_rdd.map(remove_useless_characters)

# Save or view the final cleaned RDD
final_cleaned_rdd.take(1)  # View the first cleaned records
```

Python

```
[{'id': '07040008',
  'submitter': 'Damian Swift',
  'authors': 'Damian C Swift',
  'title': 'Numerical solution shock ramp compression general material properties',
  'comments': 'Minor corrections',
  'journal-ref': 'Journal Applied Physics vol 104 073536 2008',
  'doi': '10106312975338',
  'report-no': 'LAUR072051 LLNLJRNL410358',
  'categories': 'condmatmtrlsci',
  'license': 'httparxivorglicensesnonexclusivedistrib10',
  'abstract': 'general formulation developed represent material models applications dynamic loading Numerical methods devised calculate response shock ramp compression ramp decompression generalizing previou
  'versions': [{'version': 'v1', 'created': 'Sat, 31 Mar 2007 04:47:20 GMT'},
   {'version': 'v2', 'created': 'Thu, 10 Apr 2008 08:42:28 GMT'},
   {'version': 'v3', 'created': 'Tue, 1 Jul 2008 18:54:28 GMT'}],
  'update_date': '20090205',
  'authors_parsed': [['Swift', 'Damian C.', '']]}]
```

Question 3:

Part 1: In this question I have simply counted articles in category field.

```
# Part 1: Count the number of articles in each category
category_counts = parsed_rdd.map(lambda x: (x['categories'], 1)) \
                            .reduceByKey(lambda a, b: a + b) \
                            .collect()
```

Part 2: In this question I have simply found the most articles in category field.

```
# Part 2: Find the category with the most articles
most_articles_category = max(category_counts, key=lambda x: x[1])
print("Category with the most articles:", most_articles_category)
```

```
Category with the most articles: ('astro-ph', 86911)
```

Part 3: In this question I have simply found the distribution of the number of authors and their percentage.

```
# Part 3 (Bonus): Distribution of the number of authors per article
def count_authors(record):
    return len(record['authors'].split(","))

author_counts = parsed_rdd.map(lambda x: (count_authors(x), 1)) \
                          .reduceByKey(lambda a, b: a + b) \
                          .collect()

print("Distribution of the number of authors per article:", author_counts)

# Calculate percentages
total_articles = sum(count for _, count in author_counts)
author_distribution = [(authors, count / total_articles * 100) for authors, count in author_counts]
print("Author distribution (percentage):", author_distribution)
```

Python

```
Distribution of the number of authors per article: [(131, 28), (262, 6), (786, 3), (655, 1), (393, 5), (1, 937445), (132, 26), (263, 11), (656, 4), (394, 13), (1049, 1), (525, 1), (918, 2), (2, 366847), (13:
Author distribution (percentage): [(131, 0.001082672902862587 2), (262, 0.00023200133632769726), (786, 0.00011600066816384863), (655, 3.866688938794954e-05), (393, 0.0001933344469397477), (1, 36.24808212228 6:
```

Part 4: In this question I have simply found the articles with more than 3 authors.

```python
# Part 4 (Bonus): Filter out articles with more than 3 authors and list titles and authors
filtered_articles = parsed_rdd.filter(lambda x: count_authors(x) > 3) \
                              .map(lambda x: (x['title'], x['authors'])) \
                              .take(10)


print("Titles and authors of articles with more than 3 authors (first 10):", filtered_articles)
```

Python

```
Titles and authors of articles with more than 3 authors (first 10): [('Calculation of prompt diphoton production cross sections at Tevatron and\n  LHC energies', "C. Bal\\'azs, E. L. Berger, P. M. Nadolsky,
```

Part 5: In this question I have simply ploted the time series of the number of articles per year.

```python
# Part 5: Time series of the number of articles per year
# Correctly extract the year from the 'versions' field
def extract_year(record):
    # Extract the first 'created' field from the 'versions' list
    created_date = record['versions'][0]['created']
    return created_date[-12:-8]  # Extract the year from the date string

# Compute yearly article counts
yearly_counts = parsed_rdd.map(lambda x: (extract_year(x), 1)) \
                          .filter(lambda x: x[0] is not None) \
                          .reduceByKey(lambda a, b: a + b) \
                          .collect()

# Sort by year for plotting
yearly_counts = sorted(yearly_counts, key=lambda x: x[0])
years, counts = zip(*yearly_counts)

# Plot the time series
plt.figure(figsize=(10, 6))
plt.plot(years, counts, marker='o', linestyle='-', color='b')
plt.title("Number of Articles Submitted Per Year")
plt.xlabel("Year")
plt.ylabel("Number of Articles")
plt.grid()
plt.show()
```

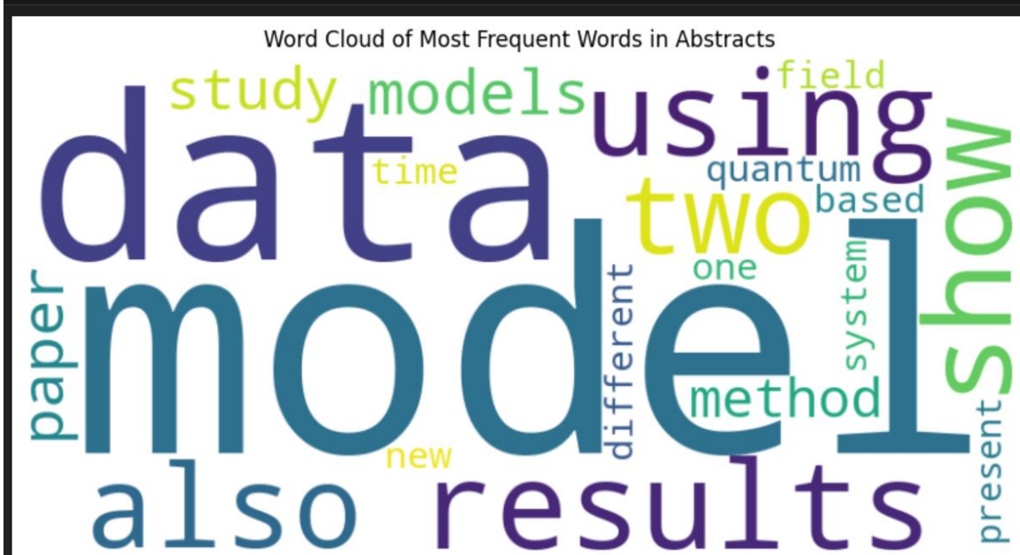
Part 6: In this question I have simply found the most frequent words in abstracts.

```python
# Part 6: Most frequent words in abstracts
# Define a function to extract and clean words from abstracts
def extract_and_clean_words(record):
    # Extract abstract, convert to lowercase, split into words, and clean text
    words = record['abstract'].lower().split()
    cleaned_words = [re.sub(r"[^a-zA-Z0-9]", "", word) for word in words]
    return [word for word in cleaned_words if word and word not in stopwords]

# Extract and clean words from all abstracts
abstract_words = parsed_rdd.flatMap(extract_and_clean_words)

# Count the frequency of each word
word_counts = abstract_words.map(lambda word: (word, 1)) \
                            .reduceByKey(lambda a, b: a + b)

# Get the top 20 most frequent words
most_frequent_words = word_counts.takeOrdered(20, key=lambda x: -x[1])

# Print the results
print("20 most frequent words in abstract:")
for word, count in most_frequent_words:
    print(word, ":", count)
```

```
20 most frequent words in abstract:
model : 1188676
data : 917131
results : 859049
show : 831879
using : 809828
also : 774216
two : 719284
models : 686537
paper : 650231
study : 596891
method : 596084
quantum : 573410
system : 559067
new : 550050
field : 544587
based : 527532
one : 518005
time : 506071
different : 497350
present : 477899
```

Part 7: In this question I have simply made a WordCloud out of the found the most frequent words in abstracts.

```python
# Part 7: Word Cloud for the most frequent words
# Generate WordCloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate_from_frequencies(dict(most_frequent_words))

# Display the Word Cloud
plt.figure(figsize=(10, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.title("Word Cloud of Most Frequent Words in Abstracts")
plt.show()
```



Question 4:

Part 1: In this question I have simply filtered articles containing the word "algorithm" in their abstract.

```
# Part 1: Filter articles containing the word "algorithm" in their abstract
def contains_algorithm(record):
    return 'algorithm' in record['abstract'].lower()

algorithm_articles_rdd = parsed_rdd.filter(contains_algorithm)
```

Part 2: In this question I have simply counted the number of words in each article's abstract.

```
# Part 2: Count the number of words in each article's abstract
def count_words(record):
    return len(record['abstract'].split())

word_count_rdd = algorithm_articles_rdd.map(lambda x: (x['title'], count_words(x)))
```

Part 3: In this question I have simply sorted by word count in descending order and display the top 5 articles.

```
# Part 3: Sort by word count in descending order and display the top 5 articles
top_articles = word_count_rdd.sortBy(lambda x: x[1], ascending=False).take(5)

# Display the results
print("Top 5 articles with the highest word counts in their abstract (containing 'algorithm'):")
for title, word_count in top_articles:
    print(f"Title: {title}, Word Count: {word_count}")
```

```
Top 5 articles with the highest word counts in their abstract (containing 'algorithm'):
Title: The Nonlinearity Coefficient - A Practical Guide to Neural Architecture
  Design, Word Count: 498
Title: Generating a Generic Fluent API in Java, Word Count: 488
Title: Boxicity and Poset Dimension, Word Count: 484
Title: An Anytime Algorithm for Optimal Coalition Structure Generation, Word Count: 484
Title: McMini: A Programmable DPOR-Based Model Checker for Multithreaded
  Programs, Word Count: 475
```