Problem 1

Assumptions:

1. If there is a line (x, y), there will not be another line with (y, x)

Part A

Map function:

$$map : (null, (x, y)) \to [(x, 1), (y, 1)]$$

Reduce function:

$$reduce : (x, [1, 1, \dots]) \to (x, sum([1, 1, \dots]))$$

The file would have each line: $x, \text{count\_of\_friends}$

Part B

Phase 1

Map function:

$$map : (null, (x, y)) \to [(x, y), (y, x)]$$

Reduce function (identity):

$$reduce : (x, [y_1, y_2, \dots]) \to (x, [y_1, y_2, \dots])$$

Intemediary Step (optional)

Filter out all pairs where the length of the friend list < k

Phase 2

Map Function:

$$map : (x_i, [y_1, y_2, \dots]) \to ((y_i, y_j), 1) \forall i, j, i \neq j$$

Additionally, we also need to remove duplicates like (y_i, y_j) and (y_j, y_i). We can do this my sorting the pairs in alphabetically order and then removing the duplicates.

Reduce Function:

$$reduce : ((y_i, y_j), [1, 1, \ldots]) \rightarrow ((y_i, y_j), sum([1, 1, \ldots]))$$

Write the (y_i, y_j) pairs that have key ≥ 7

Part C

Map function:

$$map : (x, y) \rightarrow ((x, 1), (y, 1))$$

Reduce function:

$$reduce : (x, [1, 1, \ldots]) \rightarrow \begin{cases} (x, \text{null}) & \text{with probability } 0.01 \\ \text{nothing} & \text{otherwise} \end{cases}$$

Problem 2

Part A

Consider the computation of $m_n = min a_1, a_2, \ldots, a_n$. In a single MapReduce pass, some reducer, let's call it R_n, must be responsible for computing and outputting the pair (n, m_n).

Assume the input to reducer R_n is determined by fewer than n of the original input values. This implies there exists at least one input a_j (for some $j \in \{1, \ldots, n\}$) that provides no information to R_n.

But this contradicts the definition of R_n because it must use all n input values to compute m_n correctly. Therefore, the input to R_n must be exactly the sequence $(a_1, \ldots, a_n)$ and the reducer size is n.

Part B

Idea

First we make a key observation. To compute m_i, we can do $m_i = min/m_{i-1}, a_i/$ instead of comparing all a_k where $k < i$.

We can do 2 logical steps as follows:

1. Split the input sequence into $\frac{n}{sqrtn}$ equal and ordered chunks and compute the min for the largest index in the chunk.

2. now we will have $\sqrt{n}$ pairs of local minima and we perform the naive version of the algorithm but instead of using all a_i we use only the a_i's that are greater than the largest m_i before the target m_i. Additionally we must use all the previous m_i's as well (but we could compute a running minima of all intermediate m_i's given a third pass). This will make the reducer size in the second pass as most $(\sqrt{n}-1)+(\sqrt{n}-1)$ (the second to last element) which is still $O(\sqrt{n})$

Algorithm

Pass 1 - chunk mins

Break array A into $\sqrt{n}$ equal and ordered chunks $C_1, C_2, \ldots, C_{\sqrt{n}}$. For the mapping function we need to be able to map i to a chunk id c_i. We do this as follows:

$$c_i = \lceil \frac{i}{\sqrt{n}} \rceil$$

Mapping function:

$$map : (i, a_i) \mapsto (c_i, a_i)$$

Reduce function:

$$reduce : (c_i, [a_i, \ldots, a_{i+\sqrt{n}-1}]) \mapsto (c_i, M_i)$$

where $M_i = \min\{[a_i, \ldots, a_{i+\sqrt{n}-1}]\}$

Pass 2 - Assemble

We now have inputs:

1. (c_j, M_j)
2. (i, a_i)

Mapping Function:

map: (i, a_i) \mapsto (c_i, ('val', i, a_i)) \newline
map: (c_j, M_j) \mapsto (c_{k}, ('min', M_j)) \space \forall c_k>c_j

After the shuffle phase:

$$(c_i, [('min', M_1), \ldots, ('min', M_{i-\sqrt{n}})], ('val', i, a_i), \ldots, ('val', i+\sqrt{n}, a_{i+\sqrt{n}}))$$

Reduce Function:

Each reducer is responsible for a single chunk c_i

1. Calculate the running minimum from all minimum chunks:

$$P = \min\{M_j | ('min', M_j) \in list\}$$

2. Initialize local_min as P (if no 'min' tuples like in the first chunk, initialise to infinity)

3. Sort the value tuples $('val', i, a_i)$ by i

4. Iterate through the sorted value tuples

   - for each $('val', i, a_i)$:
     - update $local_min = min(local_min, a_i)$
     - emit the final pair $(i, local_min)$

We then combine the lists from all reducers and return

Problem 3

Assumption:

- We are using a single machine for subsumpling but we could potentially distribute it. However, this will be order of n, so no effect on the big O of communication cost.
- We assume that when we sample we produce dividers that are roughly equal size. This is a fair assumption if we take k to be large enough to preserve the distribution of the data.

Part A

The goal is to broadcast $x_0, \ldots x_{k+1}$ (k+2) numbers to t machines.

The subsampling happens on the master so there is no communication cost involved. We simply pick each number with probability k/n. The broadcast is the only phase we have a communication cost.

This means we are sending a total of $(k+2) \times t = kt \times 2t$ numbers in total. This is $O(kt)$

Part B

The goal here is to compute how many numbers fall into each of the k buckets (exist on each machine).

First we count the number of elements in each bucket on each of the t local machines. We end up with a count for each of the k buckets.

Next, we need to transmit this count back to the master to sum up the total number of elements in each bucket. Here, we send k (or 2k if we are sending pairs) numbers to the master from each of the t machines. G_i is then the sum of the count sent from each machine for each bucket.

Thus, in total we are sending at most $2k \times t$ numbers across the network which is $O(kt)$

Part C

The goal here is to use the global counts that we have from the master to figure out which G_i contains the median and then which element in G_i is the median (r).

We now have $(n_i, count_i)$ on the master. Wer can find G_i and r as follows:

$$
\begin{aligned}
&\text{1. Calculate the median's overall rank} && (1)\\
&N = \sum \text{counts} && (2)\\
&\text{median}\backslash\text{\_rank} = \lceil N/2 \rceil && (3)\\
& && (4)\\
&\text{2. Find the bucket containing the median} && (5)\\
&\text{cumulative}\backslash\text{\_count} = 0 && (6)\\
&\text{for } j \text{ from } 0 \text{ to length(counts)} - 1\text{:} && (7)\\
&\quad // \text{ Check if the median falls within the current bucket} && (8)\\
&\quad \text{if } (\text{cumulative}\backslash\text{\_count} + \text{counts}[j]) \geq \text{median}\backslash\text{\_rank:} && (9)\\
&\qquad // \text{ 3. Calculate the rank 'r' within this bucket} && (10)\\
&\qquad r = \text{median}\backslash\text{\_rank} - \text{cumulative}\backslash\text{\_count} && (11)\\
&\qquad \text{return } (j, r) \text{ // Return the bucket index and the rank} && (12)\\
& && (13)\\
&\quad // \text{ If not found, add the current bucket's count to the total} && (14)\\
&\quad \text{cumulative}\backslash\text{\_count} = \text{cumulative}\backslash\text{\_count} + \text{counts}[j] && (15)
\end{aligned}
$$

There is no communication cost since this is all done on the master.

Part D

We now know G_j and r so we need to collect an ordered list of numbers in G_j to find the median.

The master instructs all t machines to send it only the numbers they have that belong to bucket G_j. The total communication is the total number of elements in G_j, which is n_j.

The network cost would be the expected size of n_j. Since we chose k samples, the data would be divided into $k+1$ chunks of roughly equal size on average.

Thus the network communication cost is approximately $\frac{n}{k+1}$ which is $O(\frac{n}{k})$

Once the master has the list of numbers in G_j, we sort it and return the element at index r. This is the median.

Part E

From the above parts we know communication cost is:

$$TotalCost(k) \approx O(kt) + O(kt) + O(n/k) = O(kt + n/k)$$

To minimize network cost we need to find k that minimizes $f(k) = kt + n/k$

$$f'(k) = t - \frac{n}{k^2}$$

Set $f'(k) = 0$ to find the minimum:

0 = t-\frac{n}{k^2} \newline
k^2=\frac{n}{t} \newline
k = \sqrt{\frac{n}{t}}

Thus we choose $k = \sqrt{\frac{n}{t}}$ to minimize network cost.

Problem 4

Config

In [9]:
```
# Import necessary libraries for Spark
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.window import Window
```

In [10]:
```
# Initialize Spark session for local machine
spark = SparkSession.builder \
    .appName("Assignment1_Problem4") \
    .master("local[*]") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
    .getOrCreate()

# Set log level to basically no verbose output
spark.sparkContext.setLogLevel("ERROR")

print(f"Spark version: {spark.version}")
print(f"Spark UI available at: {spark.sparkContext.uiWebUrl}")
```

```
Spark version: 3.5.4
Spark UI available at: http://10.228.244.25:4040
```

In [11]:
```
datasources = {
    'links' : spark.read.csv('data/links.csv', header=True, inferSchema=True
    'movies' : spark.read.csv('data/movies.csv', header=True, inferSchema=Tr
    'ratings' : spark.read.csv('data/ratings.csv', header=True, inferSchema=
    'tags' : spark.read.csv('data/tags.csv', header=True, inferSchema=True)
}
```

Part A

- Code Function: Calculates the mean number of ratings received per movie across the entire dataset.
- Implementation:
  - Groups ratings by movieID and counts ratings per movie
  - Computes the overall average of these counts
  - Output: 10.37 ratings per movie on average

In [12]:
```
df = datasources['ratings']

avg_count = (
    df.groupBy('movieID').agg(count('rating').alias('count')) #count ratings
    .agg(avg('count')) # get the average counts
    .collect()[0][0] # get the average
)
print('Average number of ratings per movie: ',avg_count)
```

```
Average number of ratings per movie:  10.369806663924312
```

Part B

- Code Function: Determines which movie genres receive the highest average ratings from users.
- Implementation:
  - Joins ratings and movies datasets on movieID
  - Splits pipe-separated genres (Action|Comedy|Drama) into individual rows using explode()
  - Each movie has many genres. we will assume that the movie appears in all of the genres it is classified in as a single entry to compute the average rating of the genre.
  - Groups by genre and calculates average rating
  - Sorts results in descending order
- Key Findings:
  - Film-Noir (3.92) - Highest rated genre
  - War (3.81) - Second highest
  - Documentary (3.80) - Third highest
  - Horror (3.26) - Lowest rated genre

In [13]:
```
movies = datasources['movies']
ratings = datasources['ratings']

# join genre on movie id in ratings
df = (
    ratings.join(
        movies.select('movieID', 'genres'),
        on='movieID',
        how='left'
    )
)

# explode the genres column
df = (df
    # first cast g1|g2|g3 to a list
    .withColumn('genres', split('genres', '\|'))
    # explode the list into multiple rows
    .withColumn('genres', explode('genres'))
)

# groupby and get average rating for each genre
genre_avg = (
    df.groupBy('genres')
    .agg(avg('rating').alias('avg_rating')) # get average rating for each ge
    .sort('avg_rating', ascending=False) # sort by average rating
)

genre_avg.show()
```

```
+------------------+------------------+
|            genres|        avg_rating|
+------------------+------------------+
|         Film-Noir| 3.920114942528736|
|               War|   3.8082938876312|
|       Documentary| 3.797785069729286|
|             Crime| 3.658293867274144|
|             Drama|3.6561844113718758|
|           Mystery| 3.632460255407871|
|         Animation|3.6299370349170004|
|              IMAX| 3.618335343787696|
|           Western| 3.583937823834197|
|           Musical|3.5636781053649105|
|         Adventure|3.5086089151939075|
|           Romance|3.5065107040388437|
|          Thriller|3.4937055799183425|
|           Fantasy|3.4910005070136894|
|(no genres listed)|3.4893617021276597|
|            Sci-Fi| 3.455721162210752|
|            Action| 3.447984331646809|
|          Children| 3.412956125108601|
|            Comedy|3.3847207640898267|
|            Horror| 3.258195034974626|
+------------------+------------------+
```

Part C

- Code Function: Identifies the highest-rated movies within each genre category.
- Implementation:
  - Similar genre explosion technique as Part B
  - Groups by movieID, title, and genres to get per-movie averages
  - Uses window functions (row_number() with partitionBy) to rank movies within each genre
  - Filters to show only top 3 movies per genre
- Key Findings:
  - Many genres have multiple movies with perfect 5.0 ratings
  - Examples include "Black Mirror", "Sonatine", "12 Chairs (1976)"
  - Shows that niche or less-rated movies can achieve perfect scores
- Technical Note: The ranking uses row_number() over a window partitioned by genre and ordered by average rating (descending).

In [14]:

```
movies = datasources['movies']
ratings = datasources['ratings']

# we need ratings, movies, and genres
df = (
    ratings.join(
        movies.select('movieID', 'genres', 'title'),
        on='movieID',
        how='left'
    )
)

# explode the genres column
df = (df
    # first cast g1|g2|g3 to a list
    .withColumn('genres', split('genres', '\|'))
    # explode the list into multiple rows
    .withColumn('genres', explode('genres'))
)

movie_avg = (
    df.groupBy('movieID', 'title', 'genres')
    .agg(avg('rating').alias('avg_rating')) # get average rating for each mo
    # keep only the top 3 movies in each genre
    .withColumn('rank',
        row_number().over(Window.partitionBy('genres').orderBy(desc('avg_rat
    .filter('rank <= 3')
    .sort('genres', 'rank')
    .drop('movieID')
)

movie_avg.show(60)
```

```
+--------------------+------------------+-----------------+----+
|               title|            genres|       avg_rating|rank|
+--------------------+------------------+-----------------+----+
|        Black Mirror|(no genres listed)|              5.0|   1|
|Death Note: Desu ...|(no genres listed)|              5.0|   2|
|The Adventures of...|(no genres listed)|              5.0|   3|
|Sonatine (Sonachi...|            Action|              5.0|   1|
|    Knock Off (1998)|            Action|              5.0|   2|
|    Max Manus (2008)|            Action|              5.0|   3|
|    12 Chairs (1976)|         Adventure|              5.0|   1|
|Junior and Karlso...|         Adventure|              5.0|   2|
|Asterix and the V...|         Adventure|              5.0|   3|
|My Life as McDull...|         Animation|              5.0|   1|
|Into the Forest o...|         Animation|              5.0|   2|
|Winnie the Pooh a...|         Animation|              5.0|   3|
|The Fox and the H...|          Children|              5.0|   1|
|Wow! A Talking Fi...|          Children|              5.0|   2|
|Junior and Karlso...|          Children|              5.0|   3|
|Unfaithfully Your...|            Comedy|              5.0|   1|
|What Happened Was...|            Comedy|              5.0|   2|
|       Presto (2008)|            Comedy|              5.0|   3|
|American Friend, ...|             Crime|              5.0|   1|
|Little Murders (1...|             Crime|              5.0|   2|
|Trailer Park Boys...|             Crime|              5.0|   3|
|Tickling Giants (...|       Documentary|              5.0|   1|
|Zeitgeist: Moving...|       Documentary|              5.0|   2|
|Martin Lawrence L...|       Documentary|              5.0|   3|
|Sisters (Syostry)...|             Drama|              5.0|   1|
|Thousand Clowns, ...|             Drama|              5.0|   2|
|Duel in the Sun (...|             Drama|              5.0|   3|
| L.A. Slasher (2015)|           Fantasy|              5.0|   1|
|       Presto (2008)|           Fantasy|              5.0|   2|
|My Left Eye Sees ...|           Fantasy|              5.0|   3|
|Rififi (Du rififi...|         Film-Noir|             4.75|   1|
|Long Goodbye, The...|         Film-Noir|4.666666666666667|   2|
|You Only Live Onc...|         Film-Noir|              4.5|   3|
| Maniac Cop 2 (1990)|            Horror|              5.0|   1|
|      Buzzard (2015)|            Horror|              5.0|   2|
|Galaxy of Terror ...|            Horror|              5.0|   3|
|         More (1998)|              IMAX|              5.0|   1|
|Happy Feet Two (2...|              IMAX|              5.0|   2|
|Journey to the We...|              IMAX|             4.75|   3|
|Woman Is a Woman,...|           Musical|              5.0|   1|
|Into the Woods (1...|           Musical|              5.0|   2|
| True Stories (1986)|           Musical|              5.0|   3|
| 'Salem's Lot (2004)|           Mystery|              5.0|   1|
|The Adventures of...|           Mystery|              5.0|   2|
|7 Faces of Dr. La...|           Mystery|              5.0|   3|
|Moscow Does Not B...|           Romance|              5.0|   1|
|Crossing Delancey...|           Romance|              5.0|   2|
|My Left Eye Sees ...|           Romance|              5.0|   3|
|SORI: Voice from ...|            Sci-Fi|              5.0|   1|
|Mystery of the Th...|            Sci-Fi|              5.0|   2|
|A Detective Story...|            Sci-Fi|              5.0|   3|
|Supercop 2 (Proje...|          Thriller|              5.0|   1|
|  Hellbenders (2012)|          Thriller|              5.0|   2|
```

```
| Maniac Cop 2 (1990)|          Thriller|                 5.0|   3|
|     Mephisto (1981)|               War|                 5.0|   1|
|Come and See (Idi...|               War|                 5.0|   2|
|Battle Royale 2: ...|               War|                 5.0|   3|
|Duel in the Sun (...|           Western|                 5.0|   1|
|Trinity and Sarta...|           Western|                 5.0|   2|
|7 Faces of Dr. La...|           Western|                 5.0|   3|
+--------------------+------------------+--------------------+----+
```

Part D

- Code Function: Identifies users who have rated the most movies.
- Implementation:
  - Groups ratings by userID and counts total ratings per user
  - Sorts in descending order by count
- Key Findings:
  - User 414: Most active with 2,698 ratings
  - User 599: Second with 2,478 ratings
  - User 474: Third with 2,108 ratings
  - Top 10 users range from 1,055 to 2,698 ratings
- Insight: Shows significant variation in user engagement, with power users rating hundreds more movies than typical users.

In [15]:

```
ratings = datasources['ratings']

user_movies = (
    ratings.select('userID', 'movieID')
    .groupBy('userID')
    .agg(count('movieID').alias('count_rated')) # count num ratings for each
    .sort('count_rated', ascending=False) # sort by num ratings
)

user_movies.show(10)
```

```
+------+-----------+
|userID|count_rated|
+------+-----------+
|   414|       2698|
|   599|       2478|
|   474|       2108|
|   448|       1864|
|   274|       1346|
|   610|       1302|
|    68|       1260|
|   380|       1218|
|   606|       1115|
|   288|       1055|
+------+-----------+
only showing top 10 rows
```

Part E

- Code Function: Finds pairs of users with the most movies rated in common (collaborative filtering foundation).
- Implementation:
  - Creates user-movie lists using collect_list() aggregation
  - Performs cross-join between users to create all possible user pairs
  - Uses array_intersect() to find common movies between user pairs
  - Calculates intersection cardinality and ranks by similarity
- Key Findings:
  - Users 414 & 599: Highest similarity with 1,338 movies in common
  - Users 414 & 474: Second highest with 1,077 movies in common
  - User 414 appears frequently in top pairs (consistent with being most active)
  - Technical Significance: This analysis forms the basis for collaborative filtering recommender systems, where users with similar viewing histories receive similar recommendations.

In [16]:
```
ratings = datasources['ratings']

print('num_users:',ratings.select('userID').agg(countDistinct('userID')).col

# first we get a dataframe with (userID, list_of_movies_rated)
user_movies = (
    ratings.select('userID', 'movieID')
    .groupBy('userID')
    .agg(collect_list('movieID').alias('movies_rated'))
)

# now we need to check for every user pair what the intersection of their mo
# first do a cross join and remove rows where user1 == user2
user_pairs = (
    user_movies.alias('u1')
    .crossJoin(user_movies.alias('u2'))
    .filter('u1.userID != u2.userID')
    .select(
        col('u1.userID').alias('userID_u1'),
        col('u1.movies_rated').alias('movies_rated_u1'),
        col('u2.userID').alias('userID_u2'),
        col('u2.movies_rated').alias('movies_rated_u2')
    )
)

# now we need to create another column with the intersection between movies_
# also get the cardinality of the intersection
user_pairs = (
    user_pairs
    .withColumn(
        'movies_rated_intersection',
        array_intersect('movies_rated_u1', 'movies_rated_u2')
```

```
    )
    .withColumn(
        'intersection_cardinality',
        size('movies_rated_intersection')
    )
)

# rename cols and drop unnessary columns. sort by intersection_cardinality
user_pairs = (
    user_pairs
    .withColumnRenamed('userID_u1', 'user_1')
    .withColumnRenamed('userID_u2', 'user_2')
    .drop('movies_rated_u1', 'movies_rated_u2', 'movies_rated_intersection')
    .sort('intersection_cardinality', ascending=False)
)


user_pairs.show(10)
```

```
num_users: 610
[Stage 77:>                                                          (0 + 1)
/ 1]
+------+------+------------------------+
|user_1|user_2|intersection_cardinality|
+------+------+------------------------+
|   414|   599|                    1338|
|   599|   414|                    1338|
|   414|   474|                    1077|
|   474|   414|                    1077|
|    68|   414|                     950|
|   414|    68|                     950|
|   414|   448|                     914|
|   448|   414|                     914|
|   274|   414|                     856|
|   414|   274|                     856|
+------+------+------------------------+
only showing top 10 rows
```