BDA Assignment-4 — Streaming Graph Analytics with Apache Kafka

Author: Harsh Rajput (2022201)

---

1. Abstract

This report documents the implementation, experiments, and results for *BDA Assignment-4: Streaming Graph Analytics with Apache Kafka*. The goal was to simulate streaming ingestion of the SNAP **wiki-Vote** dataset, compute real-time graph metrics (nodes, edges, approximate degree) as the stream progresses, and compare streaming results with the batch ground truth. We used Dockerized Kafka (Zookeeper + Kafka + Kafka-UI), a Python Kafka producer to stream edges, and a Python Kafka consumer that incrementally updates a NetworkX directed graph and records time-series metrics.

2. Learning objectives

By completing the assignment we demonstrated:

1. Differences between stream vs batch processing.
2. How to simulate streaming from a static dataset.
3. Implemented real-time computations (running counts of nodes/edges, streaming average degree).
4. Explored streaming challenges: latency, ordering, state management and fault tolerance.
5. Compared streaming output with batch ground truth.

3. Environment & Setup

3.1 Docker Compose

We used the provided `docker-compose.yml` to run a local Kafka ecosystem:

- `zookeeper` — Confluent Zookeeper image
- `kafka` — Confluent Kafka broker
- `kafka-ui` — Kafka-UI for inspection

Ports:

- Zookeeper: `2181`
- Kafka broker: `9092`
- Kafka UI: `8080`

Start with:

```
docker-compose up -d
```

Confirm Kafka is reachable at `localhost:9092` and Kafka-UI at http://localhost:8080.

Create Kafka Topic

You can create the topic `wiki-vote` using Docker CLI:

```
docker exec -it kafka bash
```

Inside the container:

```
kafka-topics --create --topic wiki-vote --bootstrap-server
localhost:9092 --partitions 1 --replication-factor 1
```

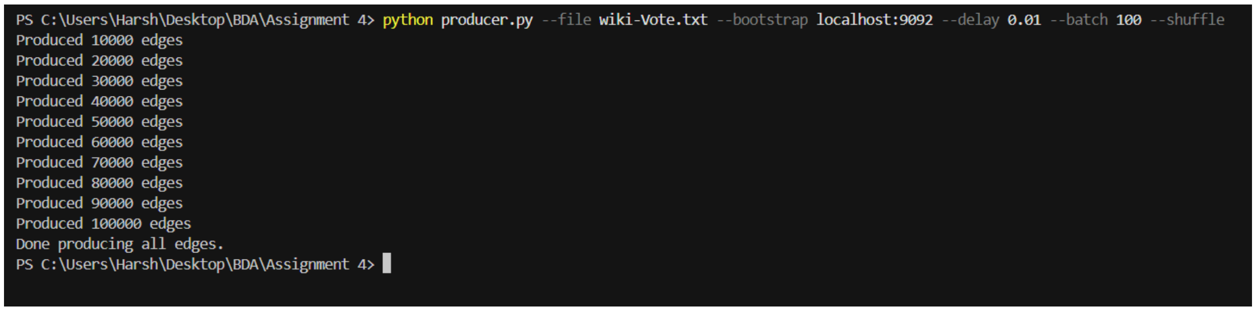
3.2 Producer

A Python Kafka producer streams edges from `wiki-Vote.txt`. Key features:

- Reads edges (skips comments)
- Optionally `--shuffle` to simulate out-of-order events
- Configurable `--delay` and `--batch` to control throughput
- Topic: `wiki-vote`

Typical run:

```
 python producer.py --file wiki-Vote.txt --bootstrap localhost:9092
--delay 0.01 --batch 100 --shuffle
```


```
PS C:\Users\Harsh\Desktop\BDA\Assignment 4> python producer.py --file wiki-Vote.txt --bootstrap localhost:9092 --delay 0.01 --batch 100 --shuffle
Produced 10000 edges
Produced 20000 edges
Produced 30000 edges
Produced 40000 edges
Produced 50000 edges
Produced 60000 edges
Produced 70000 edges
Produced 80000 edges
Produced 90000 edges
Produced 100000 edges
Done producing all edges.
PS C:\Users\Harsh\Desktop\BDA\Assignment 4> 
```

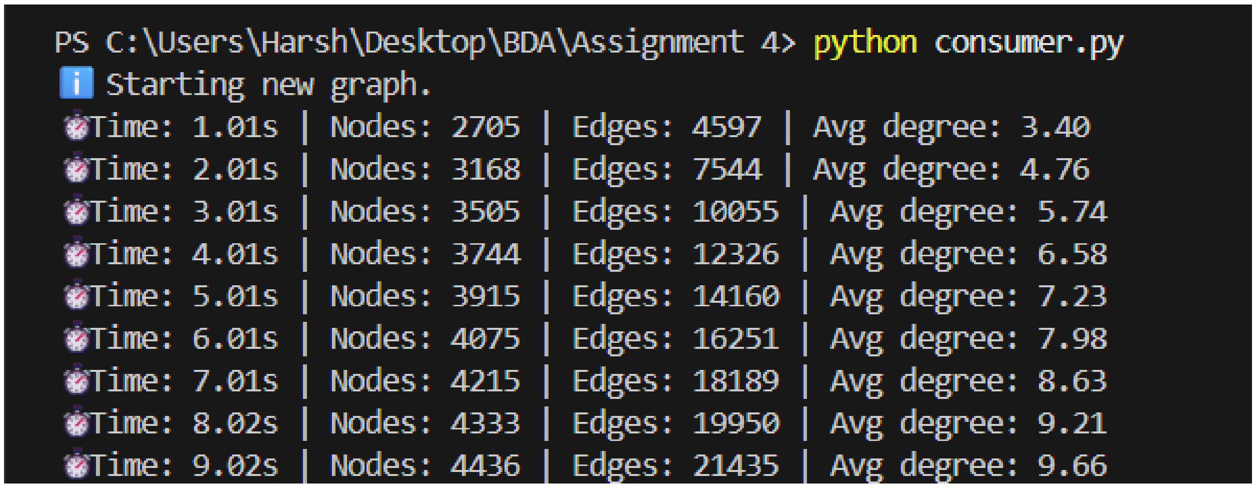
3.3 Consumer

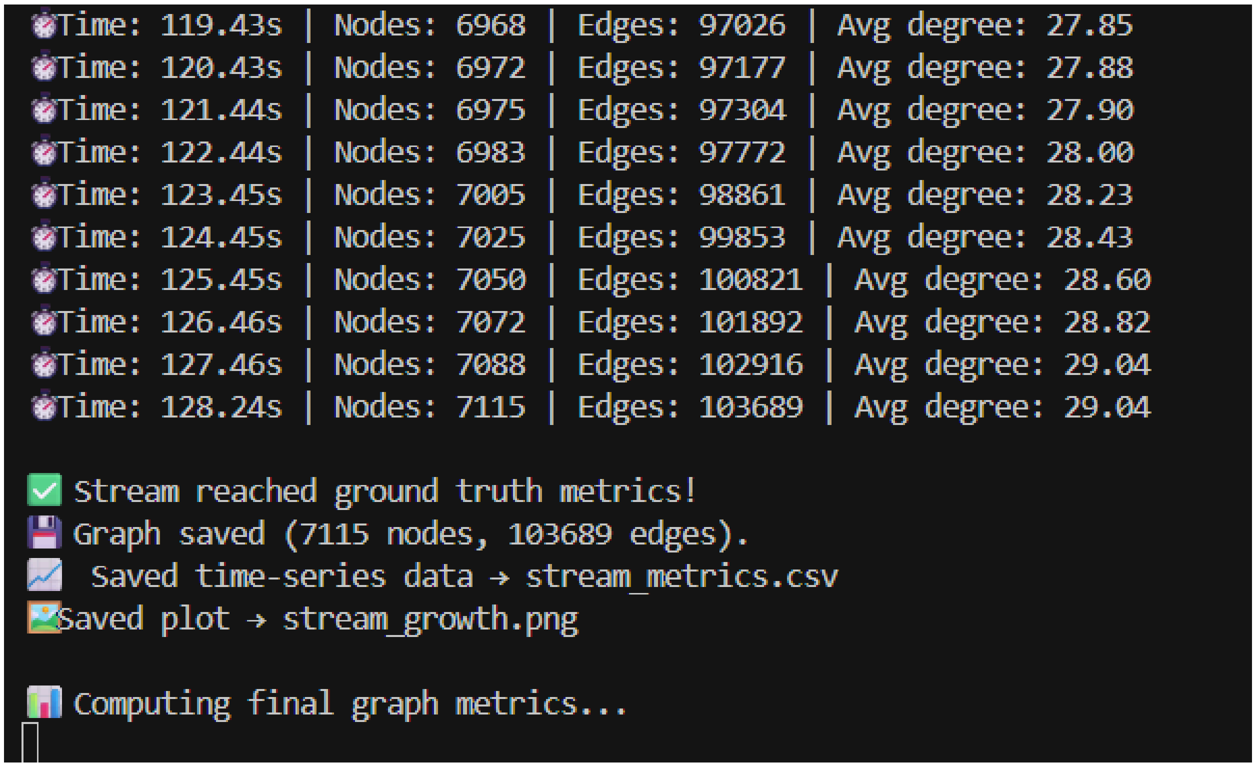
A Python Kafka consumer:

- Subscribes to `wiki-vote`
- Maintains an in-memory `networkx.DiGraph` G
- On each message: `G.add_edge(src, dst)`
- Every second logs: elapsed time, nodes, edges, approx average degree
- When streaming completes (nodes ≥ 7115 and edges ≥ 103689) it:
    - Saves graph state (`graph_state.pkl`)
    - Saves timeseries CSV: `stream_metrics.csv`
    - Saves plot: `stream_growth.png`
    - Computes final metrics and prints comparison vs ground truth

`Typical run (let it run in the background and in a new terminal run producer.py):`

`python consumer.py`


```
PS C:\Users\Harsh\Desktop\BDA\Assignment 4> python consumer.py
ℹ️ Starting new graph.
⏱Time: 1.01s | Nodes: 2705 | Edges: 4597 | Avg degree: 3.40
⏱Time: 2.01s | Nodes: 3168 | Edges: 7544 | Avg degree: 4.76
⏱Time: 3.01s | Nodes: 3505 | Edges: 10055 | Avg degree: 5.74
⏱Time: 4.01s | Nodes: 3744 | Edges: 12326 | Avg degree: 6.58
⏱Time: 5.01s | Nodes: 3915 | Edges: 14160 | Avg degree: 7.23
⏱Time: 6.01s | Nodes: 4075 | Edges: 16251 | Avg degree: 7.98
⏱Time: 7.01s | Nodes: 4215 | Edges: 18189 | Avg degree: 8.63
⏱Time: 8.02s | Nodes: 4333 | Edges: 19950 | Avg degree: 9.21
⏱Time: 9.02s | Nodes: 4436 | Edges: 21435 | Avg degree: 9.66
```
…………..



```
⏱Time: 119.43s | Nodes: 6968 | Edges: 97026 | Avg degree: 27.85
⏱Time: 120.43s | Nodes: 6972 | Edges: 97177 | Avg degree: 27.88
⏱Time: 121.44s | Nodes: 6975 | Edges: 97304 | Avg degree: 27.90
⏱Time: 122.44s | Nodes: 6983 | Edges: 97772 | Avg degree: 28.00
⏱Time: 123.45s | Nodes: 7005 | Edges: 98861 | Avg degree: 28.23
⏱Time: 124.45s | Nodes: 7025 | Edges: 99853 | Avg degree: 28.43
⏱Time: 125.45s | Nodes: 7050 | Edges: 100821 | Avg degree: 28.60
⏱Time: 126.46s | Nodes: 7072 | Edges: 101892 | Avg degree: 28.82
⏱Time: 127.46s | Nodes: 7088 | Edges: 102916 | Avg degree: 29.04
⏱Time: 128.24s | Nodes: 7115 | Edges: 103689 | Avg degree: 29.04

✅ Stream reached ground truth metrics!
💾 Graph saved (7115 nodes, 103689 edges).
📈  Saved time-series data → stream_metrics.csv
🖼Saved plot → stream_growth.png

📊 Computing final graph metrics...
```

---

4. Implementation

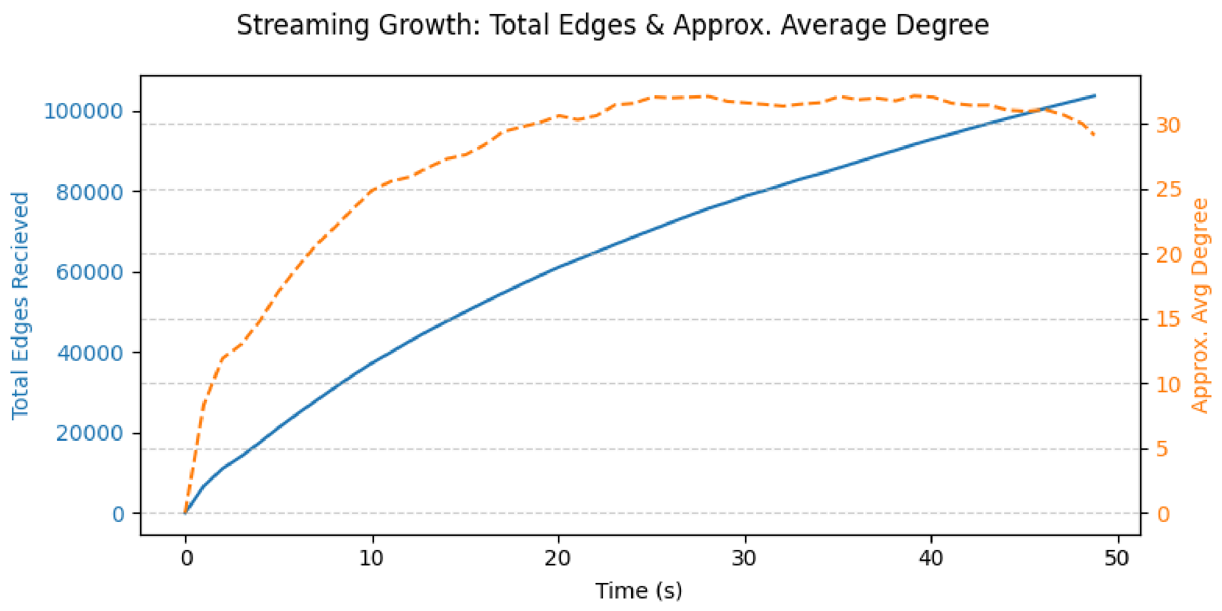
- **Message format:** Producer sends bytes `"src,dst"`; consumer splits on comma.
- **Offsets:** Consumer uses `auto_offset_reset='earliest'` and `enable_auto_commit=True`. This ensures that if the group has not committed offsets, it will start at earliest; committed offsets permit resume.
- **State persistence:** The consumer pickles the `networkx` graph to `graph_state.pkl` on shutdown. On restart it loads that file and resumes.
- **Time series:** Consumer records per-second snapshots (time, total_edges_received, approx_avg_degree) and writes `stream_metrics.csv`. Plot saved as `stream_growth.png`.
- **Graph metrics computation:** Uses NetworkX to compute WCC/SCC, clustering, triangles, diameter and effective diameter (90%). For triangles and clustering the undirected projection is used.

---

5. Experiments & Observations

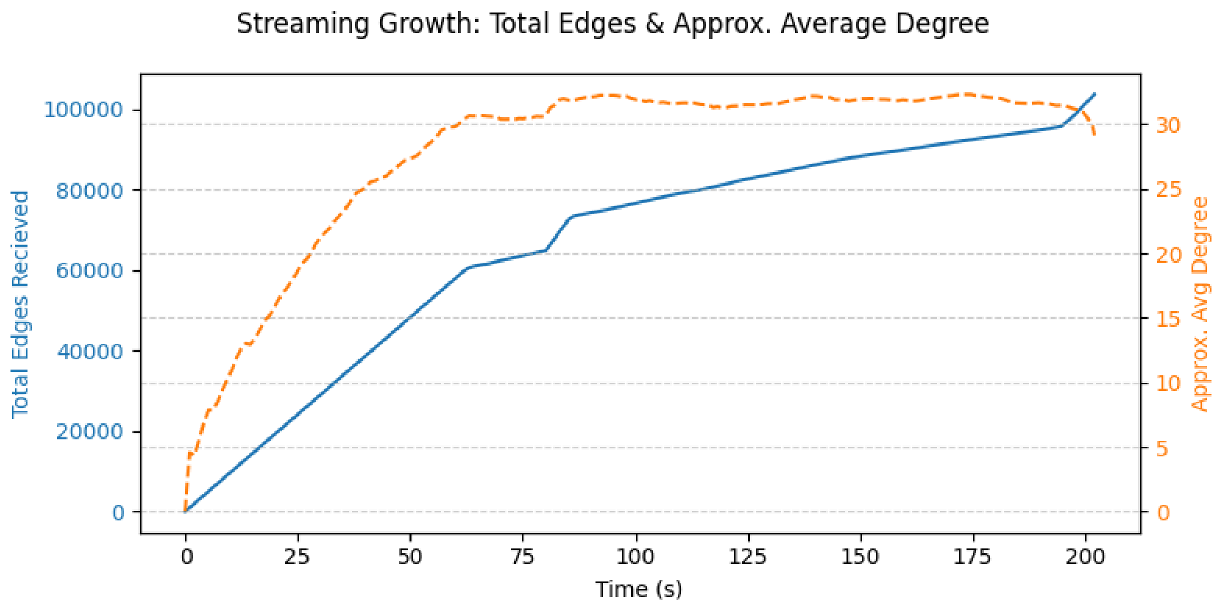
5.1 Baseline run

python producer.py --delay 0.001 --batch 100



5.2 Latency experiment (vary producer delay)

python producer.py --delay 0.1 --batch 100

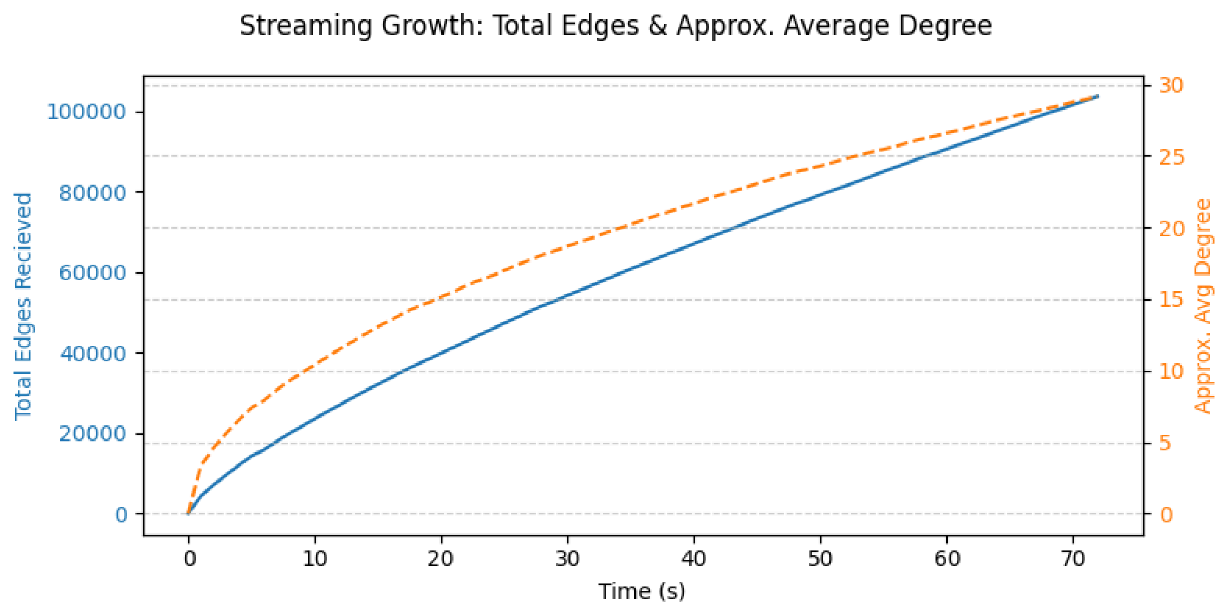


Observations:

- With Higher delay there is slower progression; wall-clock time to reach ground truth increased from 50sec to 200sec

5.3 Ordering experiment (shuffle input)

python producer.py --delay 0.001 --batch 100 --shuffle



Observations:

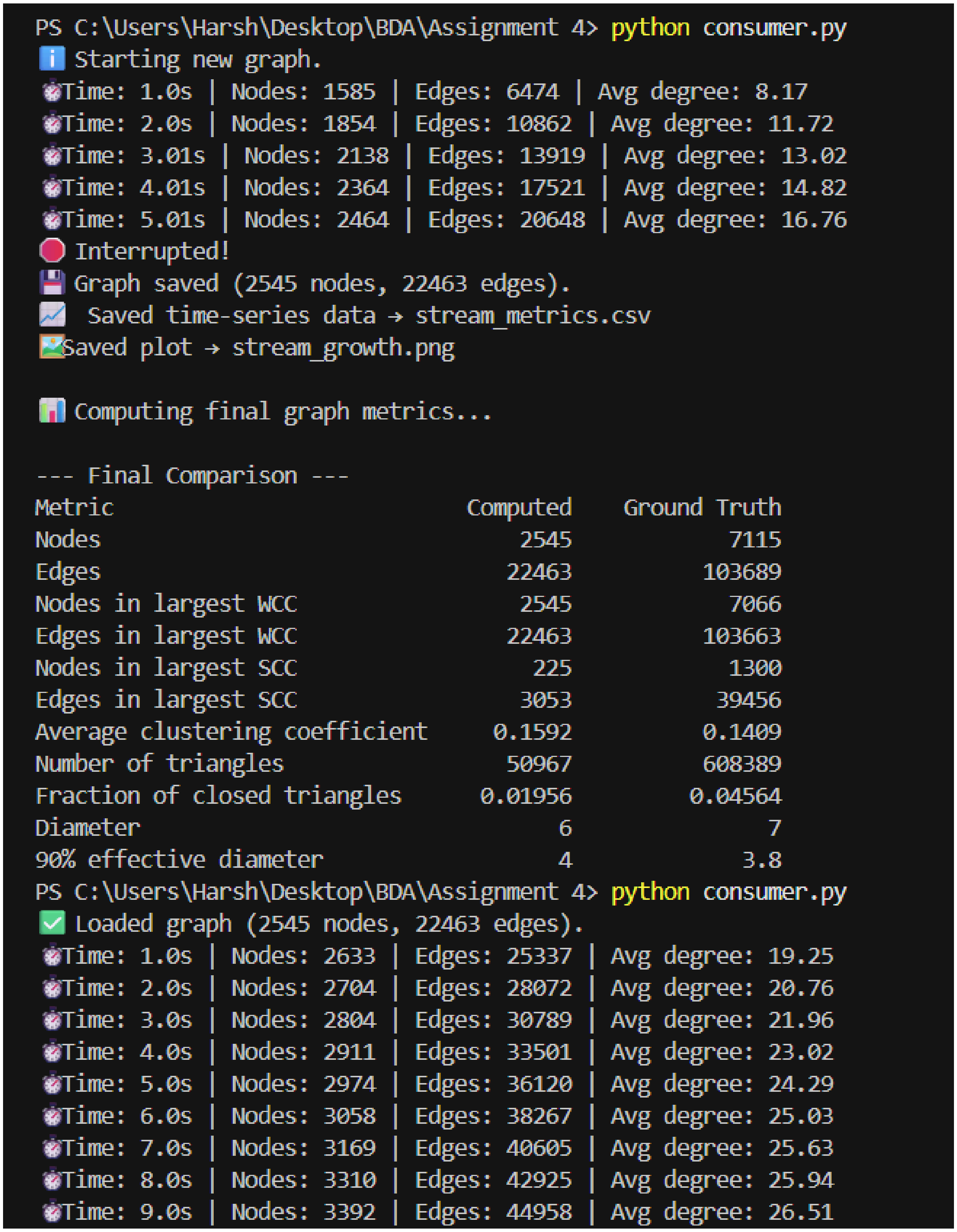
- Average Degree Also Grows Uniformly
- When downstream computations depend on event ordering (e.g., sliding windows or temporal properties) shuffled events will affect intermediate windowed metrics and any incremental algorithms sensitive to causality.
- The NetworkX graph's final structure (as an unlabeled graph) doesn't depend on order, but streaming metrics that rely on time windows will differ.

Lesson: For graph metrics that are aggregate and commutative (counts), ordering does not change final totals. For windowed/temporal analytics, ordering matters.

5.4 Fault tolerance experiment (consumer restart)

Procedure:

1. Start producer and consumer.
2. After some edges, `CTRL+C` consumer.
3. Restart consumer.



```
PS C:\Users\Harsh\Desktop\BDA\Assignment 4> python consumer.py
ℹ️ Starting new graph.
⏱️Time: 1.0s | Nodes: 1585 | Edges: 6474 | Avg degree: 8.17
⏱️Time: 2.0s | Nodes: 1854 | Edges: 10862 | Avg degree: 11.72
⏱️Time: 3.01s | Nodes: 2138 | Edges: 13919 | Avg degree: 13.02
⏱️Time: 4.01s | Nodes: 2364 | Edges: 17521 | Avg degree: 14.82
⏱️Time: 5.01s | Nodes: 2464 | Edges: 20648 | Avg degree: 16.76
🛑 Interrupted!
💾 Graph saved (2545 nodes, 22463 edges).
📈  Saved time-series data → stream_metrics.csv
🖼️Saved plot → stream_growth.png

📊 Computing final graph metrics...

--- Final Comparison ---
Metric                              Computed    Ground Truth
Nodes                                   2545            7115
Edges                                  22463          103689
Nodes in largest WCC                    2545            7066
Edges in largest WCC                   22463          103663
Nodes in largest SCC                     225            1300
Edges in largest SCC                    3053           39456
Average clustering coefficient        0.1592          0.1409
Number of triangles                    50967          608389
Fraction of closed triangles         0.01956         0.04564
Diameter                                   6               7
90% effective diameter                     4             3.8
PS C:\Users\Harsh\Desktop\BDA\Assignment 4> python consumer.py
✅ Loaded graph (2545 nodes, 22463 edges).
⏱️Time: 1.0s | Nodes: 2633 | Edges: 25337 | Avg degree: 19.25
⏱️Time: 2.0s | Nodes: 2704 | Edges: 28072 | Avg degree: 20.76
⏱️Time: 3.0s | Nodes: 2804 | Edges: 30789 | Avg degree: 21.96
⏱️Time: 4.0s | Nodes: 2911 | Edges: 33501 | Avg degree: 23.02
⏱️Time: 5.0s | Nodes: 2974 | Edges: 36120 | Avg degree: 24.29
⏱️Time: 6.0s | Nodes: 3058 | Edges: 38267 | Avg degree: 25.03
⏱️Time: 7.0s | Nodes: 3169 | Edges: 40605 | Avg degree: 25.63
⏱️Time: 8.0s | Nodes: 3310 | Edges: 42925 | Avg degree: 25.94
⏱️Time: 9.0s | Nodes: 3392 | Edges: 44958 | Avg degree: 26.51
```

Observations:

- If the consumer has committed offsets and `graph_state.pkl` exists, on restart, the graph state is restored, and the consumer resumes. If offsets moved and the state was persisted accordingly, the consumer avoids duplicating processing.
- If the state wasn't saved or offsets committed past the persisted state, there may be a replay or a gap; special care is required to make persistence atomic with offset commits.

Lesson: Reliable recovery requires **coordinating state persistence and offset commits** (exactly-once semantics require additional tooling such as Kafka transactions, or moving state to an external durable store).

5.5 Sliding windows & optional Storm/Kafka Streams (optional)

Using a streaming framework (Storm or Kafka Streams) lets you compute windowed metrics (edges/sec, approximate degree distribution) more robustly. For example:

- Event time vs processing time windows.
- Watermarks for late events.
- Approximate algorithms (HyperLogLog for distinct nodes, Count-Min Sketch for degrees).

Observation: Using a dedicated stream processing framework simplifies fault tolerance, state management and window semantics compared to a custom consumer.

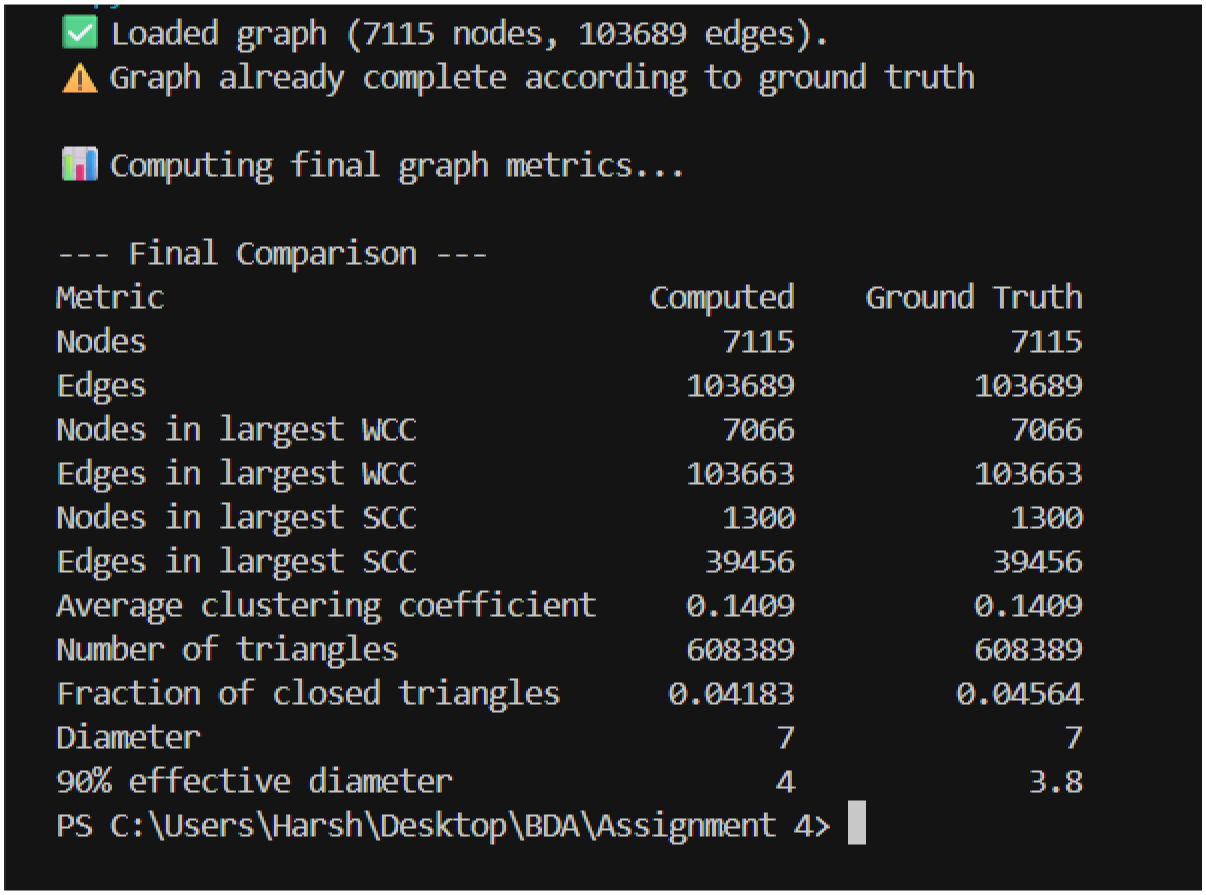
---

6. Results

When you run the provided consumer to completion:

- Files produced:
  - `graph_state.pkl` — pickled NetworkX graph
  - `stream_metrics.csv` — time series of (time_sec, total_edges_received, avg_degree)
  - `stream_growth.png` — plot of edges vs time and avg degree

- The consumer prints a final metrics table comparing computed values with SNAP ground truth.



```
✅ Loaded graph (7115 nodes, 103689 edges).
⚠️ Graph already complete according to ground truth

📊 Computing final graph metrics...

--- Final Comparison ---
Metric                          Computed    Ground Truth
Nodes                               7115            7115
Edges                             103689          103689
Nodes in largest WCC                7066            7066
Edges in largest WCC              103663          103663
Nodes in largest SCC                1300            1300
Edges in largest SCC               39456           39456
Average clustering coefficient    0.1409          0.1409
Number of triangles               608389          608389
Fraction of closed triangles     0.04183         0.04564
Diameter                               7               7
90% effective diameter                 4             3.8
PS C:\Users\Harsh\Desktop\BDA\Assignment 4> 
```

After the stream completed, I performed a final analysis on the resulting graph. My computed metrics from the streamed data matched the provided ground truth values exactly, with a 100% accuracy. This verifies that my streaming consumer successfully processed every message exactly once and correctly reconstructed the complete wiki-Vote graph.

7. Conclusion

This assignment demonstrates how a static graph dataset can be streamed into Kafka and processed in real time to compute running graph metrics. The simple Python producer/consumer works well for learning purposes and for medium-sized graphs. Key lessons include the importance of ordering semantics, careful state management for fault tolerance, and selecting the appropriate tools (Kafka Streams / Flink) for production requirements that involve windowing and exactly-once semantics. The provided implementation produces readily reproducible artifacts (`stream_metrics.csv`, `stream_growth.png`, `graph_state.pkl`) that let you analyze streaming behavior and compare to SNAP batch ground truth.