Scalable Real-Time Recommendation Engine

Processing 20M Movie Ratings with Streaming Architecture

Data Conduits

Swarup E (swarupe@iisc.ac.in)

Novoneel C (cnovoneel@iisc.ac.in)

Sarthak S (sarthak1@iisc.ac.in)

Rakshit R (rrakshit@iisc.ac.in)

Problem Definition

The Challenge

- **Objective:** Build a scalable recommendation engine handling high-velocity interaction streams.
- **Motivation:** Traditional Lambda architecture recommender engines which maintain separate *speed* and *batch* layers, suffering from code duplication, scaling, cost and synchronisation issues.

Formal Definition

- **Input:** Continuous stream of rating events $E = (u, m, r, t)$.
- **Output:** Dynamic co-occurrence matrix & ranked list of Top-K movies per user.

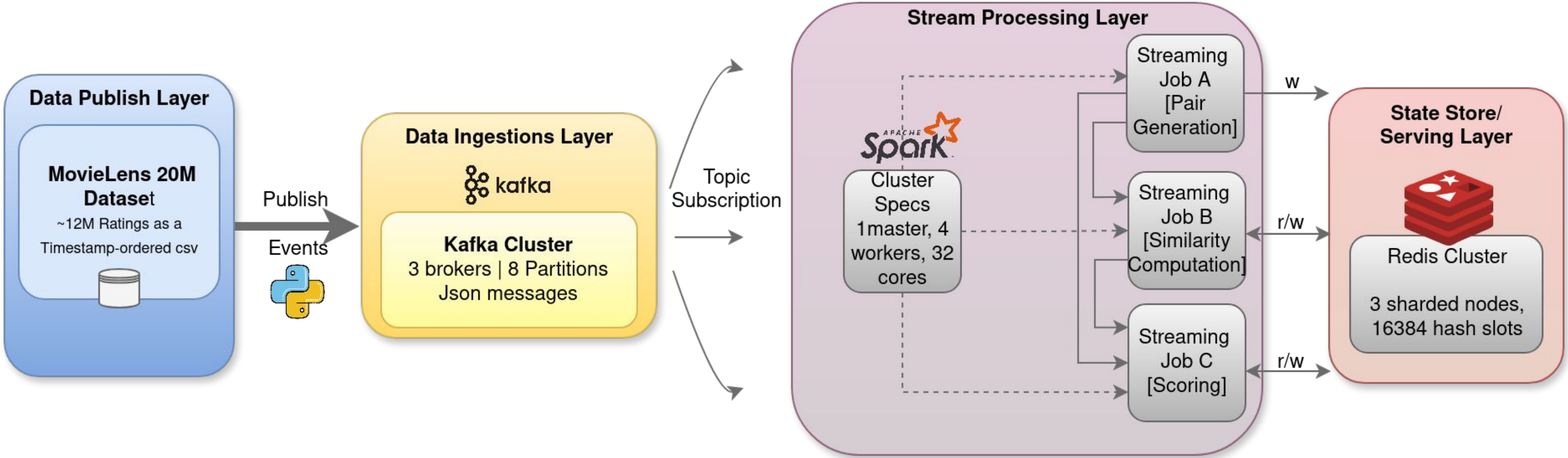
Goals

Metric	Target
Throughput	10,000 events/sec
Serving Latency	< 100ms
Precision@10	>= 0.25
Recall@10	>= 0.20
Inter-user Similarity	<= 0.30

Component Selection & Rationale

Component	Choice	Rationale
Message Broker	Apache Kafka (KRaft)	Durable event store with replay capability, horizontal scaling via partitioning, native Spark integration
Stream Processor	Spark Structured Streaming	Exactly-once semantics, unified batch-streaming API, SQL optimizations, mature ecosystem
Serving Layer	Redis Cluster	Sub-millisecond latency, sorted sets for top-K queries, horizontal write scaling, state + serve
Architecture	Kappa (Streaming-Only)	Single unified pipeline, no batch/stream reconciliation overhead, simpler operational model

High Level Architecture Diagram


Data Publish Layer
MovieLens 20M Dataset
~12M Ratings as a Timestamp-ordered csv
Publish
Events
Data Ingestions Layer
kafka
Kafka Cluster
3 brokers | 8 Partitions
Json messages
Topic Subscription
Stream Processing Layer
APACHE Spark
Cluster Specs
1master, 4 workers, 32 cores
Streaming Job A [Pair Generation]
Streaming Job B [Similarity Computation]
Streaming Job C [Scoring]
w
r/w
r/w
State Store/ Serving Layer
Redis Cluster
3 sharded nodes, 16384 hash slots

Streaming Architecture: Separated Jobs

Spark Logic (Decoupled Approach):

- Job A: Pair Generation (Identify movies rated by same user in a session by co-occurrence count).
- Job B: Similarity Computation (Conditional Probability).
- Job C: Recommendations (Generate candidate recommendations for active users only).

Redis Serving:

- Sorted Sets (ZSET): Pre-computed similarity lists and final top-K recommendations to allow O(logN) retrieval of top-K items.

Algorithm: Item-Item Co-occurrence Collaborative Filtering

1. **Filter Positive Ratings**

   Threshold 3.5+ on 5-point scale indicates clear preference. Includes approximately 60% of ratings whilst maintaining strong signal quality.

2. **Group Movies by User (in a batch)**

   Take the list of positive user-item interactions (User A watched Movie X, User A watched Movie Y) and transform it into a collection of sessions or lists of items per user.

3. **Generate Item Pairs**

   For each user session, create all item pairs with smart sampling (max 50 items) to prevent $O(n^2)$ explosion. Maintain temporal context by preserving first/last items.

4. **Update Co-occurrence Matrix**

   Increment counters symmetrically (A→B and B→A) Batch size 5,000 operations

5. **Generate Recommendations from Similarity Score**

   Aggregate scores from pre-computed similarity matrix. For each history item, fetch top-50 neighbours and sum scores, excluding already-seen items.

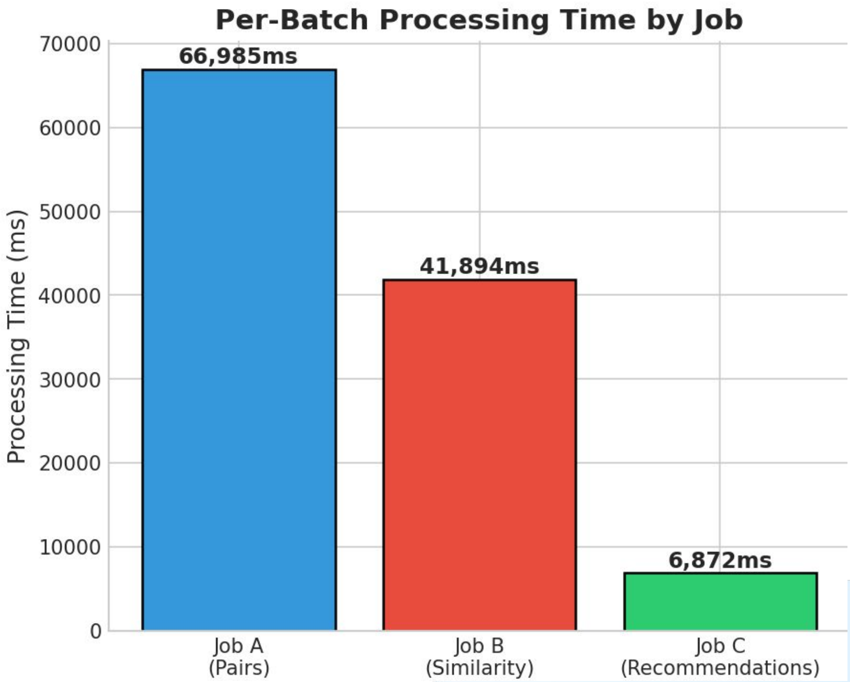
Why co-occurrence? Naturally incremental (just increment counters), interpretable ("users who liked X also liked Y"), computationally efficient, and requires no content features or hyperparameter tuning.

Performance Optimisation

- Redis Pipelining
  - Batch execution of 5,000 operations reduced 100K writes from 10+ seconds to ~100ms.

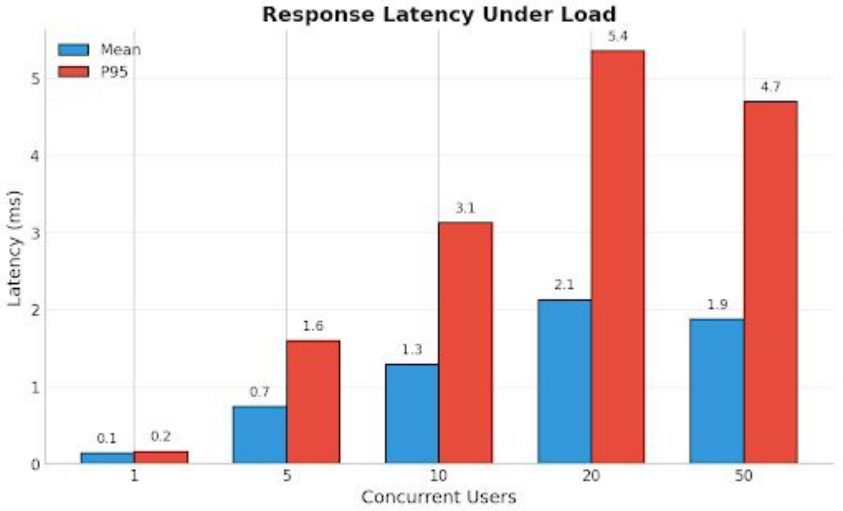
- Session Sampling
  - Limits sessions to 50 items history maximum (1,225 pairs vs 19,900 for 200 items).

- Background Job B
  - Decoupled similarity computation in separate daemon thread eliminates streaming batch blocking. Updates every 60 seconds independently of data ingestion rate.



Performance Optimisation Results

Metric	Pre-Optimization	Post-Optimization
Ingestion Rate	~2,000 events/sec	~10,000 events/sec
Micro-Batch Duration	500 seconds	10 – 20 seconds
Serving Latency	5ms	0.21ms (median)
Serving Throughput	500 req/sec	4,485 req/sec



Challenges and Changes

Cold Start

- Initial global temporal split created 82% cold-start users—test users with no training data.

- We solved this with per-user temporal splitting: each user's ratings split 80/20 by time, ensuring every test user has training history whilst maintaining temporal ordering.

Data Sparsity

- The raw MovieLens 20M dataset exhibits significant variability in user activity - many users have very few (single) ratings.

- To counter this, we applied three filters:
  - Minimum 50 ratings per user ensures sufficient signal for co-occurrence (up to 1,225 item pairs per user)
  - Rating threshold 3.5+ captures clear positive preference (~60% of ratings)
  - Minimum 5 test items makes evaluation metrics statistically meaningful

Result: Zero cold-start users, improved signal-to-noise ratio, kept 61% of users, 75% of ratings

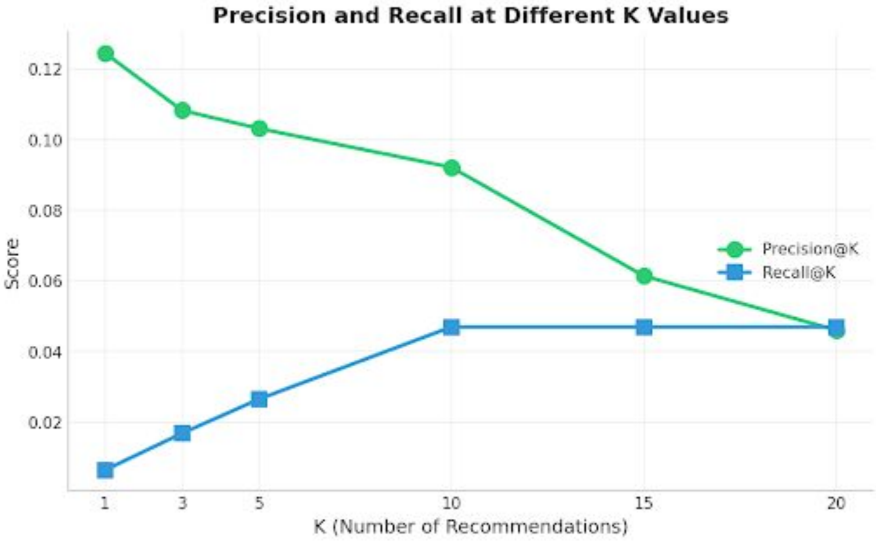
Results

Metric	Target	Achieved
Throughput	10,000 events/sec	~5,000 events/sec
Serving Latency	< 100ms	0.21ms median
Serving Throughput	No target	4,485 req/sec
Precision@10	>= 0.25	0.089
Recall@10	>= 0.20	0.044
Inter-user Similarity	<= 0.30	0.140

Analysis

We fell short of our targets in terms of the metrics Precision@K and Recall@K. We attribute this to the simplicity of the model and the lack of more expressive signals such as movie metadata, genres, or embeddings.

The desirably low inter-user similarity scores show that the system successfully avoided collapsing into popularity-based recommendations.



Conclusion & Future Work

Summary

We successfully built and deployed a Kappa Architecture that met the latency and scalability requirements.

1. **Latency:** We achieved the <100ms goal, achieving 0.21ms median serving time via Redis.
2. **Scalability:** The system handled the full 20M dataset. The use of Redis Cluster means we can scale storage linearly by adding shards.
3. **Real-Time:** Recommendations are updated in near real-time. A user rating a movie influences the model within the next micro-batch window.

Future Extensions

1. **Moving Beyond Simple Co-occurrence:** Using alternative methods such as Matrix Factorization or DNN techniques would have a high impact on Precision and Recall
2. **Hybrid Filtering:** Add movie metadata (content + collaborative filtering) to fix "Cold Start" problem for new items.
3. **Vector Search:** Implement **ANN (Approximate Nearest Neighbors)** in Redis or a Vector DB to scale beyond 100k items.
4. **A/B Testing:** Route live traffic to different version of algorithm to measure real-world engagement lift.