https://doi.org/10.26160/2474-5901-2026-52-75-82

OPTIMIZATION METHODS FOR DISTRIBUTED QUEUE PERFORMANCE IN HIGH-LOAD STREAMING SYSTEMS FOR MODERATION AND INFORMATION RETRIEVAL

Bogutskii A.D.
Perplexity AI, Inc., London, UK

Keywords: distributed message queues, stream moderation, broker architecture, load balancing, scaling, predictive routing, fault tolerance, Kafka.
Abstract. This article explores a set of architectural and algorithmic methods for optimizing the performance of distributed message queues in high-load streaming systems focused on content moderation and information retrieval tasks. Particular attention is given to scalability, resilience, and the organization of fault-tolerant message delivery under conditions of variable and intensive data flow. An overview of architectural solutions of popular platforms is provided, with emphasis on their applicability to real-time stream processing. The effectiveness of load balancing algorithms is analyzed, including static partitioning and predictive scaling methods based on machine learning.

МЕТОДЫ ОПТИМИЗАЦИИ ПРОИЗВОДИТЕЛЬНОСТИ РАСПРЕДЕЛЕННЫХ ОЧЕРЕДЕЙ В ВЫСОКОНАГРУЖЕННЫХ ПОТОКОВЫХ СИСТЕМАХ МОДЕРАЦИИ И ИНФОРМАЦИОННОГО ПОИСКА

Богуцкий А.Д.
Perplexity AI, Inc., Лондон, Великобритания

Ключевые слова: распределенные очереди сообщений, потоковая модерация, архитектура брокеров, балансировка нагрузки, масштабирование, предиктивная маршрутизация, отказоустойчивость, Kafka.
Аннотация. В данной статье исследуется комплекс архитектурных и алгоритмических методов оптимизации производительности распределенных очередей сообщений в высоконагруженных потоковых системах, ориентированных на задачи модерации и информационного поиска. Особое внимание уделяется особенностям масштабирования, устойчивости и организации отказоустойчивой доставки сообщений в условиях переменной и интенсивной нагрузки. Приводится обзор архитектурных решений популярных платформ, с акцентом на их применимость к задачам стриминговой обработки данных в реальном времени. Анализируется эффективность алгоритмов балансировки нагрузки, таких как статическое партиционирование и методы предиктивного масштабирования на основе машинного обучения.

Introduction

Contemporary real-time event processing systems have become a major component for a wide range of applications. With the growth of user activity, especially in multimedia and social platforms, both the volume of incoming messages and the complexity of their processing increase substantially. Under such conditions, architectures capable of handling millions of events per second with minimal latency and scalable resource usage in the presence of variable load become particularly important.

The key part of such systems is distributed message queues, ensuring reliable and asynchronous data transfer between microservices. The performance of such systems directly influences overall system response time, analysis completeness, and moderation decision accuracy. But high event throughput causes bottlenecks, such as an unbalanced load distribution pattern at brokers and increased response times. Such problems are especially significant in those involving a high need for adherence to service-level agreements (SLA) and fault tolerance under high event throughput processing capacity.

In response to these problems, recent years have seen active development of architectural and algorithmic approaches aimed at improving the performance of distributed queues. Methods such as horizontal scaling, intelligent message routing or and dynamic resource allocation have been widely embraced. Industry interest in these mechanisms continues to expand, which is evidenced by the general deployment of Kafka-based brokers and cloud messaging services. The purpose of article is to provide an in-depth assessment of methods for improving the performance of distributed message queues in real-time content moderation and information retrieval systems.

Main part. Architectural principles of distributed message queues

Modern event processing systems rely on the architecture of distributed message queues, which enable asynchronous interaction between components and ensure resilience to failures. Queues act as a buffer between producers and consumers, allowing the system to absorb load spikes and maintain uninterrupted operation under high pressure [1].

They provide enhanced fault-tolerance and load-balancing capabilities, effectively buffering messages and distributing workloads, which enables components to scale horizontally and operate independently from one another [2]. This decomposition of the system into independent services (microservices) and the use of asynchronous communication eliminates bottlenecks typical for synchronous request–response designs and increases the overall reliability of the architecture.

A key priority is to ensure even distribution of workloads across cluster nodes. The performance of a distributed stream-processing system directly depends on how well the load is balanced among its machines [3]. A static sharding strategy based on fixed rules often fails to cope with dynamically changing data flows, leading to uneven broker utilization and reduced throughput. To address these issues, adaptive load-balancing protocols have been proposed.

One of the advanced solutions for dynamic load balancing in streaming systems is the **SWARM protocol**, designed for adaptive routing and partition redistribution under fluctuating request intensity. Its architecture is built upon a separation of routing and execution. The global routing subsystem analyzes incoming data and query streams, mapping them to distributed indexes. The management modules on execution nodes perform adaptive workload redistribution in real time (fig. 1).

The advantages of this approach are particularly evident in scenarios with uneven distribution of data and requests, which is typical for moderation systems in

social media platforms. Similar approaches, such as STAR, Tornado, Ameoba, and others, redefine partition allocation within the cluster “on the fly” to equalize load and sustain high performance.

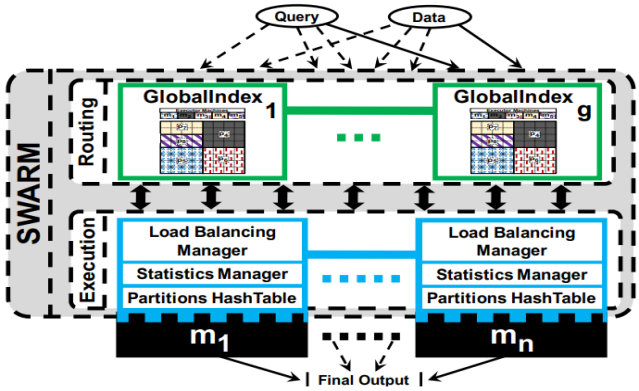

Fig. 1. The architecture of SWARM [4]

Implementations of distributed message queues vary widely in architectural design. **Apache Kafka** follows a model based on monolithic brokers with coordination through ZooKeeper (or the internal KRaft protocol), where each topic is divided into partitions – individual log files placed across different nodes. In contrast, **Apache Pulsar** adopts a separation-of-concerns architecture in which brokers handle message streams, while BookKeeper nodes are responsible for persistent data storage (fig. 2).

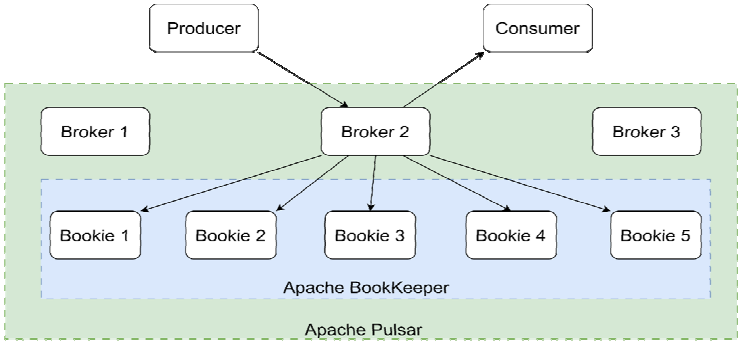

Fig. 2. Apache Pulsar architecture with broker–BookKeeper separation

This decoupling enables compute and storage resources to be scaled separately, which is quite significant, especially when under variable load conditions. This implies that, in a Pulsar cluster, new brokers can be easily introduced under load conditions without necessarily rebalancing all the already stored messages. Pulsar shows comparable performance and can handle over half of what Kafka does, with many advanced functions such as namespace multi-tenancy and ease of geo-replication, among others.

Such architectural designs, combining processing parallelism with data duplication, form the foundation of modern high-performance messaging systems. They enable smooth scaling of cluster throughput and ensure resilience in the event of individual node failures.

Load balancing methods and adaptive scaling

Optimization of distributed message queues in streaming systems is impossible without a clear load-balancing strategy. Under high event intensity, the overload of even a single node can lead to a cascading degradation of performance across the entire system. Therefore, ensuring an even distribution of messages among processing units becomes a critical first step toward improving overall efficiency [5].

The most widely used mechanism is **topic partitioning** – a static scheme in which a logical message stream is divided into independent partitions, with each partition processed by a single consumer at a time within a consumer group. At the same time, a consumer may handle one or multiple partitions. This model enables horizontal scalability: as load increases, new consumers can be added. They will automatically receive a portion of the partitions for processing. The broker transparently routes messages into the appropriate partitions based on predefined rules, for example, using a hash of the message key. This eliminates competition between consumers for the same message, reduces the likelihood of overloading a particular node, and increases overall system throughput.

In addition to static partitioning, **dynamic algorithms for message redistribution** are employed. Recent research demonstrates the use of machine learning for such tasks. For example, in an AIoT-oriented stream separation system, a DDPG-based deep reinforcement learning algorithm has been proposed to optimize queue configuration and partition selection. This approach provides a noticeable increase in throughput compared to simpler methods due to adaptive load balancing (fig. 3).

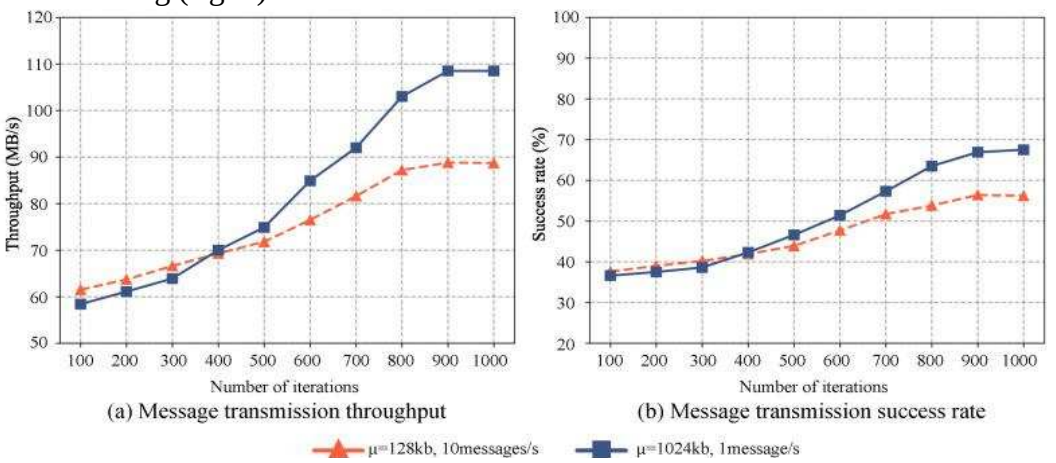

Fig. 3. The performance of message transmission with different quantities and characteristics after optimized with DDPG [6]

Fault tolerance in distributed message queues is commonly achieved through message replication and the maintenance of cluster state consistency. Messages are typically replicated across multiple brokers. It allows the system to continue serving clients even in the presence of node failures and to preserve ordering guarantees and data integrity at the partition level.

To further enhance stability, modern distributed queue architectures incorporate additional mechanisms for reliable message delivery. Systems designed for high-throughput stream processing employ strategies such as automatic retry, message redirection on failure, and isolation of problematic items into dedicated processing channels. When temporary failures occur, a message is not discarded but moved into a specialized segment or auxiliary queue, from which it can be reprocessed once the system components recover.

A notable example of an advanced fault-tolerance model in this context is Apache Kafka, which provides a well-defined delivery semantics model. By default, Kafka follows an at-least-once delivery approach, where message duplication is possible under failure conditions, and ordering guarantees are maintained only within individual partitions. An important advancement in this area is the introduction of **exactly-once processing semantics in Apache Kafka**, which are strictly guaranteed in Kafka-to-Kafka processing pipelines under transactional execution (fig. 4).

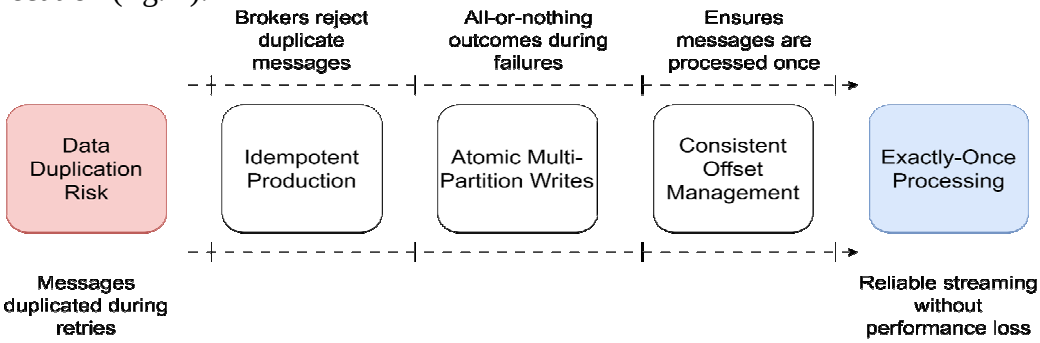

Fig. 4. Achieving exactly-once semantics in Kafka [7]

This mechanism is achieved through idempotent producers and the transactional API, which guarantee that messages are not duplicated even in cases of retries or network failures. The integration of idempotency with transactional processing helps maintain data integrity in the presence of node failures, replication events, or consumer-group rebalancing. Naturally, these reliability guarantees come at the cost of additional computational overhead.

Effective load balancing and adaptive scaling form the foundational layer of resilience in distributed queues under variable stream intensity. The combination of static partitioning, dynamic resource redistribution, and intelligent optimization techniques enables stable performance even during sharp workload fluctuations.

Latency minimization and resilience under high-intensity processing

In high-load streaming systems, not only throughput but also message delivery time is critical. The delay between a producer and a consumer directly affects the timeliness of processing and the system's ability to react to important data. Various techniques are used to minimize latency, including binary compact message formats to reduce traffic volume, tuning of polling or push mechanisms for rapid message retrieval, and optimization of network connections (for example, the use of RDMA or long-lived TCP connections via gRPC).

In addition, advanced routing methods are applied in clustered systems: high-priority messages may be processed first, and the queues themselves can be implemented in memory or on high-performance SSD storage. Latency and throughput characteristics depend not only on network-level optimizations and data transmission formats but also on the fundamental architectural choices underlying a particular queuing system. Different platforms vary in how effectively they handle increasing traffic volumes, priority processing, and storage mechanisms, which results in noticeable differences in their behavior under typical workloads (tab. 1).

Tab. 1. Messaging framework performance analysis across all workloads [8]

Service	W1: E-commerce			W2: IoT Ingestion		
	Throughput (K msg/sec)	P95 Latency (ms)	Availability (%)	Throughput (K msg/sec)	P95 Latency (ms)	Availability (%)
Apache Kafka	1,247 ± 23	18.2±2.1	99.97±0.01	1,856 ± 41	12.8±1.4	99.94±0.02
RabbitMQ	478 ± 12	32.4±3.2	99.91±0.03	623 ± 18	28.7±2.9	99.89±0.01
Apache Pulsar	892 ± 18	22.1±2.3	99.95±0.02	1,234 ± 28	18.9±1.8	99.92±0.03
Amazon SQS	312 ± 16	120.5±12.1	99.91±0.03	445 ± 25	105.2±10.3	99.89±0.04
Google Pub/Sub	367 ± 18	78.2±7.1	99.87±0.05	523 ± 28	69.8±6.3	99.84±0.06

Decoupling system components through message queues also has a positive effect on latency. With the help of separating message ingestion from message processing, the system can maintain stable operation even during spikes in incoming traffic. In chat-moderation architectures, a queue buffers incoming messages while parallel processing units read and analyze them as they become ready. This approach supports steady throughput by smoothing out short-term load anomalies. A similar pattern is used in information-retrieval systems: new documents or logs are indexed asynchronously through an update queue.

Typical queuing infrastructure also has mechanisms designed into it to deal with failures in processing messages. Such messages that cannot be processed by a consumer may be automatically sent to a dead-letter queue, which can be reviewed at some convenient time. For cases of transient messages, redeliveries may be applied, and messages can be made flexible and reliable regardless of instability (fig. 5).

Predictive mechanisms are often employed to further reduce latency. For example, Pulsar automatically rebalances partitions within the cluster when imbalance is detected through BookKeeper and ZooKeeper, thereby minimizing the risk of localized overloads.

It is also important to consider the application context of moderation and real-time search. In moderation systems, such as live-stream monitoring or chat analysis, any delay directly translates into a longer exposure time of inappropriate content to users. For this reason, such tasks typically rely on strict SLA and aim for minimal

sliding latency. Real-time search systems, in turn, require continuous index updates as new documents arrive [10]. Delays in data ingestion have a direct impact on the freshness and relevance of search results.

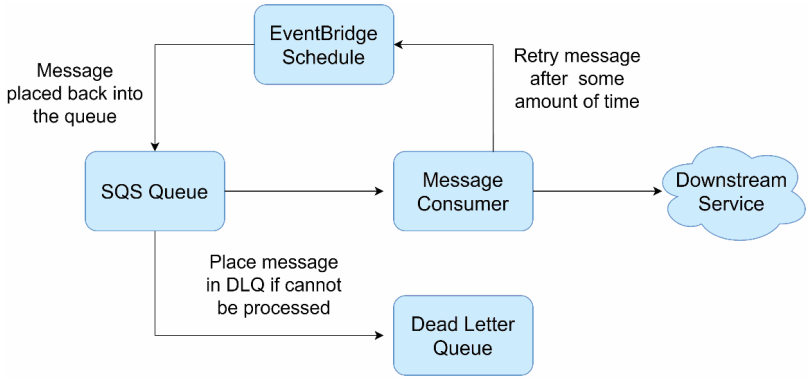

Fig. 5. SQS-Based message retry and Dead-Letter handling architecture [9]

In both cases, the architectural and algorithmic techniques discussed (load balancing, adaptive scaling, buffering, and prioritization) contribute to improved stability and responsiveness of moderation pipelines and index-update mechanisms, ensuring higher effectiveness and timely system reactions.

Conclusion

Distributed message queues provide significant help in making streaming systems resilient and scalable under high load conditions. The architectural aspects of distributed queues such as topic partitioning, data replication, and decoupling help in dealing with variable loads and ensuring reliable messaging. The concept of decoupling processing and storage in platforms like Apache Pulsar, as well as those designed in terms of massive connections and based on Kafka, showcases robust throughput and excellent fault tolerance. The static and dynamic load balancers designed around topic partitioning provide initial defense against overload conditions, which form the basis of scalability.

In modern scenarios that require low latency and consistent performance, such as real-time content moderation, adaptive and intelligent optimization methods become especially important. The use of predictive models, deep learning techniques, and flexible routing mechanisms enables dynamic redistribution of traffic, reduction of peak latencies, and maintenance of system operation within defined SLA. A combination of hardware redundancy, software elasticity, and algorithmic adaptability provides the most effective balance between performance, fault tolerance, and operational cost in real-world conditions.

References / Список литературы

1. Garifullin R. The use of modern web technologies for ensuring compatibility and interoperability in the context of medical information platforms // International Journal of Humanities and Natural Sciences. 2025, vol. 1-3(100), pp. 99-103.
2. Thallapally N. Enhancing distributed systems with message queues: architecture, benefits, and best practices // Journal of Electrical Systems. 2025, vol. 21. no. 1s, pp. 63-69.

3. Daghistani A., Khayat M., Felemban M., Aref W.G., Ghafoor A. Guard: attack-resilient adaptive load balancing in distributed streaming systems // IEEE Transactions on Dependable and Secure Computing. 2021, vol. 19, no. 6, pp. 4172-4186.
4. Daghistani A., Aref W.G., Ghafoor A., Mahmood A.R. Swarm: adaptive load balancing in distributed streaming systems for big spatial data // ACM Transactions on Spatial Algorithms and Systems. 2021, vol. 7, no. 3, pp. 1-43.
5. Smirnov A. Approaches to building fault-tolerant distributed systems: the CAP theorem and its practical application // International Journal of Engineering and Computer Science. 2025, vol. 14, no. 07, pp. 27488-27491.
6. Xie Z., Ji C., Xu L., Xia M., Cao H. Towards an optimized distributed message queue system for AIoT edge computing: a reinforcement learning approach // Sensors (Basel). 2023, vol. 23, no. 12, p. 5447.
7. Desai P. Ensuring exactly-once semantics in Kafka streaming systems // Journal of Computer Science and Technology Studies. 2025, vol. 7, no. 9, pp. 423-432.
8. Arafat J., Tasmin F., Poudel S., Tareq A.H. Next-generation event-driven architectures: performance, scalability, and intelligent orchestration across messaging frameworks // arXiv preprint arXiv:2510.04404. 2025.
9. Create a serverless custom retry mechanism for stateless queue consumers / AWS Amazon [Electronic resource]. – Access mode: https://aws.amazon.com/blogs/architecture/create-a-serverless-custom-retry-mechanism-for-stateless-queue-consumers/.
10. Drogunova Y. The impact of test automation on the effectiveness of business digital transformation // Cold Science. 2025, no. 16, pp. 53-60.

Богуцкий Александр Дмитриевич – разработчик ПО	**Bogutskii Aleksandr Dmitrievich** – software developer
alexander.bogutskij@yandex.ru	

Received 06.02.2026