StreamRAG: A Lock-Aware and Traffic-Aware Query Coordinator in Stream-Based RAG Systems

Yeonwoo Jeong[1], Kyuli Park[1], Sungyong Park[1†]
[1]*Sogang University*, Seoul, Republic of Korea
{akssus12, kyuripark, parksy}@sogang.ac.kr

***Abstract*—Stream-based retrieval augmented generation (RAG) systems integrate stream processing engines (SPEs) with real-time document retrieval to support dynamic indexing and search over unstructured datasets. However, efficiently executing queries in these systems is challenging, as seamless coordination between SPEs and vector databases is essential for maintaining low latency and high throughput. The lack of mutual awareness between these components results in two major performance bottlenecks. First, SPEs are unaware of ongoing indexing operations in vector databases, leading to metadata lock contention when indexing and search operations overlap, which increases query latency. Second, vector databases lack visibility into query traffic from SPEs and rely solely on internal metrics for scaling. As a result, they respond reactively to traffic spikes, often leading to instance overload and delayed query processing. To address these issues, we propose StreamRAG, a lock-aware and traffic-aware query coordination mechanism that facilitates real-time exchange of metadata lock statuses and query traffic metrics between SPEs and vector databases. By optimizing query routing and enabling proactive instance scaling, StreamRAG enhances the performance and scalability of real-time RAG systems. Experimental results demonstrate that StreamRAG reduces tail latency by up to 4× at the 99th percentile and significantly improves overall system performance under varying traffic conditions.**

***Index Terms*—RAG System, Stream Processing Engine, Query Coordinator, Burst Traffic**

I. Introduction

To improve accuracy in scenarios where large language models (LLMs) generate hallucinated answers [1] due to missing training data, retrieval augmented generation (RAG) systems have emerged [2]. RAG enhances original queries by incorporating relevant information from domain-specific knowledge bases. To achieve this, RAG systems convert raw data (e.g., documents, images, corpora) into multidimensional vector embeddings offline and store them in vector databases.

In today's digital landscape, new information is continuously generated from various sources, such as online retail markets, web browsers, and chatbot services. As a result, users increasingly demand fresh content and more relevant results. For instance, Amazon leverages RAG to analyze recent purchase patterns and deliver product recommendations [3].

However, indexing newly generated data in real-time while performing vector searches presents two key challenges. First, much of the newly generated data is unstructured, making it difficult to extract meaningful features. Preprocessing tasks, such as tokenization and metadata enrichment, are required to create vector embeddings but can be time-consuming. Second, RAG systems must support scalable vector search to effectively handle continuous and high query loads.

To address these challenges, stream processing engines (SPEs) [4]–[7] have recently been integrated into RAG systems [8]–[10], resulting in stream-based RAG systems. In these systems, SPEs convert in-flight unstructured data into high-dimensional vector embeddings, which are stored in a vector database. The SPEs then query the database to perform vector searches. Plugin-based connectors [11], [12] allow users to easily integrate real-time indexing and vector search with minimal code changes.

However, the weak coupling between SPEs and vector databases causes performance bottlenecks due to limited visibility between the two engines. Our preliminary studies identify two key issues where this lack of coordination negatively impacts the performance of the RAG system:

Metadata lock contention. SPEs are unaware of real-time indexing operations in the vector database when issuing vector search requests. For example, while metadata is being updated during document indexing, vector searches also rely on metadata for retrieval. If a search is made while the metadata file is locked, the request is delayed, which increases tail latency.

Reactive scaling for burst traffic. The vector database lacks visibility into query traffic from SPEs. While it can scale horizontally based on internal metrics, this scaling is reactive. This often leads to traffic bursts being directed to a single instance, resulting in overloaded instances and increased search latency.

In this paper, we propose StreamRAG, a lock-aware and traffic-aware query coordination mechanism designed to address these challenges. The core idea is to deploy an agent process on each node where the engines run, allowing them to seamlessly exchange metadata lock statuses and traffic information. When the vector database updates its index (e.g., adding new vectors or reorganizing data), the database agent launches an additional instance with a replica of the index and communicates this to the SPEs. The SPEs then dynamically route queries to the new instance, preventing delays due to metadata lock contention.

Furthermore, StreamRAG monitors real-time query traffic, including both current and past query volumes. This data is sent to the database agent, which detects traffic bursts using a smoothed Z-score algorithm [13] and preemptively launches

†S. Park is the corresponding author.

2993-2114/25/$31.00 ©2025 IEEE
DOI 10.1109/CCGRID64434.2025.00065

additional instances. Once this information is communicated back to the SPEs, they distribute queries to the new instances, ensuring efficient handling of burst traffic.

We implemented STREAMRAG on Apache Spark [4] and ChromaDB [14], and developed an agent layer to facilitate communication between the two engines using gRPC [15]. To evaluate its effectiveness, we created three traffic scenarios with varying query arrival and indexing rates. In all scenarios, STREAMRAG significantly improved overall performance, reducing tail latency by up to 4× at the 99th percentile.

In summary, this paper makes the following key contributions:

- We demonstrate that query routing without effective coordination between the SPE and the vector database results in suboptimal performance.
- STREAMRAG introduces lock-aware and traffic-aware query coordination modules, which reduce vector search latency in real-time RAG systems.
- We implemented a working system on Apache Spark and demonstrated its effectiveness in handling burst traffic scenarios.

II. BACKGROUND AND RELATED WORK

This section provides an overview of vector databases and stream-based RAG systems, along with a review of related works relevant to STREAMRAG.

A. Vector Database

Recently, RAG systems have incorporated vector databases [14], [16], [17] to enable efficient retrieval. Before the retrieval process, the vector database prepares vectors through several steps. First, domain-specific documents are converted into multi-dimensional vectors using embedding models (e.g., BERT [18], GPT [19], LLaMA [20]). These models encode textual or multi-modal content into dense vector representations that capture the semantic meaning of the documents. The vectors are then stored alongside metadata such as document IDs, timestamps, and labels, allowing for context-aware searches. The metadata is dynamically updated to reflect changes such as vector insertions, deletions, and modifications, ensuring search consistency and accuracy.

As the metadata and the number of vectors grow, vector search can become computationally intensive. To improve search efficiency, modern vector databases utilize various indexing algorithms, including quantization-based methods (i.e., IVFPQ [21]), graph-based methods (i.e., HNSW [22], SCANN [23], VStore [24]), and tree-based methods (i.e., rpForest [25]).

Once indexing is complete, user queries are transformed into multi-dimensional vectors and sent to the vector database for similarity search. Given a query vector, the search process identifies the nearest vectors in the index based on distance functions such as cosine similarity or Euclidean distance. The retrieved vectors, along with their associated metadata are then returned to the users.

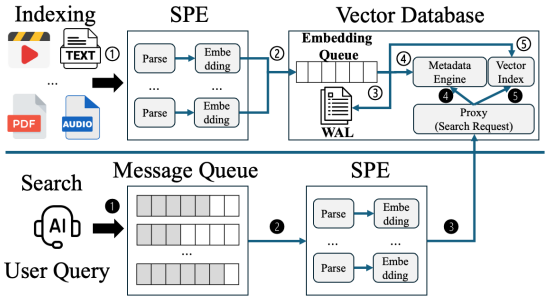


Fig. 1: Overview of a stream-based RAG system.

B. Stream-Based RAG System

A stream-based RAG system is an advanced architecture that integrates SPEs with RAG models to enable real-time data processing.

In a stream-based RAG system, newly arriving data must be indexed while real-time vector searches are performed, ensuring that the latest documents are always available for retrieval and minimizing outdated responses. In real-time scenarios, the volume of user queries can fluctuate significantly, with the system occasionally handling thousands of queries in a short time. Moreover, newly generated documents often contain irrelevant content, such as markdown tags and meta characters, which complicates their conversion into a structured format suitable for embedding vector generation. To address these challenges, SPEs are being integrated into real-time RAG systems [8]–[10]. These SPE engines provide scalable processing by parallelizing tasks like documents parsing and handling vector search requests.

Figure 1 illustrates the workflow of a stream-based RAG system. In the indexing phase, documents from various data sources are sent to the SPE ①. These documents are parsed into a structured format and then transformed into embedding vectors through parallel processing. The SPE then sends the generated embedding vectors to the vector database ②. Upon arrival, the embedding vectors are temporarily stored in the embedding queue. They are periodically saved to a write-ahead log (WAL) file to ensure the safe storage of the vector insertions ③. Afterward, the vector database updates the embedded metadata engine ④ and rebuilds the index structure to incorporate the newly inserted vectors ⑤.

When documents are prepared in the vector database, the RAG system can retrieve relevant contents in response to user queries. As users submit multiple queries (i.e., questions, images, device information), these queries are buffered in message queues ❶. The SPE periodically retrieves queries from the message queues and generates embedding vectors ❷. The query vectors are then sent to vector database ❸. Before traversing the index structure, the vector database first filters metadata from each query to extract search conditions, such as timestamps and the number of results for the top-k search ❹. Using this filtered information, the database retrieves vectors closest to each query vector ❺, continuing the search until all query vectors have been processed.

C. Related Work

Vector databases have become essential for managing and querying high-dimensional data, with various systems offering tailored solutions to meet performance and scalability demands.

Milvus [16] introduces a high-performance vector database designed to efficiently manage large-scale, high-dimensional vector data. It offers a unified platform for handling diverse vector data types and supports multiple indexing methods. Manu [26] presents a cloud-native vector database that combines log-based architecture, multi-version concurrency control, and tunable consistency to handle dynamic workloads. SPFresch [27] focuses on efficient in-place updates for large-scale vector search indices, implementing a lightweight rebalancing protocol for partition splits and vector reassignment. Recent vector databases are increasingly built on serverless platforms. Mosaic [28], for example, provides a scalable solution for embedding storage and retrieval within existing data intelligence platforms [10]. It supports real-time vector search through a serverless architecture and offers auto-sync capabilities to accommodate underlying data updates.

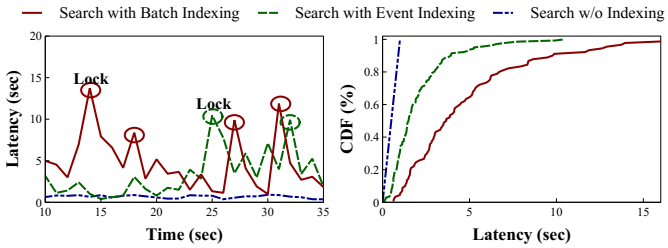
While these systems excel in performance and scalability, they have notable limitations, especially in integrating with stream processing engines. In real-time scenarios, concurrent document insertion and query searches compete for shared resources, leading to lock contention. Additionally, vector databases struggle to detect high query volumes during continuous data ingestion, preventing them from adapting proactively to burst traffic. As a result, performance degrades under such conditions. To address these challenges, we propose a lock-aware and traffic-aware query coordinator to improve system efficiency and performance.

III. Motivation

The lack of coordination between the SPE and the vector database can lead to significant performance degradation in query execution. In our preliminary studies, we identified key issues that arise when these systems operate without awareness of each other's state. For our experiment, we utilized the Wikipedia dataset [29] along with a synthetic traffic distribution. We chose Apache Spark [4] as the SPE, ChromaDB [14] as the vector database, and Apache Kafka [30] as the message queue framework. Our findings highlighted two primary issues that caused high tail latency during system interactions. A detailed analysis of these problems is presented below.

Problem 1. Metadata lock contention. In a stream-based RAG system, document indexing and vector search occur simultaneously. Before these operations, the vector database retrieves embedded metadata, which includes auxiliary information such as document IDs, labels and tags associated with the embedding vectors. During the vector search, the database reduces the search space by applying distance functions along with additional filters, such as document categories or timestamps, based on metadata lookups.

However, accessing metadata introduces a complex locking process to maintain data consistency, placing a significant burden on users to manage these operations manually. To simplify metadata management, modern vector databases incorporate embedded storage engines such as SQLite [31] and PostgreSQL [32], which allow SQL-based manipulation of metadata for scalable and efficient query execution. These engines ensure data integrity through database-level locking mechanisms. However, this locking process introduces latency, as transactions that perform metadata lookups must wait for concurrent metadata modification transactions to complete.



(a) Time-series search latency (b) Tail latency

Fig. 2: Search latency and the cumulative distribution function (CDF) of search tail latency across three scenarios involving interactions between the SPE and vector database.

We conducted a synthetic experiment to evaluate vector search performance in a stream-based RAG system where coarse-grained lock states are not shared between the SPE and vector database. To simulate query arrival and insert traffic, we defined three scenarios: search without indexing, search with event indexing, and search with batch indexing. In the search without indexing scenario, no indexing occurs when the SPE requests a vector search. In contrast, in the search with indexing scenarios, vector searches overlap with the indexing phase. We generated synthetic insert traffic as described in Section VI-A. Simultaneously, our query generator sends 100 Wikipedia queries per second (QPS) to the message queue. The SPE retrieves accumulated queries from the message queue, distributes them across multiple threads, and converts them into embedding vectors. It then sends similarity search requests to a vector database instance pre-indexed with 10,000 Wikipedia documents. The top-k similarity search is configured with a k value of 5.

Figure 2 (a) shows the time-series search latency. The search without indexing maintains stable latency, while both indexing scenarios exhibit significant latency fluctuations. In these cases, latency spikes occur, with batch indexing showing higher spikes compared to event-based insertion. This is due to metadata access being locked during high-volume indexing, causing vector searches to be temporarily queued. As shown in Figure 2 (b), the 99th percentile tail latency for searches with batch indexing is around 16.3 seconds, approximately 1.64 times higher than the latency observed with event indexing at the 99th percentile. This suggests that issuing vector searches from the SPEs without recognizing metadata lock contention during indexing in the vector database leads to high search tail latency.

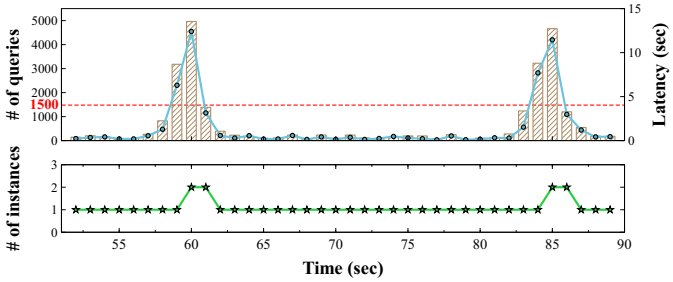


Fig. 3: The relationship between # of queries, search latency, and # of instances. An additional instance is launched when the query count exceeds the threshold of 1500. In the graph above, the bar graph represents the number of queries, while the line graph represents search latency.

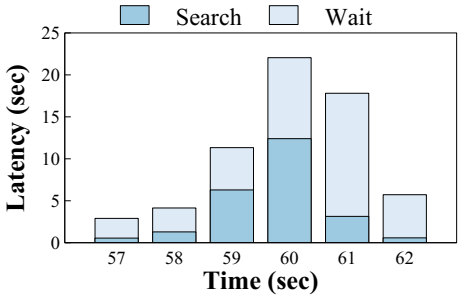


Fig. 4: Breakdown of total latency during peak load periods (57-62). Total latency includes both search latency and wait latency.

Problem 2. Reactive scaling for burst traffic. The query arrival rate fluctuates based on user demand [33], [34], causing the SPE to send varying numbers of query requests to the vector database. As expected, search latency increases as the query volume rises. To address this, modern vector databases implement scaling policies [16], [35]. These databases scale out by launching additional instances that replicate metadata, indexes, and associated data from the original instance. Incoming queries are then distributed across all available instances.

However, instance scaling is typically reactive, triggered by predefined conditions such as query quota limits or CPU utilization thresholds. Due to the lack of coordination between the SPE and the vector database, the database cannot detect the actual number of incoming queries in real time. As a result, during sudden traffic surges, the system experiences increased search latency before scaling mechanisms can take effect.

We conducted a synthetic experiment to evaluate query latency without proactive scaling. Using the synthetic traffic from Section VI-A, we analyzed query performance under varying loads and implemented a scale-out policy triggered when processed queries exceed a threshold.

Figure 3 illustrates how the number of instances, search latency, and query volume evolve over time based on the implemented scaling policy. We identify two distinct periods with significant latency spikes. In both cases, the system maintained only one instance, even when the query volume surpassed the predefined threshold (e.g., 1500 queries). For instance, at 59 seconds, instance scaling was not triggered despite exceeding the threshold, causing search latency to spike to 6.29 seconds as a single vector database instance handled all incoming queries.

To analyze the impact of the lack of proactive scaling, we focus on the period from 57 to 62 seconds, as shown in Figure 4. As query processing slowed, the backlog of waiting queries in the message queue grew rapidly. The increasing latency further delayed queries in the message queue, leading the SPE to issue even more search requests. Consequently, search latency peaked at 12.4 seconds, which is 5.27 times higher than the latency under idle load conditions.

Our Observations. The lack of coordination between the SPE and the vector database, which operate independently without sharing internal state, leads to two key performance issues in query search. First, issuing query search requests without awareness of metadata lock contention during document indexing results in high tail latency. Second, reactive scaling of vector database instances, which fails to account for incoming query volumes, causes latency spikes during burst traffic.

To overcome these challenges, a lock-aware and traffic-aware query coordinator is essential to mitigate performance bottlenecks and improve query efficiency.

IV. Design

A. Overall Architecture

StreamRAG is an adaptive query routing strategy implemented at the SPE for vector search in stream-based RAG systems. StreamRAG has two modules: (i) **Lock-aware** query routing module and (ii) **Traffic-aware** proactive database instance provision module. Our mechanism centers on two key designs:

- The lock acquisition status in the vector database can be predicted in advance, helping to prevent high tail latency when the SPE issues vector search requests.
- The SPE is aware of offset information, which indicates the number of queries to fetch before retrieval, allowing the system to proactively provision a vector database instance.

Figure 5 illustrates the architecture of StreamRAG. To enable communication between the SPE and vector database, we have designed an agent layer that interacts with both systems. The agent layer consists of two agents: the lock agent and the traffic agent. These two agents operate concurrently across the two layers without interfering with the system's operations. Additionally, the agent layer in the vector database includes a provision manager, responsible for scaling instances and notifying the DB agent of any changes in instance information. The detailed communication steps for each agent are described in Sections IV-B and IV-C.

Retrieving information for each engine in the SPE agent and DB agent layers often delays the vector search at runtime, reducing overall performance. To alleviate this issue, we

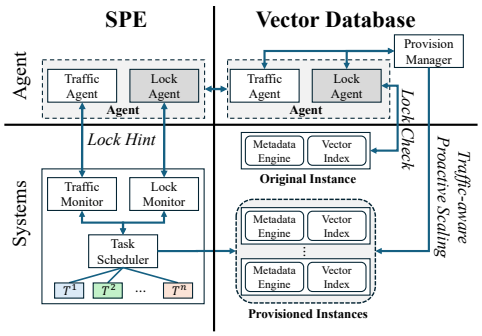


Fig. 5: An overview of STREAMRAG.

employ lazy evaluation [36], a strategy widely adopted in SPEs [4], [5]. Lazy evaluation defers computations until their results are required. In Apache Spark [4], for instance, query execution occurs only when specific functions (e.g., *trigger*, *count*) are invoked. To execute the query with lazy evaluation, the SPE constructs a query execution plan that defines the sequence of tasks [37]. Once established, the SPE executes all functions sequentially according to the query plan. Exploiting this property, we pipeline the exchange process in the agent layer with task execution, ensuring that the task of fetching queries from the message queue is completed before the vector search is performed.

B. Lock-aware Query Routing Module

Figure 6 illustrates a workflow of the lock-aware query routing module. The SPE agent begins when the SPE initializes a query and make query execution plan. At this time, the SPE sets necessary metadata to access vector database such as database IP address, port number, and collection name where the vector index is stored. Then, the SPE forwards the metadata to the SPE lock agent. Finally, the SPE lock agent forwards the metadata to the DB lock agent.

When lazy evaluation is triggered, the SPE starts to fetch queries from the message queue while the DB lock agent checks the metadata lock status for the collection. If the lock is held, the provision manager in DB layer replicates the original instance and returns information of the new instance including port number per provisioned instance. Occasionally, task which fetches the queries from the message queue may complete before it receives the updated instance information from the DB lock agent, potentially causing the search to run with outdated information.

To prevent this, we integrate the *await* function provided in Future interface [38], which ensures that updated instance information is retrieved prior to the query search. Before performing a vector search, the SPE checks the lock status updated by the SPE lock agent. Finally, the SPE routes the queries to the provisioned instances instead of original instance. To optimize resource utilization, the module terminates the provisioned instances when the metadata file lock is released during subsequent vector searches.

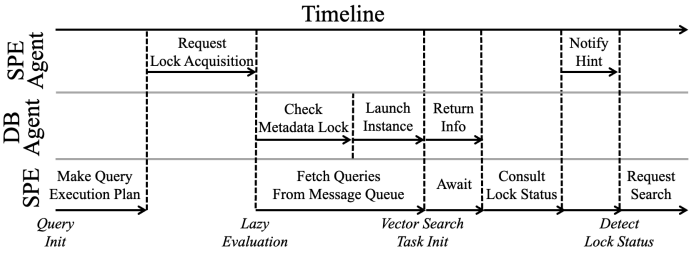


Fig. 6: A workflow of lock-aware query routing module.

C. Traffic-aware Proactive Instance Provision Module

During burst traffic, an effective solution is to pre-provision additional instances and route queries to these pre-launched instances. To detect burst traffic, we use a smoothed Z-score algorithm [13].

This algorithm utilizes a sliding window to analyze a dataset that contains historical and current data, smoothing out short-term fluctuations. It calculates a moving average at time t, denoted as M_t, using a window size w and the datasets within the window x. Based on M_t, the standard deviation σ_t is calculated. Both equations are presented as Equation 1 and Equation 2.

$$M_t = \frac{1}{w} \sum_{i=t-w+1}^{t} x_i \tag{1}$$

$$\sigma_t = \sqrt{\frac{1}{w} \sum_{i=t-w+1}^{t} (x_i - M_t)^2} \tag{2}$$

Using M_t and σ_t, the algorithm calculates Z-score Z_t as shown in Equation 3. Z_t measures the deviation of the current data point from the moving average, expressed in terms of standard deviations. We compare Z_t to a predefined threshold α, and if Z_t exceeds this threshold, burst traffic is detected.

$$Z_t = \frac{x_t - M_t}{\sigma_t} \tag{3}$$

While detecting abnormal traffic is essential, determining the appropriate number of additional instances to launch is equally important. The SPE fetches queries from the message queue immediately after completing previous vector search. If the vector search shows a stable search latency, it indicates that the current instance has sufficient processing capacity. Based on this fact, we define the maximum processing capacity of a single vector database instance as certain number of queries. A lower threshold reflects higher sensitivity to search latency. The parameter β controls this sensitivity and is configurable by the user. A detailed analysis of β's impact on search latency is presented in Section VI-E.

As explained in Section IV-B, the SPE constructs a query execution plan and configures several metadata to facilitate task executions. For example, the SPE sets the metadata including offset information that records the last-read position in the message queue. By utilizing the offset information, the SPE fetches messages accumulated since the last-read

position to determine the current query volume. Based on the current query volume, DB provision manager calculates the required number of additional instances, N_t^{new}, by dividing the number of incoming queries at time t, Q_t^{cur}, by the predefined processing capacity per instance, β. The number of additional instances, N_t^{add}, is then obtained by subtracting the number of current active instances, N_t^{cur}, from N_t^{new}. The corresponding equations are presented in Equation 4.

$$N_{\text{new}} = \frac{Q_t^{\text{current}}}{\beta}, \quad N_{\text{add}} = \max(0, \lceil N_t^{\text{new}} - N_t^{\text{cur}} \rceil) \tag{4}$$

Algorithm 1 presents a pseudo-code that outlines the workflow for proactive traffic-aware scaling of vector database instances. Before performing vector search, the SPE checks whether new instances have been provisioned. At this stage, the SPE collects traffic information including the number of incoming queries and historical query volume. Then, the SPE sends traffic information to SPE traffic agent. Hereafter, the SPE traffic agent forwards this information to the DB traffic agent. Upon receiving the data, the DB traffic agent calculates Z-score and sends the computed value along with the current query volume to the DB provision manager. The provision manager then evaluates whether the current query volume indicates burst traffic based on pre-computed Z-score. If burst traffic is detected, the DB provision manager determines the number of additional instances using Equation 4 and launches new instances. Information regarding provisioned instances is forwarded to the DB traffic agent. Also, the DB traffic agent sends the information to the SPE traffic agent. Finally, the SPE traffic agent receives the information about the newly provisioned instances. Based on the knowledge, the SPE distributes vector search requests across all active instances. This seamless exchange of information between both systems reduces search latency by preventing a single vector database instance from becoming a bottleneck, instead leveraging multiple instances for parallel processing.

V. Implementation

StreamRAG introduces a cross-layer design to facilitate information sharing between the SPE and the vector database, enhancing vector search performance in real-time RAG systems. We implemented a prototype of StreamRAG using Apache Spark [4] v3.5.1 and ChromaDB [14] v0.5.20. To exchange information between both systems, we used gRPC [15] to enable communication between them. The agent on node hosting the SPE, co-located with Spark executor responsible for processing tasks, operates as a gRPC client, while the DB agent functions as a gRPC server. To compute the Z-score for values within a sliding window, including the mean and standard deviation, we leveraged the Z-score library provided by SciPy [39]. We integrated lock-aware and traffic-aware query coordination into User-defined Function (UDF) of Spark. The hint and user-defined parameters are incorporated by the UDF to optimize query routing.

Algorithm 1: Traffic-aware proactive database instance provision algorithm

Input: Traffic Information: Q_t^{cur}, $\{x_{t-w+1}, \cdots, x_t\}$
Function `SPEVectorSearch` (*Traffic Information*) **:**
  // Send traffic information
  $instance \leftarrow$ **SPETrafficAgent**(Q_t^{cur}, $\{x_{t-w+1}, \cdots, x_t\}$)
  if $instance_{count} \geq 2$ **then**
    Route $\lceil \frac{Q_t^{cur}}{instance_{count}} \rceil$ to each instance;
  else
    Route Q_t^{cur} to single instance;

Input: Traffic Information: Q_t^{cur}, $\{x_{t-w+1}, \cdots, x_t\}$
Function `SPETrafficAgent` (*Traffic Information*) **:**
  $instance \leftarrow$ **DBTrafficAgent**(Q_t^{cur}, $\{x_{t-w+1}, \cdots, x_t\}$)
  Return $instance$;

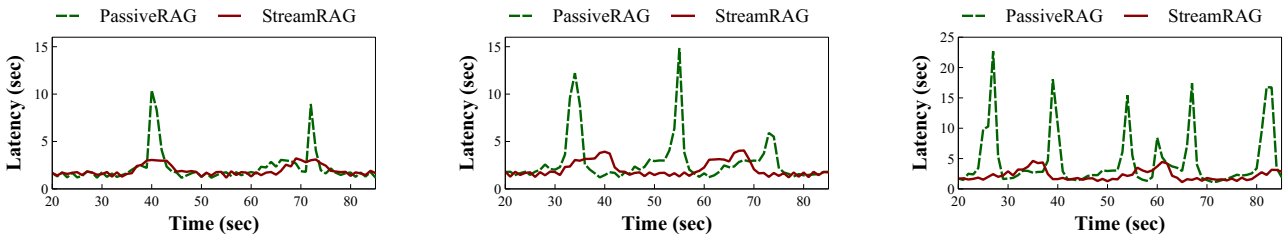
Input: Traffic Information: Q_t^{cur}, $\{x_{t-w+1}, \cdots, x_t\}$
Function `DBTrafficAgent` (*Traffic Information*) **:**
  // Calculate values for Z-score
  $M_t \leftarrow \frac{1}{w} \sum_{i=t-w+1}^{t} x_i$
  $\sigma_t \leftarrow \sqrt{\frac{1}{w} \sum_{i=t-w+1}^{t} (x_i - M_t)^2}$
  $Z_t \leftarrow \frac{x_t - M_t}{\sigma_t}$
  // Send Z-score and current query volume
  $instance \leftarrow$ **DBProvisionManager**(Z_t, Q_t^{cur})
  Return $instance$;

Input: Z-score: Z_t, Current Traffic: Q_t^{cur}
Function `DBProvisionManager` (Z_t, Q_t^{cur}) **:**
  // Detect burst traffic
  if $Z_t \geq \alpha$ **then**
    $\text{N}_{\text{new}} \leftarrow \frac{Q_t^{cur}}{\beta}$
    $\text{N}_{\text{add}} \leftarrow \max(0, \lceil N_t^{\text{new}} - N_t^{\text{cur}} \rceil)$
    // Provision N_{add} instances
    Return new instance information;
  else
    Return $\emptyset$;

VI. Evaluation

A. Experimental Setup

Configuration. We conducted a series of experiments using Spark cluster consisting of one master node and two worker nodes. Each worker node ran a Spark executor with 8 CPU cores and 48GB of memory, using 8 data partitions to enable parallel processing. We used ChromaDB [14] as vector database. In vector database configurations, we chose index algorithm as HNSW [22], which is a state-of-the-art algorithm for approximate nearest neighbor search. Additionally, we set the number of top-k neighbors for each user query to 5. The vector database instance is running on master node and



(a) Low traffic (b) Medium traffic (c) High traffic

Fig. 7: Time-series search latency in three traffic scenarios.

additional replicated instances are provisioned on master node when the query burst is detected. To detect burst traffic, we used a Z-score threshold of 0.5 and set β to 800 to determine the number of vector database instances to launch. We used Apache Kafka 3.2.3 [30] as the message queue and Apache Spark [4] 3.2.3 as the stream processing engine. Default values were used for detailed hardware specifications can be found in Table I. All experiments include a one-minute warm-up phase to ensure the stabilization of system components such as caches and buffers.

TABLE I: Testbed specification.

CPU	AMD Ryzen 9 3900X 12-core 2.80 GHz
Memory	DDR4, 64 GB
Storage	Samsung SSD 970 EVO NVMe SSD
Ethernet	1 Gbps

Workloads. For representing an online scenario with document indexing and vector search, we implemented two applications that store and search embedding vectors on the vector database, respectively. One is to extract wikipedia [29] summaries from narrativeQA [40] dataset and transform the wikipedia document to embedding vectors using pre-trained embedding model (i.e., all-MiniLM-L6-v2 [41]). The other is to transform questions from narrativeQA to embedding vectors and requests vector search.

Traffic. To simulate a burst query traffic, we made query generator to follow Poisson distribution. In addition, document indexing patterns can vary depending on application requirements, document indexing patterns can vary. We considered two representative indexing types: event indexing and batch indexing. Those types are reflecting fluctuated traffic distribution. Query and insert traffic types are described as below.

- Query traffic: In the Poisson distribution, we set the query arrival rate (λ_{prob}) to ensure that the average query arrival rate remains within 10,000 messages per minute [42]. Specifically, a fraction of queries, determined by λ_{prob}, arrives within the first 10 seconds, while the rest are distributed over the remaining time. This configuration reproduces query bursts at specific points in time.
- Batch insert: the generator inserts 5,000 documents at once every 30 seconds.
- Event insert: the generator inserts 5,000 documents randomly within 30 seconds.

Based on the traffic patterns of query and insert, we designed three traffic scenarios that mix query traffic and insert traffic. Three scenarios are provided in Table II.

TABLE II: Traffic scenarios.

Low	event insert, λ_{prob}=1000
Medium	batch insert, λ_{prob}=3000
High	batch insert, λ_{prob}=5000

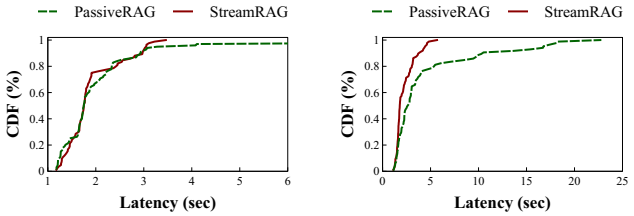
Comparison Targets. We evaluate vector search latency of STREAMRAG under conditions where queries are ingested in bursts and documents are indexed concurrently. In Spark, our selected SPE, queries are processed in batches rather than individually. To align with the SPE, we measured vector search latency as the time from batch submission to completion.

Throughout the experiment, we refer to vector search latency as search latency. The baseline system routes the queries without considering metadata lock contention. Moreover, it reactively adds an additional instance when the number of current queries exceeds data size threshold. We call the baseline system as PASSIVERAG that performs vector search without considering metadata lock contention and fluctuated query traffic in RAG system.

B. Search Latency

Figure 7 presents the time-series search latency across three traffic scenarios. Overall, our proposed mechanism outperforms PASSIVERAG in all cases. In Figure 7 (a), two peaks in search latency are observed, with latency reaching up to 10 seconds from 39 to 42 seconds. Notably, STREAMRAG maintains a stable search latency below 3 seconds. As traffic fluctuations intensify, the performance gap between STREAMRAG and PASSIVERAG becomes more pronounced.

In Figure 7 (c), five peaks in search latency are observed. Compared to low-traffic scenario, STREAMRAG exhibits a slight increase in latency. Still, the latency at peak period is 4.52 seconds, which is 4.9 times lower than the baseline. This significant reduction during burst traffic is attributed to our core components: the lock-aware and traffic-aware modules. STREAMRAG mitigates performance degradation caused by the lack of coordination between the two engines through transparent and seamless coordination. The effectiveness of both modules is demonstrated in following sections.



(a) Low traffic (b) High traffic

Fig. 8: Comparison of tail latency (CDF) under low and high traffic scenarios.

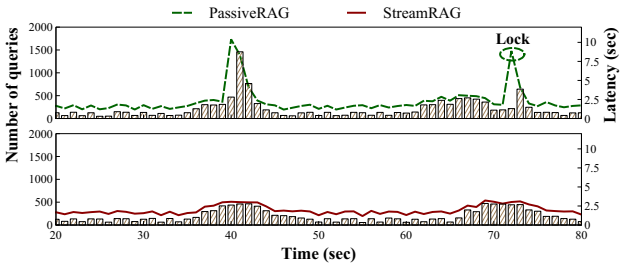


Fig. 9: Analysis of how vector search interacts with metadata locking during indexing.

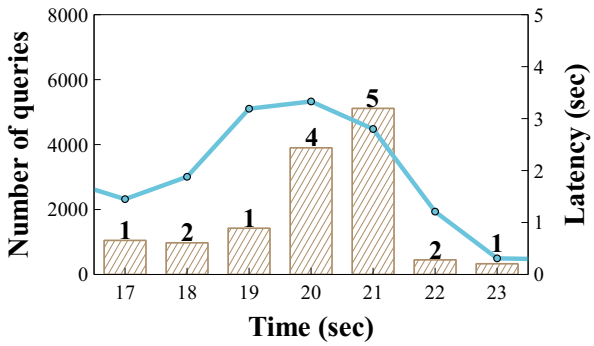


Fig. 10: Effectiveness of traffic-aware instance scaling in handling fluctuating query volumes. Bars indicate the number of queries, line shows search latency, and numbers on bars represent the number of instances.

C. Effectiveness of Lock-aware Query Routing

Figure 8 shows tail latency in low and high traffic scenarios. Regardless of the traffic scenarios, median latency is nearly same in both systems. It indicates that idle traffic and vector search without indexing do not contribute to latency increases.

Before the 95th percentile, search latency remains similar in both systems, staying within 3.3 seconds. However, beyond the 95th percentile, tail latency sharply increases. For instance, in Figure 8 (a), the 99th percentile latency for StreamRAG is 3.3 seconds. Whereas, the 99th tail latency in PassiveRAG reaches 8.8 seconds, which is 2.6 times higher than that of StreamRAG. The disparity in tail latency becomes more pronounced as the traffic rate increases. Under the worst-case latency (e.g., 100th percentile) in a high-traffic scenario, PassiveRAG reaches 22.69 seconds, which is 4.72 times higher than StreamRAG.

Figure 9 presents time-series the number of queries and search latency in low traffic scenario. Generally, large requests lead to increased search latency due to higher computation. For example, at 41 seconds, the number of queries is almost 1400. As expected, the search latency is also increased to 10 seconds due to higher query volume. However, the search latency increases significantly despite a moderate number of incoming queries at 72 seconds. It indicates vector search is delayed until the metadata access in indexing phase is completed. As a result, the search latency is increased regardless of query volume. As we explained in Section IV-B, the SPE and vector database exchanges information regarding metadata lock status and profile of newly added database instance. Through transparent communication, the SPE becomes aware of metadata locks and reroutes queries to alternative instances. Consequently, it avoids lock contention as much as possible during vector searches and significantly reduces tail latency.

D. Effectiveness of Traffic-aware Proactive Instance Provision

Figure 10 illustrates how the number of vector database instance is adjusted in response to sudden increases in query volume. As discussed in Section III, PassiveRAG relies on reactive scaling and does not effectively respond to burst traffic. Consequently, noticeable latency spikes occur when vector database instance scaling is not triggered in a timely manner. In Figure 4, the search delay spikes to 12.4 seconds, a significant increase compared to the idle traffic scenario, where the search delay remains below 1 second.

In contrast, traffic-aware proactive scaling mechanism mitigates the latency spikes at burst traffic. From 20 to 21 seconds, queries more than 4000 are arrived. This is, on average, 3.5 times higher than the traffic generated during idle periods. StreamRAG rapidly provisions four instances within 21 seconds, effectively minimizing latency increases. This latency reduction during burst traffic results from effective coordination between underlying systems. As explained in Section IV-C, SPE in StreamRAG tracks the number of previously processed query volume. For instance, the SPE calculates the number of incoming queries using offset information from message queue in query planning phase. After combining volumes of past and current incoming queries, the SPE send them to DB agent.

The DB agent calculates the Z-score and checks whether it exceeds a predefined threshold. Upon detecting abnormal traffic (e.g., burst traffic), the DB agent determines the number of additional instances using β to launch proactively and sends their information to the SPE. As a result, the SPE in StreamRAG becomes aware of the newly added instances. Leveraging this information, the SPE can distribute burst query traffic across multiple instances, which reduces computation burden in single instance. By the way, search latency remains similar despite a lower query volume during burst traffic at 19

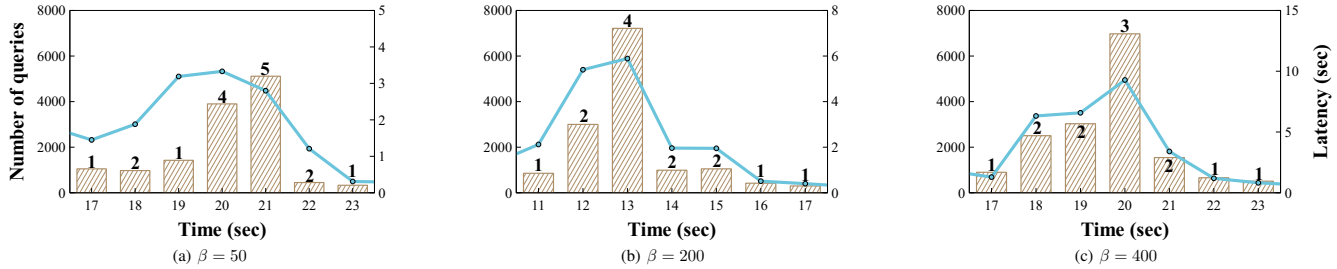


Fig. 11: Time-series search latency trend according to β. The bar graph represents the number of queries, while the line graph represents search latency. Number on each bar indicates the number of instances.

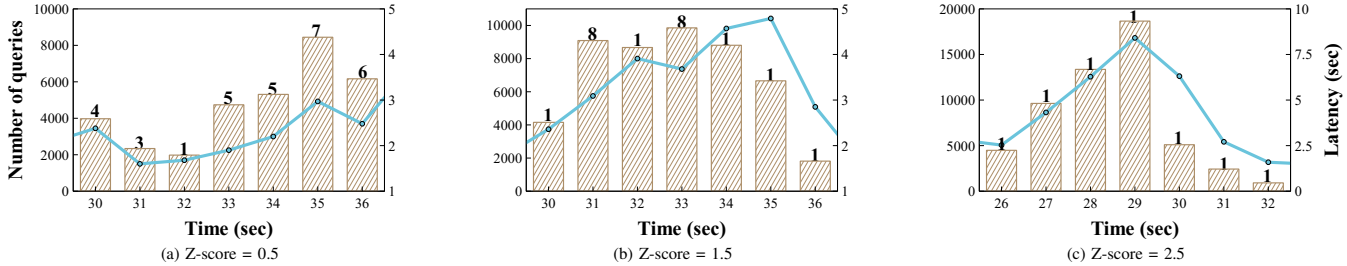


Fig. 12: Time-series search latency trend according to Z-score threshold. The bar graph represents the number of queries, while the line graph represents search latency. Number on each bar indicates the number of instances.

seconds. In mixed traffic scenarios, document indexing can be overlap with vector search. Despite significant efforts of StreamRAG to avoid lock contention, the wait for lock status updates at runtime slightly increases search latency.

E. Sensitivity of β

β is a parameter used to determine the number of additional vector database instances in proactive scaling. To examine how β affects search latency, we conducted experiments under a high-traffic scenario. From Figure 11 (a) to (c), the effectiveness of proactive scaling in reducing latency diminishes as β increases. In Figure 11 (a) and (c), the maximum search latency reaches 2 seconds, whereas in Figure 11 (b), it peaks at 9 seconds. These results suggest that a lower β enables the system to respond more effectively to burst traffic, maintaining lower search latency. However, provisioning additional instances increases memory consumption. In StreamRAG, we set an upper limit of 12 instances. The β parameter is user-defined, making it crucial to balance memory consumption and performance when determining its value.

F. Sensitivity of Z-score

To scale vector database instance proactively, we adopted Z-score to detect as burst traffic. Since the Z-score threshold is a user-defined parameter, sensitivity of scaling instance varies based on its configuration. We conducted an experiment under a high-traffic scenario to examine the trend of the number of instances and search latency across Z-score thresholds.

As shown in Figure 12 (a), when the Z-score threshold is set to 0.5 (e.g., used in our experiments), the number of instances is effectively adapted to query volume fluctuations. As the Z-score threshold increases to 1.5, the instance scaling becomes more erratic. For example, at the 32 and 34 seconds, the number of instances drops to 1, despite the query volume being more than 4 times that of idle traffic. Consequently, the search latency spikes to 3.91 and 4.57 seconds, respectively. When abnormal traffic is detected, our traffic-aware proactive instance provision module activates. However, in this case, the query volume is misclassified as normal traffic. When the Z-score threshold is set to 2.5, the number of instances remains unchanged regardless of query volume. As we discussed with Figure 4 in Section III, long search latency causes queries to accumulate in message queue, leading to a surge in query volume and further increasing search latency. A higher Z-score threshold makes detection more conservative, identifying only very extreme spikes and reducing false positives. Therefore, users should consider the traffic patterns when setting Z-score thresholds to optimize the vector database instance scaling and achieve performance gains.

VII. Conclusion

This paper identifies performance bottlenecks in stream-based RAG systems, highlighting how the lack of mutual awareness between stream processing engines and vector databases degrades performance. We propose StreamRAG, a lock-aware and traffic-aware query coordinator that optimizes communication between these systems, significantly reducing search latency. However, challenges remain, including real-time lock contention detection and efficient query routing under locking conditions. Future work will focus on dynamically tuning user parameters to enhance performance in resource-constrained environments.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00453929) (RS-2024-00416666).

REFERENCES

[1] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, 2023.

[2] F. Cuconasu, G. Trappolini, F. Siciliano, S. Filice, C. Campagnano, Y. Maarek, N. Tonellotto, and F. Silvestri, "The power of noise: Redefining retrieval for rag systems," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 719–729, 2024.

[3] B. Baruah, A. Vivekanandha, and K. Ramadoss, *Business Transformation with AI/ML*, pp. 271–337. Berkeley, CA: Apress, 2024.

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[6] M. H. Iqbal, T. R. Soomro, *et al.*, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[7] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, p. 1634–1645, Aug. 2017.

[8] D. Wu, J. Li, B. Wang, H. Zhao, S. Xue, Y. Yang, Z. Chang, R. Zhang, L. Qian, B. Wang, *et al.*, "Sparkra: A retrieval-augmented knowledge service system based on spark large language model," *arXiv preprint arXiv:2408.06574*, 2024.

[9] H. Kang, Y. Zhu, Y. Zhong, and K. Wang, "Implementing streaming algorithm and k-means clusters to rag," *arXiv preprint arXiv:2407.21300*, 2024.

[10] Databricks, "Rag on databricks." https://docs.databricks.com/aws/en/generative-ai/retrieval-augmented-generation, Jan. 2025. [Online; accessed 14-Mar-2025].

[11] J. Chen, "Introducing the databricks connector, a well-lit solution to streamline unstructured data migration and transformation." https://zilliz.com/blog/introducing-databricks-connector, Last accessed on 2024-02-08.

[12] Databricks, "Question answering over custom datasets with langchain and dolly." https://www.dbdemos.ai/minisite/llm-dolly-chatbot/03-Q&A-prompt-engineering-for-dolly.html, Last accessed on 2024-12-14.

[13] Jean-Paul, "Robust peak detection algorithm (using z-scores)." https://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data, Last accessed on 2019-11-15.

[14] Chroma-core, "Ai-native open-source embedding database." https://github.com/chroma-core/chroma, Last accessed on 2024-12-14.

[15] K. Indrasiri and D. Kuruppu, *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[16] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 2614–2627, 2021.

[17] Weaviate, "A cloud-native, open source vector database that is robust, fast, and scalable." https://github.com/weaviate/weaviate, Last accessed on 2024-12-14.

[18] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[19] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[20] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[21] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[22] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[23] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *International Conference on Machine Learning*, pp. 3887–3896, PMLR, 2020.

[24] S. Liang, Y. Wang, Z. Yuan, C. Liu, H. Li, and X. Li, "Vstore: in-storage graph based vector search accelerator," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, (New York, NY, USA), p. 997–1002, Association for Computing Machinery, 2022.

[25] D. Yan, Y. Wang, J. Wang, H. Wang, and Z. Li, "K-nearest neighbor search by random projection forests," *IEEE Transactions on Big Data*, vol. 7, no. 1, pp. 147–157, 2019.

[26] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, *et al.*, "Manu: a cloud native vector database management system," *arXiv preprint arXiv:2206.13843*, 2022.

[27] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, P. Cheng, and M. Yang, "Spfresh: Incremental in-place update for billion-scale vector search," in *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, (New York, NY, USA), p. 545–561, Association for Computing Machinery, 2023.

[28] Databricks, "Mosaic ai vector search." https://docs.databricks.com/aws/en/generative-ai/vector-search, Last accessed on 2025-3-17.

[29] W. Foundation, "Wikimedia downloads." https://huggingface.co/datasets/legacy-datasets/wikipedia, Last accessed on 2024-12-14.

[30] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, pp. 1–7, Athens, Greece, 2011.

[31] J. Kreibich, *Using SQLite*. " O'Reilly Media, Inc.", 2010.

[32] J. Worsley and J. D. Drake, *Practical PostgreSQL*. " O'Reilly Media, Inc.", 2002.

[33] M. Perron, R. Castro Fernandez, D. DeWitt, M. Cafarella, and S. Madden, "Cackle: Analytical workload cost and performance stability with elastic pools," *Proc. ACM Manag. Data*, vol. 1, Dec. 2023.

[34] Y. Chen, D. Wang, N. Wu, and Z. Xiang, "Mobility-aware edge server placement for mobile edge computing," *Computer Communications*, vol. 208, pp. 136–146, 2023.

[35] Qdrant, "High-performance, massive-scale vector database and vector search engine for the next generation of ai." https://github.com/qdrant/qdrant, Last accessed on 2024-12-14.

[36] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, p. 359–411, Sept. 1989.

[37] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), p. 1383–1394, Association for Computing Machinery, 2015.

[38] C. Hattingh, *Using Asyncio in Python: understanding Python's asynchronous programming features*. " O'Reilly Media, Inc.", 2020.

[39] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[40] T. s Koˇ ciský, J. Schwarz, P. Blunsom, C. Dyer, K. M. Hermann, G. Melis, and E. Grefenstette, "The NarrativeQA reading comprehension challenge," *Transactions of the Association for Computational Linguistics*, vol. TBD, p. TBD, 2018.

[41] all MiniLM-L6-v2, "all-minilm-l6-v2." https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, Last accessed on 2024-12-14.

[42] M. R. Hoseiny Farahabady and A. Y. Zomaya, "Geo-distributed analytical streaming architecture for iot platforms," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 263–274, 2024.