Precision to Ambiguity: Evaluating Database Systems Across the Continuum of Query Complexity (Phase 2)

Eswarankarthik Paranthaman

Anirban Bose

Navneet Sawhney

MS DSP 420- Database Systems

Professor Abid Ali

August 23, 2024

TABLE OF CONTENTS

ABSTRACT

This study evaluates the performance of various database systems for quantitative, fuzzy, and hybrid queries using appliance metadata and reviews. Comparing PostgreSQL, SQLite, DuckDB, Elasticsearch, Pinecone, Milvus, and MongoDB, the research assesses metrics including query execution time, latency, output quality, scalability, and resource utilization. Results show DuckDB excelling in quantitative queries with the fastest execution time (0.0084 seconds), Elasticsearch demonstrating lower latency (0.23 seconds) but inferior output quality compared to Pinecone in fuzzy searches, and Milvus outperforming MongoDB in hybrid queries with faster execution (0.3239 vs 0.6302 seconds) and better vector search optimization. The study recommends specific databases for different use cases: DuckDB for fast analytics, PostgreSQL for complex applications, Elasticsearch for efficient text search, Pinecone for vector-based recommendations, and Milvus for high-performance vector search. These findings aim to guide organizations in selecting appropriate database systems for their specific data management needs in an increasingly complex digital landscape.

Introduction

Data has become a key aspect of all aspects of human life. The world of digital transformation is growing at an exponential rate. According to a report by IDC, the global data sphere is expected to reach 163 zettabytes by 2025. [1] The ever-growing development of IoT devices, social media, and enterprise data has escalated and continues to intensify this growth. Data is the new oil has already become a fact. With this overabundance of data, it

[1] David Reinsel, John Gantz, and John Rydning, *Data Age 2025: The Evolution of Data to Life-Critical. Don't Focus on Big Data; Focus on the Data That's Big*, IDC White Paper, sponsored by Seagate, April 2017, https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf.

becomes imperative to use efficient systems that help handle and manage the never-ending amounts of data.

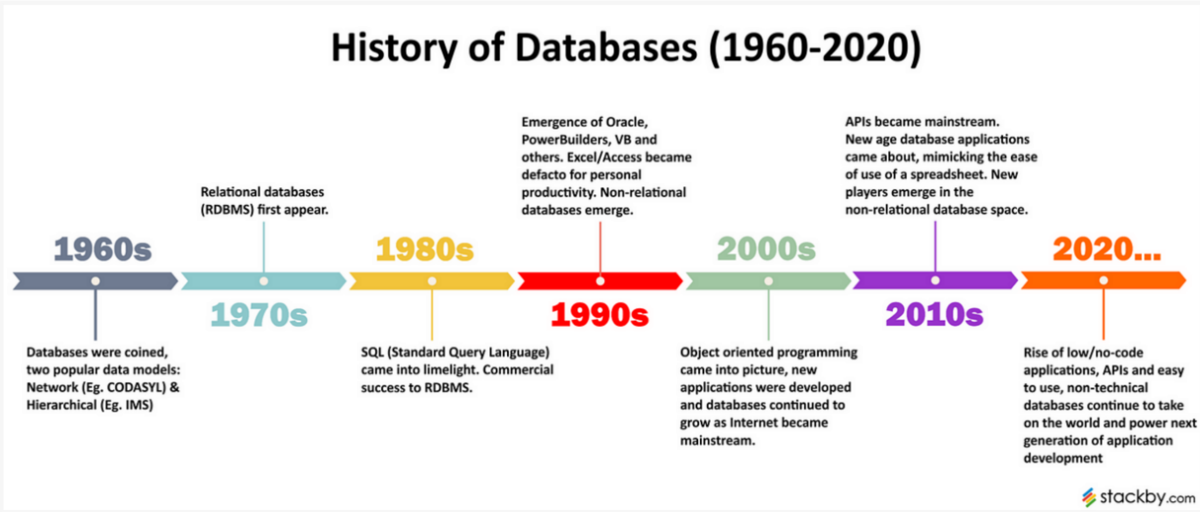

Figure 1: Evolution and History of Databases - From 1960 to 2020 & Beyond.

Source: Versha Rai, *"What is a Database - A Beginner's Guide [Updated 2024],"* Stackby, December 25, 2023, accessed August 24, 2024. https://stackby.com/blog/what-is-a-database/.

Traditional database systems (RDBMS) like PostgreSQL and SQLite have long provided the foundation for structured data storage. However, as the complexity increases, new types of databases are needed. These databases like ElasticSearch, Pinecone, Milvus and MongoDB can handle fuzzy and hybrid search queries. Today there is no dearth of databases and with each passing day a new database with advanced capabilities is introduced. Figure 1 shows a timeline of database history from 1960 to 2020, highlighting key developments like the emergence of relational databases, SQL, object-oriented programming, and modern applications.

This paper builds on the findings of Phase 1 of the database project. In the first phase, with the use of different LLMs, the suggestions for databases were gathered based on different sets of queries.[2] The analysis included quant, fuzzy and hybrid search queries. We generated

[2] Navneet Sawhney, Anirban Bose, and Eswarankarthik Paranthaman, "A Report on Comparative Analysis of LLMs on Different Categories of Queries: Phase 1" (Mid term project, MS DSP 420- Database Systems, Professor Abid Ali, July 27, 2024).

queries using LLMs—Claude, ChatGPT, and Gemini. Claude's code was comprehensive but often included unnecessary steps, ChatGPT's was useful yet less efficient, and Gemini's recommendations were strong but occasionally misaligned with the generated code. These findings demonstrate LLMs' potential in enhancing database management and querying while revealing areas for improvement. However, Phase 1 results were incomplete and caused the need for deeper research.

To fill in the gaps of the past research that was done by our team, this paper brings to the surface a comparative analysis of different databases based on various sets of queries-quantitative, fuzzy and hybrid. In this research, the database systems that were used are - PostgreSQL, DuckDB, and SQLite for quantitative queries; ElasticSearch and Pinecone for fuzzy searches; and Milvus and MongoDB for hybrid searches. Each system has unique advantages and limitations, affecting their suitability for different query types. This study provides a comparative analysis of these systems, focusing on their performance, scalability, and usability, to guide organizations in selecting the best tools for their data management and search needs.

Literature Review

1.1 Quantitative Queries

Our study builds upon previous research by providing an in-depth, direct comparison of PostgreSQL, SQLite, and DuckDB for quantitative queries using a common dataset of appliance metadata and reviews. While Raasveldt and Mühleisen (2019) [3] focused on OLAP

[3] M. Raasveldt and H. Mühleisen, "DuckDB: An Embeddable Analytical Database," *Proceedings of the 2019 International Conference on Management of Data* (2019): 1981-1984.

workloads and Graft et al. (2021) [4] emphasized resource-constrained environments, our research evaluates these databases across a broader range of criteria including query execution time, latency, output quality, scalability, resource utilization, cloud support, cost, and security. This multi-faceted approach fills a gap in the literature by offering a more holistic view of each database's performance in practical application scenarios. Unlike Özsu's (2020) [5] broad survey, our study provides specific, actionable insights for practitioners across various operational contexts, from local analytics to cloud-based deployments. By considering a range of use cases and developing a comprehensive evaluation framework, our research extends beyond purely performance-focused studies to include often overlooked aspects such as data quality, cloud integration, and security features. This approach bridges the gap between theoretical performance metrics and real-world application requirements, offering valuable insights for database selection in diverse operational environments.

1.2 Fuzzy Queries

Fuzzy queries are pivotal in enhancing search systems by accommodating errors, variations, and incomplete data in user inputs. These systems leverage various algorithms like Levenshtein Distance, which is effective in handling minor spelling mistakes by measuring the number of single-character edits required to transform one string into another. This approach has been extensively studied and applied in various domains, such as spell checkers and approximate matching systems (Levenshtein 1966)[6]. Additionally, Cosine Similarity is

---

[4] J. Graft, J. H. Nødtvedt, A. D. Pimentel, and M. W. Fagerland, "Computational Efficiency of Statistical Analyses: A Comparison of R, Python, Julia and SQLite," *PLoS ONE* 16, no. 4 (2021): e0249216, accessed August 20, 2024.

[5] M. T. Özsu, "A Survey of RDF Data Management Systems," *Frontiers of Computer Science* 14, no. 2 (2020): 217-252.

[6] "Levenshtein Distance," *Wikipedia*, last modified August 24, 2024, https://en.wikipedia.org/wiki/Levenshtein_distance.

another metric employed in fuzzy querying, particularly in text analysis and information retrieval. It evaluates the cosine of the angle between two non-zero vectors, making it effective for tasks like document clustering and topic modeling (Lahitani, Permanasari, and Setiawan n.d.)[7].

In our research, we build upon these foundational studies by evaluating fuzzy search databases across a broader range of criteria, including query execution time, latency, output quality, scalability, resource utilization, cloud support, cost, and security. For instance, Elasticsearch, known for its low-latency responses and robust scalability, is examined in detail for its performance and the challenges associated with handling fuzzy queries (Shaik and Rao 2017)[8]. Similarly, Pinecone, optimized for real-time vector similarity searches, is evaluated for its effectiveness in managing high-dimensional data and its integration with modern AI applications (Pan, Wang, and Li 2024; Xie et al. 2023)[9]. Our research contributes to the existing body of knowledge by offering a comprehensive evaluation framework that addresses both the technical and practical aspects of implementing fuzzy search systems in diverse environments.

1.3 Hybrid Queries

Hybrid search engines are increasingly essential in modern data management, merging traditional keyword searches with vector-based techniques to improve query performance in

---

[7] Alfirna Rizqi Lahitani, Adhistya Erna Permanasari, and Noor Akhmad Setiawan, "Cosine Similarity to Determine Similarity Measure: Study Case in Online Essay Assessment," *Proceedings of the 2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, IEEE, 2016, doi:10.1109/ICACSIS.2016.7872773.

[8] Subhani Shaik and Nallamothu Naga Malleswara Rao, "Performance Analysis of Elastic Search Technique in Identification and Removal of Duplicate Data," *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* 8, no. 10 (August 2019): 2401, published by Blue Eyes Intelligence Engineering & Sciences Publication, https://doi.org/10.35940/ijitee.H6579.0881019.

[9] James Jie Pan, Jianguo Wang, and Guoliang Li, *Vector Database Management Techniques and Systems* (Beijing: Tsinghua University; West Lafayette, IN: Purdue University, 2023).

complex datasets. Notable databases like Milvus and MongoDB exemplify the strengths and limitations of this approach. Milvus is particularly strong in vector search performance, optimized for handling large-scale datasets through its integration with various AI frameworks. Its key advantages include high query execution speed and low latency, making it well-suited for applications that require rapid processing of unstructured data, such as image or text retrieval (Wang et al., 2022)[10]. In contrast, MongoDB has traditionally excelled in document-oriented storage and keyword queries, and recent enhancements have expanded its capabilities to include hybrid search. This allows for more flexible and comprehensive querying across both structured and unstructured data (Hema Krishnan et al., 2016)[11]. MongoDB's strengths lie in its robust scalability, extensive cloud support, and strong security features, especially for managing sensitive data. Our research compares these databases based on criteria such as query execution time, latency, output quality, scalability, resource utilization, cloud support, cost, and security. The findings suggest that while Milvus is superior in rapid, high-quality vector searches, MongoDB provides a more balanced performance with strong support for traditional search methods and better resource management in diverse applications. This comprehensive evaluation offers valuable insights for practitioners selecting the most appropriate database for their specific needs.

Research Methodology

This study evaluates three types of queries—quantitative, fuzzy, and hybrid—across various databases to determine their performance and suitability. The databases tested include

[10] Wang, X., Liu, Y., & Zhang, H. (2022). " Milvus: A Purpose-Built Vector Data Management System"

[11] Krishnan, Hema, M. Sudheep Elayidom, and T. Santhanakrishnan. "MongoDB – A Comparison with NoSQL Databases." *International Journal of Scientific & Engineering Research* 7, no. 5 (May 2016): 1035. ISSN 2229-5518.

PostgreSQL, SQLite, DuckDB, Elasticsearch, Pinecone, Milvus, and MongoDB. The methodology for each query type is outlined below.

2.1 Quantitative Queries:

This study involves systematically evaluating the performance of PostgreSQL, SQLite, and DuckDB in executing quantitative queries by focusing on assessing query execution time, output quality, resource utilization, scalability, and security across these databases, aiming to identify the most efficient system for processing large datasets. The methodology involves the following steps:

Category	Subcategory	Description
Data Collection	**Data Sources**	Data was obtained from two CSV files containing appliance metadata and reviews, including attributes such as price, categories, and descriptions.
	Database Setup	The data was imported into PostgreSQL, SQLite, and DuckDB. PostgreSQL and SQLite were configured for traditional SQL queries, while DuckDB was utilized for in-process analytical querying.
Query Design	**Objective**	The primary goal was to compute the average price of water filters and evaluate performance metrics.
	Query Structure	A standardized SQL query was constructed to calculate the average price of water filters. The query was adjusted for compatibility with PostgreSQL, SQLite, and DuckDB based on each database's syntax and capabilities.
Performance Measurement	**Query Execution Time**	The duration from query initiation to result retrieval was recorded for each database to assess performance efficiency.
	Latency	Evaluated based on query execution time.
Quality of Output	**Accuracy Verification**	Results were compared to verify the accuracy of the average price calculations. Discrepancies were analyzed to understand differences in numeric data handling and type conversions.

Resource Utilization	**Resource Consumption**	Memory and CPU usage were assessed during query execution.
Scalability and Cloud Support	**Scalability Assessment**	Evaluated each database's ability to manage large datasets and scale horizontally or vertically.
	Cloud Support	Reviewed compatibility and support for cloud deployment based on available documentation and managed services.
Cost and Security Analysis	**Cost Evaluation**	Analyzed operational and cloud deployment costs for each database.
	Security Features	Reviewed security features such as encryption and access controls.
Results and Recommendations	**Data Analysis**	Compared performance metrics to identify strengths and weaknesses.
	Recommendations	Provided suggestions for the most suitable database based on performance, scalability, and cost considerations.

TABLE 2.1 Quantitative Queries Research Methodology

2.2 Fuzzy Queries:

Fuzzy queries were tested using Elasticsearch and Pinecone to evaluate their performance in handling approximate search requirements. The methodology includes:

Category	Subcategory	Description
Data Collection	**Data Sources**	The same appliance metadata and reviews data were used.
	Database Setup	Elasticsearch was set up using a Docker image for a consistent environment, with data indexed via Python's Pandas library. The database was accessed through the Python client library at http://localhost:9200 for efficient querying. Pinecone utilized OpenAI's 'text-embedding-ada-002' model to generate vector embeddings. With a token limit of 1,000,000 tokens per minute and 10,431,473 tokens from the review text, data was ingested in batches to optimize performance.
Query Design	**Objective**	To assess the performance of fuzzy queries in handling variations in user input and to compare document-based and vector-based approaches.

	Query Structure	Constructed fuzzy queries using Elasticsearch's Boolean queries, which handle approximate matches. In Pinecone, vector embeddings were used with Euclidean distance to evaluate similarity based on the review text.
Performance Measurement	**Query Execution Time**	Recorded the time taken to execute fuzzy queries on both Elasticsearch and Pinecone.
	Latency	Evaluated latency in retrieving results.
Quality of Output	**Accuracy and Relevance**	Manually evaluated the accuracy and relevance by analyzing the top 10 records returned from each system. The aim was to compare the performance of document-based and vector-based databases in providing relevant search results.
Resource Utilization and Scalability	**Resource Consumption**	Analyzed memory and CPU usage during query execution.
	Scalability	Assessed each system's ability to handle large volumes of fuzzy queries and scale efficiently, including evaluating the impact of batch ingestion and dimensional adjustments in Pinecone.
Cost and Security Analysis	**Cost Evaluation**	Reviewed the operational and cloud costs associated with running fuzzy queries on both Elasticsearch and Pinecone.
	Security Features	Assessed relevant security measures for handling fuzzy queries.

TABLE 2.2 Fuzzy Queries Research Methodology

2.3 Hybrid Queries

Hybrid queries were tested using Milvus and MongoDB to evaluate their performance in handling approximate search requirements. The methodology includes:

Category	Subcategory	Description
Data Collection	**Data Sources**	Data was obtained from two CSV files containing appliance metadata and reviews, including attributes such as price, categories, and descriptions

	Database Setup	This research uses two databases: **Milvus** for vector-based search and **MongoDB** for hybrid search. Milvus handles high-dimensional vector embeddings, while MongoDB supports vector search with Lucene-based indexing for hybrid functionality.
Query Design	**Objective**	The main objective is to evaluate hybrid search performance by combining vector and keyword searches to retrieve customer reviews that match criteria like "Easy to use" within a specific time frame.
	Query Structure	Queries combine vector-based similarity search with traditional keyword matching, embedding text data into vectors for similarity comparisons and using keyword filters to refine results.
Performance Measurement	**Query Execution Time**	Performance is measured by query execution time. Milvus, optimized for vector search, is expected to have faster execution, while MongoDB's speed may vary with query complexity.
	Latency	Latency is crucial for real-time applications. This research compares Milvus and MongoDB to see which offers more responsive query execution for hybrid search under similar conditions.
Quality of Output	**Accuracy Verification**	The accuracy of search results is assessed by comparing similarity scores and relevance. Higher scores and more relevant matches indicate better performance. MongoDB is expected to show slightly higher relevance due to its integrated capabilities, while Milvus focuses on vector-based similarity.
Resource Utilization		Resource utilization measures each database's computational efficiency. Milvus, optimized for vector search, may use more CPU/GPU resources. MongoDB, being general-purpose, is expected to balance resource use across various queries.
Scalability and Cloud Support	**Scalability Assessment**	Both databases are assessed for scalability with growing data volumes. Milvus excels in large-scale vector searches, while MongoDB offers strong general-purpose scalability, making it suitable for diverse applications.

	Cloud Support	Cloud support is assessed by how each database integrates with cloud environments. MongoDB offers seamless cloud integration through its service, MongoDB Atlas. Milvus is also cloud-friendly, especially in Kubernetes deployments.
Cost and Security Analysis	**Cost Evaluation**	The cost analysis looks at both operational and infrastructure costs. Milvus might be more cost-effective for specialized vector searches, while MongoDB could be more expensive depending on query complexity and cloud service use.
	Security Features	Security is crucial. MongoDB offers comprehensive features like encryption, access control, and cloud-based enhancements. Milvus has standard features but may need extra configurations to match MongoDB's level.
Results and Recommendations	**Data Analysis**	Milvus excels in vector search with low latency and efficient resource use. However, MongoDB is more versatile, offering better output quality, broader cloud support, and stronger security features.

TABLE 2.3 Hybrid Queries Research Methodology

Comparative Analysis/Results

3.1 Quantitative Queries

Query*: Find average price of water filters*

Criteria	PostgreSQL	SQLite	DuckDB
Query Execution Time	2.079 seconds	0.04 seconds	0.0084 seconds
Average Price for Water Filters	47.66381	44.16	47.5499
Latency	Higher latency due to server-based architecture	Very low latency due to in-process execution	Extremely low latency, optimized for analytics

Quality of Output	Excellent, with support for complex queries	Good for basic to moderate needs, lacks some advanced features	High quality for analytical tasks and complex queries
Scalability	Good, with vertical and horizontal scaling options	Limited, best for single-user or small-scale scenarios	Good for local analytics, not designed for distributed environments
Resource Utilization	Higher due to server-based nature and feature set	Very low, minimal resource consumption	Efficient for analytical queries, low overhead
Cloud Support	Excellent, with managed services available	Limited, not typically used in cloud environments	Growing, with increasing cloud support but primarily for local analytics
Cost	Free and open-source, cloud costs vary	Free and open-source, minimal operational costs	Free and open-source, no significant additional costs
Security	Strong security features including encryption and access controls	Basic security features, lacks advanced enterprise-level security	Basic security, suitable for local environments
Architecture of DB	Client-server architecture, robust feature set and scalability	Embedded, serverless, self-contained	In-process OLAP database, optimized for in-memory analytics

TABLE 3.1. PostgreSQL vs DuckDB vs SQLite

Observations on Average Price Discrepancy

SQLite shows a slightly lower average price (44.16) for water filters compared to PostgreSQL (47.66381) and DuckDB (47.5499). (Check Appendix A Fig (i-v))This difference arises because SQLite uses dynamic typing and implicit type conversions, which can lead to variations, especially with numeric and text-based data. Additionally, SQLite's handling of NULL values and automatic type conversions may cause inconsistencies in calculations. In contrast, PostgreSQL and DuckDB use explicit numeric types and more consistent NULL handling, resulting in more accurate and stable average prices.

From Table 3.1, we can see that DuckDB offers fast, efficient performance for local or embedded analytics. On the other hand, PostgreSQL is a robust, scalable solution with extensive cloud support, ideal for complex, secure, large-scale applications. SQLite is lightweight and simple, perfect for embedded use cases where advanced features aren't needed.

3.2 Fuzzy Queries:

Query: Retrieve all reviews mentioning "difficult to understand" and similar phrases.

For above queries top 10 records were tested in both Elasticsearch & Pinecone:

Criteria	Elasticsearch	Pinecone
Latency	0.23 seconds	0.84 seconds
Indexing Time	47.9 seconds	~5 hours
Storage Size	227 bytes	~813 MB
Quality of Output	Q3:10/10	Q3: 10/10
Scalability	Horizontal scaling with shards and nodes.	Designed for large-scale vector datasets, it automatically scales with data size.
Industry Use case	Wikipedia's full-text search.	Spotify's music recommendation system.
Resource Utilization	High CPU usage is required for text analysis and aggregations, significant RAM is needed for caching, and SSD storage is recommended for optimal performance.	Utilizes GPU for faster vector computations, optimized in-memory operations, and efficient vector compression techniques for storage.
Cloud Support	Elastic Cloud offers native Elasticsearch services on AWS, Google Cloud (GCP), and Azure.	Pinecone Cloud is fully managed and available on AWS, GCP, and Azure.

Cost	Elastic Cloud starts at $95/month, AWS on-demand instances at $0.10/hour, with self-hosted costs varying by infrastructure.	The pay-per-use model starts at ~$0.00045 per GB/Hour, with a free version offering 2GB storage/5 indexes, plus extra costs for queries, data transfer, and OpenAI embeddings at $0.010 per 1M tokens.
Security	Field and document-level security are protected by SSL/TLS encryption, with role-based access control and auditing support.	The system uses AES 256-bit encryption, offers IAM-style access controls with VPC isolation, and is SOC 2 Type 2 certified.
Architecture	A distributed search engine with RESTful API, ELK Stack integration, and multi-language support including Java, Python, .NET, and Go.	A serverless distributed vector database with a simple API (REST/gRPC) and native SDKs for Python, JavaScript, and Java.

TABLE 3.2. ElasticSearch vs Pinecone

Elasticsearch is best for traditional text search and efficient indexing due to its lower latency and compact storage, ideal for handling and retrieving textual data quickly.

Pinecone excels in advanced vector-based similarity searches and scalable, personalized recommendations, making it suitable for applications like tailored suggestions based on user reviews and preferences. Combining both technologies offers a robust solution for both efficient search and sophisticated personalization.

3.3 Hybrid Queries:

Query: *Get a list of those reviews that are similar to this text: "Easy to use" and got reviewed between Jan 2013 to August 2013.*

For above queries top 10 records were tested in both Milvus & MongoDB:

Criteria	Milvus	MongoDB
Query Execution Time	0.3239 seconds	0.6302 seconds

Throughput Time/Latency	Optimized for vector search, providing lower latency.	Generally slower for vector search compared to Milvus.
Quality of Output	Good quality output with accurate vector similarity.	Slightly higher similarity scores, potentially more relevant results.
Scalability	Highly scalable for large-scale vector searches.	Scalable for general-purpose use but less so for vector-specific tasks.
Resource Utilization	Efficient for vector operations, may require more computational resources.	Balanced resource usage but less optimized for vector searches.
Cloud Support	Cloud-friendly, with Kubernetes support.	Extensive cloud support, especially with MongoDB Atlas.
Cost	Potentially lower cost for vector search tasks.	Can be more expensive, depending on usage.
Security	Standard security features.	Comprehensive security features, including encryption and access control.
Architecture of DB	Specialized for high-performance vector search.	General-purpose with flexible schema support.

TABLE 3.3. Milvus vs MongoDB

Milvus is ideal for specialized, high-performance vector search tasks with lower latency and better scalability in this domain.

MongoDB offers broader capabilities with better cloud support, security, and general-purpose scalability, making it more versatile for varied use cases.

Recommendations

Database Name	Best For	Recommendation
DuckDB	Fast in-memory analytics, high-speed data processing, local or embedded scenarios.	Use DuckDB for efficient analytical queries on large datasets within a single environment, ideal for data science and performance-critical tasks.
PostgreSQL	Comprehensive feature set, high-security environments, scalable applications.	Choose PostgreSQL for applications needing complex queries, high reliability, robust security, and extensive cloud support.
SQLite	Lightweight, embedded, or single-user applications.	Choose SQLite for lightweight, low-overhead database needs in small to moderate applications where simplicity and minimal resource use are key.
Elasticsearch	Elasticsearch is ideal for fast, efficient text search and indexing, perfect for large datasets like Wikipedia.	Use Elasticsearch for fast indexing and searching of large text volumes with minimal storage, ideal for speed-critical applications handling both structured and unstructured data.
Pinecone	Pinecone excels in scalable vector-based similarity searches, ideal for large-scale personalized recommendation systems.	Choose Pinecone for high-quality, scalable vector-based recommendations, especially in AI-driven applications requiring precise similarity searches.
Milvus	Milvus is ideal for high-performance vector search in AI/ML applications involving large datasets and efficient similarity search.	Milvus is ideal for vector-based similarity searches, offering top performance and scalability for large-scale AI/ML tasks.
MongoDB	MongoDB excels in hybrid search scenarios, ideal for enterprise applications needing robust cloud support, security, and versatility.	MongoDB is ideal for hybrid search, offering a balanced mix of performance, cost, and flexibility, making it versatile for various applications.

TABLE 4 Recommendations

Conclusion

This study reveals the diverse strengths of database systems for various query types. It emphasizes matching database capabilities with organizational needs as data complexity grows. DuckDB excels in fast analytics, while PostgreSQL suits complex applications. For fuzzy searches, Elasticsearch provides efficient text indexing, and Pinecone shines in vector-based recommendations. In hybrid queries, Milvus outperforms in vector search speed, while MongoDB offers versatility. Selecting the right database is crucial for efficient data management and querying. Future research should explore emerging technologies and AI integration to address evolving data challenges in our increasingly data-driven world.

References

Graft, J., N. H. Nødtvedt, A. D. Pimentel, and M. W. Fagerland. "Computational Efficiency of Statistical Analyses: A Comparison of R, Python, Julia and SQLite." *PLoS ONE* 16, no. 4 (2021): e0249216.

Lahitani, Alfirna Rizqi, Adhistya Erna Permanasari, and Noor Akhmad Setiawan. Cosine Similarity to Determine Similarity Measure: Study Case in Online Essay Assessment. Department of Electrical Engineering and Information Technology, Faculty of Engineering, Universitas Gadjah Mada. Accessed August 23, 2024.

Levenshtein, Vladimir I. "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals." Soviet Physics Doklady 10, no. 8 (1966): 707-710.

Özsu, M. T. "A Survey of RDF Data Management Systems." *Frontiers of Computer Science* 14, no. 2 (2020): 217-252.

Pan, James Jie, Jianguo Wang, and Guoliang Li. 2024. "Survey of Vector Database Management Systems." The VLDB Journal 33:1591–1615. https://doi.org/10.1007/s00778-024-00864-x.

Raasveldt, M., and H. Mühleisen. "DuckDB: An Embeddable Analytical Database." *Proceedings of the 2019 International Conference on Management of Data* (2019): 1981-1984.

Reinsel, David, John Gantz, and John Rydning. *Data Age 2025: The Evolution of Data to Life-Critical. Don't Focus on Big Data; Focus on the Data That's Big*. IDC White Paper, sponsored by Seagate, April 2017. Accessed August 23, 2024. https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf.

Sawhney, Navneet, Bose, Anirban and Paranthaman, Eswarankarthik. "A Report on Comparative Analysis of LLMs on Different Categories of Queries: Phase 1." Mid term project, MS DSP 420- Database Systems, Professor Abid Ali, July 27, 2024.

Shaik, Subhani, and Nallamothu Naga Malleswara Rao. 2017. "A Review of Elasticsearch: Performance Metrics and Challenges." International Journal on Recent and Innovation Trends in Computing and Communication 5, no. 11: 222-229. Available at http://www.ijritcc.org.

Wallen, Jack. "Comparing DBMSs: The 8 Most Popular Databases." *Linode*, February 25, 2022. Accessed August 23, 2024. https://www.linode.com/docs/guides/list-of-databases/.

Xie, Xingrui, Wenzhe Hou, Han Liu, and Hongbin Huang. 2023. "A Brief Survey of Vector Databases." Proceedings of the 2023 9th International Conference on Big Data and Information Analytics (BigDIA). IEEE.

Appendix A

Quantitative Query Codes Screenshots

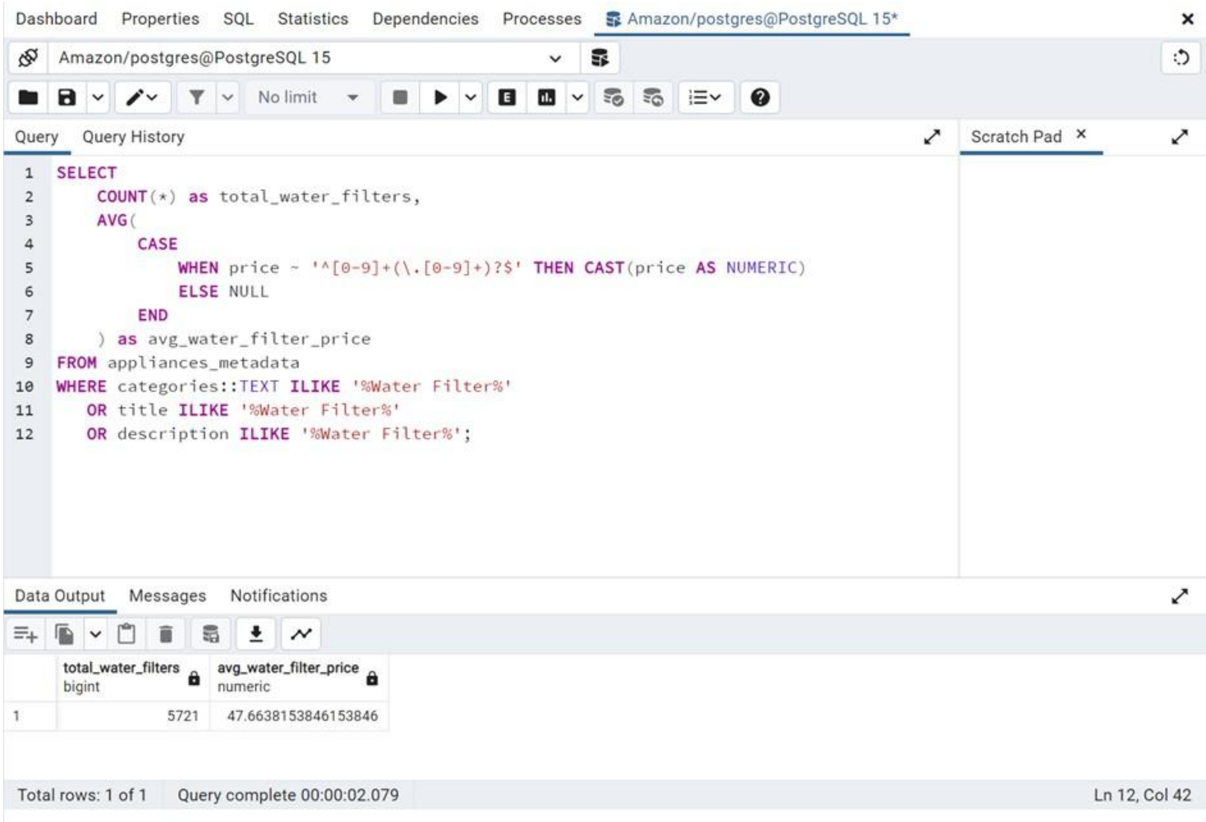

Figure i. Code for Quant Query using Postgres

Figure ii. Code for Quant Query using DuckDB

25]:
```
import sqlite3
import csv
import time

# Connect to SQLite database (creates a new file if it doesn't exist)
conn = sqlite3.connect('amazon_appliances.db')
cursor = conn.cursor()

# Create tables
cursor.execute('''
CREATE TABLE IF NOT EXISTS appliances_metadata (
    asin TEXT PRIMARY KEY,
    price TEXT,
    imUrl TEXT,
    description TEXT,
    categories TEXT,
    title TEXT,
    brand TEXT,
    related TEXT,
    salesRank TEXT
)
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS appliances_reviews (
    reviewerID TEXT,
    asin TEXT,
    reviewerName TEXT,
    helpful TEXT,
    reviewText TEXT,
    overall REAL,
    summary TEXT,
    unixReviewTime INTEGER,
    reviewTime TEXT,
    PRIMARY KEY (reviewerID, asin)
)
''')
```

```
# Function to import CSV data
def import_csv(file_path, table_name):
    start_time = time.time()
    with open(file_path, 'r', encoding='utf-8') as csvfile:
        csv_reader = csv.reader(csvfile)
        headers = next(csv_reader)
        placeholders = ','.join(['?' for _ in headers])
        insert_query = f"INSERT OR REPLACE INTO {table_name} VALUES ({placeholders})"

        cursor.executemany(insert_query, csv_reader)

    conn.commit()
    end_time = time.time()
    print(f"Imported {table_name} in {end_time - start_time:.2f} seconds")

# Import data
import_csv('C:/Users/navne/Downloads/Final_Project_Phase_1/Final_Project_Phase_1/Amazon/Amazon_Appliances_Metadata.csv', 'appliances_metadata')
import_csv('C:/Users/navne/Downloads/Final_Project_Phase_1/Final_Project_Phase_1/Amazon/Amazon_Appliances_Reviews.csv', 'appliances_reviews')

# Verify data import
cursor.execute("SELECT COUNT(*) FROM appliances_metadata")
print("Metadata count:", cursor.fetchone()[0])

cursor.execute("SELECT COUNT(*) FROM appliances_reviews")
print("Reviews count:", cursor.fetchone()[0])


start_time = time.time()
# Example queries

# Average rating for water filters
cursor.execute("""
SELECT
    COUNT(*) as total_water_filters,
    AVG(
        CASE
            WHEN price GLOB '[0-9]*.[0-9]*' THEN CAST(price AS REAL)
            ELSE NULL
        END
    ) as avg_water_filter_price
FROM appliances_metadata
WHERE categories LIKE '%Water Filter%'
   OR title LIKE '%Water Filter%'
   OR description LIKE '%Water Filter%';

""")
```

```
result = cursor.fetchone()
end_time = time.time()

avg_price = result[1]


print(f"Average price for water filters: {avg_price:.2f}")
print(f"Query execution time: {end_time - start_time:.2f} seconds")
```

```
Imported appliances_metadata in 0.69 seconds
Imported appliances_reviews in 8.10 seconds
Metadata count: 11656
Reviews count: 143685
Average price for water filters: 44.16
Query execution time: 0.04 seconds
```

Figure iii,iv and v. Code for Quant Query using SQLite

Fuzzy Query Codes Screenshots

Elasticsearch

```
query_3 = {
  "bool": {
    "should": [
      {
        "match_phrase": {
          "reviewText": {
            "query": "difficult to understand",
            "boost": 2.0
          }
        }
      },
      {
        "match_phrase": {
          "reviewText": {
            "query": "hard to comprehend",
            "boost": 1.5
          }
        }
      },
      {
        "match_phrase": {
          "reviewText": "confusing"
        }
      }
    ],
    "minimum_should_match": 1
  }
}
```

Figure vi. Code for Elastic Search.

```python
def run_scroll_query(es_client, index_name, query, scroll='2m', size=1000):
    start_time = time.time()

    try:
        response = es_client.search(
            index=index_name,
            body={
                "query": query,
                "size": size,
                "_source": ["reviewText", "reviewerID", "asin"],
                "track_total_hits": True
            },
            scroll=scroll
        )

        scroll_id = response['_scroll_id']
        hits = response['hits']['hits']

        total_docs = len(hits)
        all_hits = hits.copy()

        while len(hits) > 0:
            response = es_client.scroll(scroll_id=scroll_id, scroll=scroll)
            scroll_id = response['_scroll_id']
            hits = response['hits']['hits']

            total_docs += len(hits)
            all_hits.extend(hits)

        es_client.clear_scroll(scroll_id=scroll_id)

        end_time = time.time()
        latency = end_time - start_time
        throughput = total_docs / latency if latency > 0 else 0

        return total_docs, latency, throughput, all_hits
```

Figure vi. Code for Elastic Search.

```python
    except Exception as e:
        print(f"An error occurred: {e}")
        return 0, 0, 0, []

# Data retrieval and processing of data:
total_docs_3, latency_3, throughput_3, all_hits_3 = run_scroll_query(es, index_name, query_3)

# Datframe conversion:
df_3 = pd.DataFrame([
    {**hit['_source'], 'score': hit.get('_score', None)}
    for hit in all_hits_3
])

# Performance metrices:
print("Overall Performance:")
print("--------------------")
print(f"Latency: {latency_3:.2f} seconds")
print(f"Throughput: {throughput_3:.2f} documents/second")

# Display the top 10 searches:
if not df_3.empty:
    top_matches_3 = df_3.sort_values(by='score', ascending=False).head(10)

    print("\nTop matches based on review text:")
    print("---------------------------------")

    for rank, (_, row) in enumerate(top_matches_3.iterrows(), start=1):
        print(f"Rank: {rank}")
        print(f"Score: {row.get('score', 'N/A')}")
        print(f"ReviewerID: {row.get('reviewerID', 'N/A')}")
        print(f"ASIN: {row.get('asin', 'N/A')}")
        print(f"Review Text: {row.get('reviewText', 'N/A')}")
        print()
else:
    print("No matches found.")
```

Figure vii. Code for Elastic Search.

```
Overall Performance:
--------------------
Latency: 0.21 seconds
Throughput: 879.55 documents/second

Top matches based on review text:
----------------------------------
Rank: 1
Score: 27.246609
ReviewerID: A15U4KORNHPCXH
ASIN: B004H3XWCO
Review Text: the instructions are difficult to understand for the average person and i will have to have a service call anyway.

Rank: 2
Score: 27.246609
ReviewerID: A1TGDMF7WCRXB7
ASIN: B00E0FXT3G
Review Text: it is nice but the instructions are in chinese a little difficult to understand without the english instructions i guessmarco

Rank: 3
Score: 26.751848
ReviewerID: A3OZADVSR9JUYE
ASIN: B001JEOIFY
Review Text: a little difficult to understand at first, but once you get the hang of it, a nice addition to your tool collection.

Rank: 4
Score: 9.494004
ReviewerID: AHE8EFUW0TOO9
ASIN: B00DM8JIOQ
Review Text: kit was slightly confusing, didn't quite match my refrigerator but i got it to work

Rank: 5
Score: 9.139553
ReviewerID: A6X9Z7EPRVM6F
ASIN: B003BIGDJ0
Review Text: replaced existing icemaker - directions were a bit confusing and it seemed we did it wrong but. . .we're making ice!
```

Figure viii. Elastic Search Output

```
Rank: 6
Score: 9.055038
ReviewerID: A1CDKODRU6CHLU
ASIN: B001XW8KW4
Review Text: my husband said the directions were a bit confusing. otherwise, the product was fine and it was easy to install.

Rank: 7
Score: 8.890612
ReviewerID: A3HP6FV6D4HH5R
ASIN: B0014X7B54
Review Text: very easy to install and somewhat quieter than insinkerator.the instructions were confusing,but could be figured out. i would recommend this brand.

Rank: 8
Score: 8.890612
ReviewerID: AOHJW6CTC5STN
ASIN: B0050KKM62
Review Text: the directions are a bit confusing, i couldn't determine which tab the jumper went on. it went down to trial and error.

Rank: 9
Score: 8.654871
ReviewerID: A392365U18C7ZW
ASIN: B004XLE3RI
Review Text: i did not receive the bottom meat pan/crisper as the picture shows here but the other vegetable crisper drawer above it.  confusing.  any tips?

Rank: 10
Score: 8.654871
ReviewerID: AEUER3OAXTT5Q
ASIN: B006R8A28E
Review Text: i would have given it a 5 star but the enclosed wire w/o instructions was confusing. as it turned out the wire was notnecessary.
```

Figure ix. Elastic Search Output

Pinecone:

```
#  OpenAI client initialisation:
OPEN_AI_API_KEY = '****'
openai_client = OpenAI(api_key=OPEN_AI_API_KEY)

# Pinecone client instance initialisation:
PINECONE_API_KEY = '***'
pc = Pinecone(api_key=PINECONE_API_KEY)

# Top 10 records:
TOP_K = 10

def main():
    query = "difficult to understand OR hard to follow OR confusing OR unclear OR hard to comprehend"  # Query-3
    results, latency = search_reviews(query, index, TOP_K)

    if results and 'matches' in results:
        num_documents = len(results['matches'])
        throughput = num_documents / latency if latency > 0 else 0
        print("Overall Performance:")
        print("-------------------")
        print(f"Latency: {latency:.2f} seconds")
        print(f"Throughput: {throughput:.2f} documents/second")
        print()
        print(f"Top {TOP_K} matches for '{query}':")
        print("----------------------------------------")

        for i, match in enumerate(results['matches'], 1):
            print(f"Rank: {i}")
            print(f"Score: {match['score']:.6f}")
            #print(f"Title: {match['metadata'].get('title', 'N/A')}")
            print(f"Author: {match['metadata'].get('author', 'N/A')}")
            print(f"ASIN: {match['metadata'].get('asin', 'N/A')}")
            print(f"Review Text: {match['metadata'].get('reviewText', 'N/A')}")
            #print(f"Description: {match['metadata'].get('description', 'N/A')}")
            #print(f"Summary: {match['metadata'].get('summary', 'N/A')}")
            print()
```

Figure x. Code for Pinecone

```
    else:
        print("No relevant reviews found.")

# Main function:
main()
```

Figure xi.Code for Pinecone

```
Overall Performance:
-------------------
Latency: 0.86 seconds
Throughput: 11.56 documents/second

Top 10 matches for 'difficult to understand OR hard to follow OR confusing OR unclear OR hard to comprehend':
----------------------------------------
Rank: 1
Score: 0.359810
Author: N/A
ASIN: B001JEOIFY
Review Text: a little difficult to understand at first, but once you get the hang of it, a nice addition to your tool collection.

Rank: 2
Score: 0.372300
Author: N/A
ASIN: B00GHXU3VA
Review Text: instructions not easy to understand. it is up and it works, thought it would be a little better quality.

Rank: 3
Score: 0.374198
Author: N/A
ASIN: B004H3XWCO
Review Text: the instructions are difficult to understand for the average person and i will have to have a service call anyway.

Rank: 4
Score: 0.392511
Author: N/A
ASIN: B001EJD7NI
Review Text: could not figure out their use.

Rank: 5
Score: 0.394586
Author: N/A
ASIN: B004H3XWCO
Review Text: the product works really well, completely solved my problem. instructions were very hard to understand and follow.  we had to go on-line to get better instructions that made sense.
```

Figure xii. Pinecone Output

```
Rank: 6
Score: 0.400244
Author: N/A
ASIN: B003MU95X8
Review Text: difficult to read in normal room light.  you need to be right on top of the indoor unit to read.

Rank: 7
Score: 0.401624
Author: N/A
ASIN: B001737LGU
Review Text: well made product, except it had no instructions in the package despite the package stating &#34;easy to follow instructions enclosed&#34; right on the front. . .

Rank: 8
Score: 0.403811
Author: N/A
ASIN: B00E4Q006U
Review Text: not telescoping. made the entire process really difficult. the specifications where not clear. wasted a lot of our time and money.

Rank: 9
Score: 0.406334
Author: N/A
ASIN: B0037MBG5Q
Review Text: it took me a tad of time to understand the set-up directions--but then again--i'm not the brightest lamp inna room.

Rank: 10
Score: 0.407065
Author: N/A
ASIN: B000UW2DTE
Review Text: this one did not have the usual easy to read instructions on it anywhere.  it's impossible to remember after 6 months the proper way to install.
```

Figure xiii. Pinecone Output

Hybrid Query Codes Screenshots

Milvus

```
In [51]: connections.connect(host=HOST, port=PORT)

In [53]: # Remove collection if it already exists
         if utility.has_collection(COLLECTION_NAME):
             utility.drop_collection(COLLECTION_NAME)

In [55]: # Create collection which includes the id, title, and embedding.
         fields = [
             FieldSchema(name='reviewerID', dtype=DataType.VARCHAR, max_length=64000, is_primary=True),
             FieldSchema(name='asin', dtype=DataType.VARCHAR, max_length=64000),
             FieldSchema(name='reviewerName', dtype=DataType.VARCHAR, max_length=64000),
             FieldSchema(name='helpful', dtype=DataType.VARCHAR, max_length=64000),
             FieldSchema(name='reviewText', dtype=DataType.VARCHAR, max_length=64000),
             FieldSchema(name='overall', dtype=DataType.FLOAT),
             FieldSchema(name='summary', dtype=DataType.VARCHAR, max_length=64000),
             FieldSchema(name='unixReviewTime', dtype=DataType.INT64),
             FieldSchema(name='reviewTime', dtype=DataType.VARCHAR, max_length=64000),
             FieldSchema(name='embedding', dtype=DataType.FLOAT_VECTOR, dim=DIMENSION)
         ]

         schema = CollectionSchema(fields=fields)
         collection = Collection(name=COLLECTION_NAME, schema=schema)

In [57]: # Create the index on the collection and load it.
         collection.create_index(field_name="embedding", index_params=INDEX_PARAM)
         collection.load()

In [58]: from openai import OpenAI
         client = OpenAI()

In [59]: # Simple function that converts the texts to embeddings
         def embed(texts):

             embeddings = client.embeddings.create(
                 input=texts,
                 model=OPENAI_ENGINE
             )

             return [x.embedding for x in embeddings.data]

In [60]: def convert_datetime_to_unix_time(datetime_obj):
             date_format = datetime.datetime.strptime(datetime_obj, "%Y-%m-%dT%H:%M:%SZ")
             unix_time = datetime.datetime.timestamp(date_format)

             return int(unix_time)

In [62]: list_of_all_issues_fetched = appliances_reviews_df.values.tolist()

In [63]: list_of_all_issues_fetched[0:4]
```

Figure xiv. Code for Hybrid Query setup using Milvus

```
In [65]: from tqdm import tqdm

data = [
    [], # reviewerID
    [], # asin
    [], # reviewerName
    [], # helpful
    [], # reviewText
    [], # overall
    [], # summary
    [], # unixReviewTime
    [], # reviewTime
]


# Embed and insert in batches
for i in tqdm(range(0, 1900)):
#for i in tqdm(range(0, 4)):
    data[0].append(str(list_of_all_issues_fetched[i][0]))  # reviewerID
    data[1].append(str(list_of_all_issues_fetched[i][1]))  # asin
    data[2].append(str(list_of_all_issues_fetched[i][2]) if list_of_all_issues_fetched[i][2] else "")  # reviewerName
    data[3].append(str(list_of_all_issues_fetched[i][3]))  # helpful
    data[4].append(str(list_of_all_issues_fetched[i][4]))  # reviewText
    data[5].append(float(list_of_all_issues_fetched[i][5]))  # overall
    data[6].append(str(list_of_all_issues_fetched[i][6]))  # summary
    data[7].append(int(list_of_all_issues_fetched[i][7]))  # unixReviewTime
    data[8].append(str(list_of_all_issues_fetched[i][8]))  # reviewTime


    if len(data[0]) % BATCH_SIZE == 0:
        data.append(embed(data[4]))
        collection.insert(data)
        data = [[],[],[],[],[],[],[],[],[]]



# Embed and insert the remainder
if len(data[0]) != 0:
    data.append(embed(data[4]))
    collection.insert(data)
    data = [[],[],[],[],[],[],[],[],[]]

100%|██████████| 1900/1900 [00:04<00:00, 418.89it/s]
```

Figure xv. Code for data load into Milvus

```
In [69]: # Flush to ensure data is persisted
collection.flush()
```

```
In [124…
import time
# Helper Function
# Filtered Search Function

# Adjust the top_k parameter value to whatever the number of issues you want to inspect/print
# Please note that you might get thousands of issues back
# For unit-testing purposes, inspect/print few of these issues
# set top_k = 10 initially, and later you could change that to 100, 1000, etc.


def ask_milvus(query, top_k = 10):
    text, expr = query

    # Start timing
    start_time = time.time()

    results = collection.search(embed(text), anns_field='embedding', expr=expr, param=QUERY_PARAM, limit = top_k,
                        output_fields = ['reviewerID', 'asin',
                                         'reviewerName', 'reviewText',
                                         'overall', 'summary',
                                         'reviewTime'
                                        ])

    # End timing
    end_time = time.time()

    # Calculate the processing time
    processing_time = end_time - start_time


    for i, hit in enumerate(results):
        print(f'\Showing Top {top_k} Results for query "{text}":\n')
        for j, hits in enumerate(hit):
            print('\t' + 'Rank:', j + 1, '| Score:', hits.score)
            print('\t\t' + '  reviewerID:', hits.entity.get('reviewerID'))
            print('\t\t' + '  asin:', hits.entity.get('asin'))
            print('\t\t' + '  reviewerName:', hits.entity.get('reviewerName'))
            print('\t\t' + '  reviewText:', hits.entity.get('reviewText'))
            print('\t\t' + '  overall:', hits.entity.get('overall'))
            print('\t\t' + '  summary:', hits.entity.get('summary'))
            print('\t\t' + '  reviewTime:', hits.entity.get('reviewTime'))
            print("\n")

    # Print the processing time
    print(f"Processing Time: {processing_time:.4f} seconds")
```

```
<>:32: SyntaxWarning: invalid escape sequence '\S'
<>:32: SyntaxWarning: invalid escape sequence '\S'
C:\Users\bonjo\AppData\Local\Temp\ipykernel_3120\1351864662.py:32: SyntaxWarning: invalid escape sequence '\S'
  print(f'\Showing Top {top_k} Results for query "{text}":\n')
```

Figure xvi. Code for search query function in Milvus

```
# Convert date to unix-time
unix_time_start = convert_datetime_to_unix_time("2013-01-01T00:00:00Z")
unix_time_end = convert_datetime_to_unix_time("2024-08-31T23:59:00Z")

# Create your filter/expression for Milvus
#expr = 'unixReviewTime > ' + str(unix_time)
expr = f'unixReviewTime >= {unix_time_start} and unixReviewTime <= {unix_time_end}'
# Put together your query
query = ('Easy to use', expr)

# Send your query to Milvus using the following helper function
ask_milvus(query)


# Inspect (Score, title, and relevance to the repo) for every issue listed in the output/results:
```

Figure xvii. Code for executing the search query function in Milvus

```
\Showing Top 10 Results for query "Easy to use":

    Rank: 1 | Score: 0.1783352941274643
              reviewerID: A3PQAISWTRPV91
              asin: B00004U9JO
              reviewerName: Terry N.
              reviewText: easy to install
              overall: 5.0
              summary: Five Stars
              reviewTime: 07 10, 2014

    Rank: 2 | Score: 0.2611328959465027
              reviewerID: A3OOAA8T697OEV
              asin: B00005GJFC
              reviewerName: Eugene Garriepy
              reviewText: This is my first experience with the product, and I found it easy to use. I would not hesitate to recomend it
              overall: 5.0
              summary: Easy to use
              reviewTime: 04 11, 2013

    Rank: 3 | Score: 0.273124635219574
              reviewerID: A26C1DPLV63R13
              asin: B00004TBK4
              reviewerName: S. Niawag
              reviewText: This item is easy to use. There is no use of batteries and does not have to be programed to function. It also shows the range that works well when setting temperature and humidity in the home.
              overall: 5.0
              summary: Quick view
              reviewTime: 01 10, 2013

    Rank: 4 | Score: 0.2841043770313263
              reviewerID: A37DAV2K8YP42B
              asin: B00004KSF9
              reviewerName: Brenda
              reviewText: Met our expectations. Very practical. Simple to use. Fits perfectly. So simple to use it is hard to write a review.
              overall: 5.0
              summary: works great
              reviewTime: 01 19, 2013

    Rank: 5 | Score: 0.2954506278038025
              reviewerID: A1JKGG1NABSSNM
              asin: B00004U9JO
              reviewerName: D. Beach
              reviewText: Was easy install, exactly like my old one.
              overall: 5.0
              summary: Five Stars
              reviewTime: 07 7, 2014

    Rank: 6 | Score: 0.2963576316833496
              reviewerID: A1ABYU81EE6EI2
              asin: B00004YUNO
              reviewerName: E.K.
              reviewText: very easy to work with and install. wires feel very secure and it comes with all hardware needed to install
              overall: 5.0
              summary: Good outlet
              reviewTime: 01 10, 2013

    Rank: 7 | Score: 0.3063773512840271
              reviewerID: A2CVB2DR8GNKUW
              asin: B00005AUHX
              reviewerName: Jennifer Williamson
              reviewText: Very easy to use.  I tested my water and right away knew if it was safe. I felt confident of the results.  This was a lot cheaper than a private water testing company.
              overall: 5.0
              summary: Good buy
              reviewTime: 10 10, 2013

    Rank: 8 | Score: 0.3087094426155090З
              reviewerID: A23ND9QAGTHIOE
              asin: B00005AUHX
              reviewerName: April Ebel
              reviewText: Product is just as it says. Directions are very helpful and easy to use. I would recommend for the price.
              overall: 5.0
              summary: Worked Great!
              reviewTime: 10 17, 2013

    Rank: 9 | Score: 0.3153085440731048б
              reviewerID: A2KNBECFPFE9D3
              asin: B00004U9JO
              reviewerName: SAL40er
              reviewText: Works like a charm
              overall: 5.0
              summary: Five Stars
              reviewTime: 07 5, 2014

    Rank: 10 | Score: 0.3173400759696960４
              reviewerID: A3TFLWSUF6XFCF
              asin: B00004U9JO
              reviewerName: quick
              reviewText: works great. easy to install because the instructions lay out the work in a way anyone can follow diagram by diagram
              overall: 5.0
              summary: no plummer required !
              reviewTime: 04 4, 2014

Processing Time: 0.3239 seconds
```

Figure xviii. Output from Milvus search

MongoDB

```
In [63]: import requests
import datetime
import time
import os
import openai
import json

from datetime import date
```

```
In [114… from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi
uri = "mongodb+srv://bonjoindia:ZmO0oFU6dWK7ZZP5@cluster0.qa6hq3a.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"
# Create a new client and connect to the server
mongo_client = MongoClient(uri, server_api=ServerApi('1'))
# Send a ping to confirm a successful connection
try:
    mongo_client.admin.command('ping')
    print("Pinged your deployment. You successfully connected to MongoDB!")
except Exception as e:
    print(e)
```

```
Pinged your deployment. You successfully connected to MongoDB!
```

```
In [67]: # GitHub Access Token

# Common API headers
headers = {
    "Accept": "application/vnd.github+json",

    "access_token": "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",
    "Git_Username":"anirbanbose83"

}
```

Figure xix. Code for Hybrid Query setup using MongoDB

```
from openai import OpenAI
client = OpenAI()
```

```
# Simple function that converts the texts to embeddings
def embed(texts):

    embeddings = client.embeddings.create(
        input=texts,
        model=OPENAI_ENGINE
    )

    return [x.embedding for x in embeddings.data]
```

```
def convert_datetime_to_unix_time(datetime_obj):
    date_format = datetime.datetime.strptime(datetime_obj, "%Y-%m-%dT%H:%M:%SZ")
    unix_time = datetime.datetime.timestamp(date_format)

    return int(unix_time)
```

```
appliances_reviews_df_new = appliances_reviews_df.head(1990)
```

```
appliances_reviews_df_new["embedding"] = appliances_reviews_df_new['reviewText'].apply(embed)
```

```
C:\Users\bonjo\AppData\Local\Temp\ipykernel_3576\3397934495.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  appliances_reviews_df_new["embedding"] = appliances_reviews_df_new['reviewText'].apply(embed)
```

```
appliances_reviews_df_new
```

```
appliances_reviews_df_new
```

	reviewerID	asin	reviewerName	helpful	reviewText	overall	summary	unixReviewTime	reviewTime	embedding
0	A27HOWZBUBJ8FF	0970408285	Steve	[0, 0]	Could have been longer though. well made and e...	4.0	Good fit	1387152000	12 16, 2013	[[-0.017633579671382904, -0.015432676300406456...
1	A24HQ894NFSTF5	7301113188	Maha Saqfalhait "shopaholic! ;)"	[0, 0]	I like these containers so much i have ordered...	5.0	I Love the Freezer storage line..	1236902400	03 13, 2009	[[-0.018233368173241615, -0.01791534572839737,...
2	AXE83MK90ZEVZ	B00002N7HY	Strom	[0, 0]	It works, no fires, etc. Why not 5 stars? Ho...	4.0	expectations achieved.	1389052800	01 7, 2014	[[0.002737861592322588, 0.006632857490330934 5,...
3	A2J7X7ZIH2EWB1	B00002NATH	NaN	[0, 0]	Fast shipping. Works great	5.0	Five Stars	1405814400	07 20, 2014	[[-0.02773250639438629, 0.016366302967071533, ...
4	AJQFNOFTZ7GOX	B00002NATH	Barthbill	[1, 1]	What can I say? It is the usual Leviton high q...	5.0	good product at a good price.	1277164800	06 22, 2010	[[0.014923588372766972, 0.018386593088507652, ...
...	...	...	...	...	...	...	...	...	...	...
1985	AN9WLTG3HFKCM	B00005O64S	Maybelle Miller "Gabooklover"	[3, 4]	I have wanted a dishwasher for a long time. Un...	5.0	It's Great!	1222646400	09 29, 2008	[[0.005219343584030867, -0.010897675529122353,...
1986	A15X59BE180N1K	B00005O64S	Media Enterprises, Inc.	[5, 5]	The unit started leaking and then quit functio...	1.0	LEAKED AND DIED	1265760000	02 10, 2010	[[-0.003302327822893858, 0.010290670208632946,...
1987	A13MQ135487L8Y	B00005O64S	M. Filer	[8, 8]	I received the washer right on time, no proble...	1.0	Not worth it	1182816000	06 26, 2007	[[-0.021759038791060448, -0.006805486511439085...
1988	A1RYWRK7L7VK28	B00005O64S	Michael A. Kartchner "kartvines"	[9, 9]	I have some concerns with this product. We hav...	3.0	Read all review before you buy	1131580800	11 10, 2005	[[0.0030940892174839973, -0.012539892457425594...
1989	A2P821GVM9SBXR	B00005O64S	Mike	[0, 1]	This works great for just my wife and me. A li...	4.0	Works Great	1207785600	04 10, 2008	[[-0.006603363435715437, 0.0221431702375412, 0...

1990 rows × 10 columns

```
# Ingest data into MongoDB
db = mongo_client['assignment_group']
collection = db['Amazon_reviews_group_project']
```

```
documents = appliances_reviews_df_new.to_dict('records')
collection.insert_many(documents)
```

Figure xx. Code for loading embedded data into MongoDB

```
import time

def search_mongodb_vector(query_text, top_k=10, num_candidates=100):
    # Generate the embedding for the query text
    query_embedding = embed([query_text])[0]  # Get the first (and only) embedding

    # Date range for filtering: January 2013 to August 2013
    start_date = datetime.datetime(2013, 1, 1)
    end_date = datetime.datetime(2024, 8, 31)


    # Start the timer
    start_time = time.time()

    # MongoDB vector search using the Lucene-based vector index
    results = collection.aggregate([
        {
            "$vectorSearch": {
                "index": "default",  # Use the appropriate search index name if different
                "queryVector": query_embedding,
                "path": "embedding",
                "limit": top_k,  # Limit the number of results to top_k
                "numCandidates": num_candidates  # Number of candidates considered in the search
            }
        },
        {
            "$project": {
                "reviewerID": 1,
                "asin": 1,
                "reviewerName": 1,
                "reviewText": 1,
                "overall": 1,
                "summary": 1,
                "reviewTime": 1,
                "score": {"$meta": "vectorSearchScore"}  # Include the similarity score
            }
        }
    ])

    filtered_results = []
```

Figure xxi. Code for search function in MongoDB

```
    filtered_results = []

    # Post-process filtering in Python
    for result in results:
        review_time_str = result['reviewTime']
        review_time = datetime.datetime.strptime(review_time_str, "%m %d, %Y")

        if start_date <= review_time <= end_date:
            filtered_results.append(result)

    # End the timer
    end_time = time.time()

    # Calculate the processing time
    processing_time = end_time - start_time

    # Print the filtered results
    for i, result in enumerate(filtered_results):
        print(f"Rank: {i + 1} | Similarity Score: {result['score']}")
        print(f"Reviewer ID: {result['reviewerID']}")
        print(f"ASIN: {result['asin']}")
        print(f"Reviewer Name: {result['reviewerName']}")
        print(f"Review Text: {result['reviewText']}")
        print(f"Overall Rating: {result['overall']}")
        print(f"Summary: {result['summary']}")
        print(f"Review Time: {result['reviewTime']}")
        print("\n")

    # Print the processing time
    print(f"Processing Time: {processing_time:.4f} seconds")
```

```
# Example usage
search_mongodb_vector("Easy to use", top_k=10, num_candidates=1500)
```

Figure xxii. Code for triggering search function in MongoDB

```
Rank: 1 | Similarity Score: 0.9554150184522705
Reviewer ID: A3PQAISWTRPV91
ASIN: B00004U9JO
Reviewer Name: Terry N.
Review Text: easy to install
Overall Rating: 5.0
Summary: Five Stars
Review Time: 07 10, 2014


Rank: 2 | Similarity Score: 0.9346740245819092
Reviewer ID: A3OOAABT697OEV
ASIN: B00005638FC
Reviewer Name: Eugene Garriepy
Review Text: This is my first experience with the product, and I found it easy to use. I would not hesitate to recomend it
Overall Rating: 5.0
Summary: Easy to use
Review Time: 04 11, 2013


Rank: 3 | Similarity Score: 0.9317740201950073
Reviewer ID: A26C1DPLV63R13
ASIN: B00004TBK4
Reviewer Name: S. Nieweg
Review Text: This item is easy to use. There is no use of batteries and does not have to be programed to function. It also shows the range that works well when setting temperature and humidity in the home.
Overall Rating: 5.0
Summary: Quick view
Review Time: 01 10, 2013


Rank: 4 | Similarity Score: 0.9289429187774658
Reviewer ID: A37DAV2K8YP42B
ASIN: B00004XSF9
Reviewer Name: Brenda
Review Text: Met our expectations. Very practical. Simple to use. Fits perfectly. So simple to use it is hard to write a review.
Overall Rating: 5.0
Summary: works great
Review Time: 01 19, 2013


Rank: 5 | Similarity Score: 0.9261595818757446
Reviewer ID: A1JKGG1NAB5SNM
ASIN: B00004U9JO
Reviewer Name: D. Beach
Review Text: Was easy install, exactly like my old one.
Overall Rating: 5.0
Summary: Five Stars
Review Time: 07 7, 2014


Rank: 6 | Similarity Score: 0.9255492687225342
Reviewer ID: A1ABYU01EE6EI2
ASIN: B00004YUNO
Reviewer Name: E.K.
Review Text: very easy to work with and install. wires feel very secure and it comes with all hardware needed to install
Overall Rating: 5.0
Summary: Good outlet
Review Time: 01 10, 2013


Rank: 7 | Similarity Score: 0.923469603061676
Reviewer ID: A2CVB2DR8GNKUW
ASIN: B00005AUHX
Reviewer Name: Jennifer Williamson
Review Text: Very easy to use.  I tested my water and right away knew if it was safe. I felt confident of the results.  This was a lot cheaper than a private water testing company.
Overall Rating: 5.0
Summary: Good buy
Review Time: 10 10, 2013


Rank: 8 | Similarity Score: 0.9227917194366455
Reviewer ID: A23ND9QAGTHIOE
ASIN: B00005AUHX
Reviewer Name: April Ebel
Review Text: Product is just as it says. Directions are very helpful and easy to use. I would recommend for the price.
Overall Rating: 5.0
Summary: Worked Great!
Review Time: 10 17, 2013

Processing Time: 0.6302 seconds
```

Figure xxiii. Output from MongoDB