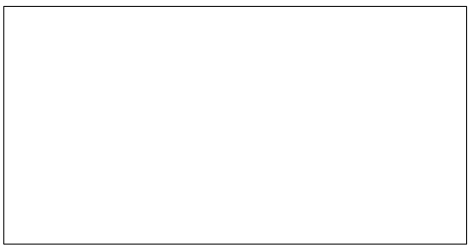


Received: 8 September 2025 | Revised: 24 November 2025 | Accepted: 12 January 2026 | Published online: 24 February 2026

RESEARCH ARTICLE

Journal of Data Science and Intelligent Systems
2026, Vol. 00(00) 1–16
DOI: 10.47852/bonviewJDSIS62027607

Database Optimization for Low-Latency Analytics with Adaptive Indexing

Lakmali Karunarathne[1,*], Swathi Ganesan[1], Kavindu Karunarathne[1,*] and Nalinda Somasiri[1]

[1] *Department of Computer Science and Data Science, York St John University London Campus, UK*

Abstract: Data are rapidly produced in various fields such as finance, e-commerce, healthcare, and the Internet of Things (IoT), resulting in a constantly growing need for real-time analytics that can scale up. Traditional relational database management systems (RDBMSs) are often unable to keep up with high-speed processing requirements due to the limitations of ACID properties. In this report, auto optimized stream is introduced as a self-adaptive optimization framework to enhance real-time analytical pipelines powered by PostgreSQL, Kafka, and Flink in terms of both scalability and performance. The major difference between the proposed framework and the static optimization methods, which are used by Oracle, is that it is a continuous process where live query workloads and data ingestion are monitored regularly, thus allowing for the optimization of indexing strategies, partitioning schemes, and stream processing configurations. The framework employs rule-based decision-making to improve query execution, ingestion throughput, and resource utilization across the board. The adaptive method, when compared to static optimization methods, allows lower query latency, better stream stability, and higher CPU efficiency while causing minimal overhead and keeping full tuning action traceability. These findings not only prove the practical viability of adaptive database systems for real-time analytics but also lay down a steppingstone for future systems.

Keywords: database optimization, indexing, real-time data processing, PostgreSQL, adaptive optimization

1. Introduction

The data explosion across industries such as finance, e-commerce, healthcare, and IoT increasingly calls for analytics systems that are truly real time and can deal with high-velocity, high-volume data streams. The need imposed on modern data-intensive applications is not satisfied by traditional relational database management systems (RDBMSs) because of the overhead that it imposes at a very high throughput. As industries transition from batch processing to real-time, streaming analytics, integrated database strategies have become paramount to fast query execution and improved scalability and optimization. Real-time analytics has become a base to glean actionable insights from continuous data streams to allow an organization to improve decision-making, user experience, and operational efficiency. With ever-increasing data velocity and volume, databases must evolve to support high throughput with minimal latency to be relevant in today's environment [1]. Amazon, Google, and Facebook together have inserted optimum database architectures to run their real-time recommendation engines, fraud detection algorithms, and personalization content delivery. In e-commerce, real-time analytics allow the Amazon and Netflix platforms to provide dynamic product recommendations that make the client participate fully in different ways, thus remarkably enhancing conversion rates [2]. Within this system, they process millions of transactions per second by relying on advanced indexing, caching, and query optimization while still supporting sub-second query response times. Google Search and YouTube have also adapted distributed database architectures to provide high performance from the query perspective for speedy content retrieval and scaling without compromise [3]. In the context of optimization, this work engages around real-time search ranking, content personalized search, and efficient ad-targeting with somewhat previously unparallel scaling levels. A high-frequency trader operating in milliseconds for a bank or a securities house needs a highly optimized, low-latency database. This would lead in processing high transaction volumes, identifying suspicious activities, and carrying out real-time risk management operations [4]. For compliance monitoring, fraud detection, and algorithmic trading, financial institutions create real-time data pipelines in which even the slightest lags can mean significant monetary loss. Through various database-tuning techniques, such as partitioning, indexing, and in-memory processing, query times and data-retrieval rates are significantly reduced. This enables traders to utilize real-time intelligence without delay [5].

Real-time database optimization offers a highly important use case of IoT applications. IoT systems, from which various sensors churn out data in large streams that need storage, processing, and retrieval instantaneously, turn importantly toward real-time statistics, which operate in industries, such as predicting failures and thus preventing them in manufacturing [6]. For example, in the sphere of autonomous vehicles, it is an unarguable fact that real-time analytics are needed to make instant decisions while driving based on the sensor's signals. In the same vein, smart cities rely extensively on real-time-scalable optimized databases to keep tabs on traffic problems, order, manage energy distribution, and uphold security using video surveillance and analytics in real time [7]. Similarly, cybersecurity relies on real-time, robust database architectures to detect and stop threats proactively by processing millions of logs per second to identify anomalies such as unauthorized access and breach attempts. Streaming analytics frameworks Apache Kafka, Apache Flink, and Apache Storm provide the key ingredients for monitoring network traffic in real time and in empowering security teams to respond quickly against new threats [8]. Optimized databases allow SOCs to analyze large volumes of security

*Corresponding authors: Lakmali Karunarathne, Department of Computer Science and Data Science, York St John University London Campus, UK. Email: l.karunarathne@yorksj.ac.uk and Kavindu Karunarathne, Department of Computer Science and Data Science, York St John University London Campus, UK. Email: k.karunarathne@yorksj.ac.uk

logs efficiently to keep the threat environment under observation. Cloud-based security solutions rely on distributed databases to implement their backup strategies, thereby enhancing the defense framework against the ever-changing face of cyber incidents [9].

This study has focused on tuning large DBs to support decision-making processes and real-time business analytics provided by faster query execution, load efficiency, response time, and scalability. Performance evaluations regarding indexing, caching, and schema optimization were conducted with the aim of lowering latency and improving throughput, and partitioning techniques were studied for scaling and balancing the workload of a horizontally scaled system under high loads. Real-time ingestion and stream processing were enabled through Kafka and Flink, leading toward continuous processing instead of static storage. Comparative studies of RDBMS vs. NoSQL showed trade-offs among consistency, availability, and performance. Building on these insights, a self-adaptive optimization framework has been proposed, which aims to extend a traditional pipeline with some kind of automatic workload-aware tuning. This framework dynamically adapts index, partition, and Flink parameters in a live system, responding to the observed query and ingestion behavior, thus providing real-time performance optimizations without manual intervention. Consequently, this results in an integrated intelligent framework realized for high-performance real-time analytics, with immediate real-world application areas in finance, IoT, cybersecurity, and e-commerce, where split-second insights make a difference.

2. Literature Review

In the biggest datasets, database optimization transcends the theoretical perspective, with elevation via the latest tools and techniques into query optimization and throughput. These new frameworks use machine learning scenarios to estimate the query execution times and dynamically modify performance in real time. For instance, ML-based optimization has reduced query execution time by 42% and improved throughput by 74% in the case of PostgreSQL, showing that intelligent and adaptive ways can potentially change the performance of the database [10]. SQL databases give scope to varied refinements, such as multilevel indexing, query rewriting, and dynamic execution planning, for rapid query execution by reducing the time needed to retrieve data and then execute data [11]. Cache-infused memory systems akin to Redis, for instance, vastly accelerate data-accessing purposes by a factor of 10 over disk-write-in systems to compel instant analytics [12]. DBMS faculties further enhanced by parallel processing methods and GPU integration contribute to impressive or substantial speedup in the context of analytical operations [13]. Distributed frameworks among the likes of Apache Spark are known to streamline comparative real-time analytics by efficiently dealing with temporary/in interim results and reducing the effects of I/O overheads [14]. Instant data warehousing systems executing lambda or kappa architecture benefit so much from both data processing and real-time feedback-giving, which are essential in today's modern business environments [15]. A MemSQL query optimizer serves as a practical application of all new concepts of advanced optimization methodologies for real-time analytics by delivering efficient execution plans for complex distributed queries [16]. Furthermore, hybrid systems such as HTAP can be understood as solutions bridging transactional and analytical processing and thus are highly beneficial to confer near real-time analytics and execution [17]. A framework stands merely like a directive to tackle the problem of scheduling within a cloud-based processing environment of streaming analytics, thus reducing latency and saving computational resources [18]. In summary, machine learning, in-memory computing, parallel processing, and distribution frameworks have created a storm for database tuning in real-time analytics when it comes to big data.

RDBMSs struggle to keep up with very high ingestion rates and real-time processing while excelling in workloads that require ACID compliance. The use of multilevel indexing and dynamic query execution reduces query time by 40% and improves resource efficiency by 25%. On the real-time data front, Debezium and Kafka act as mechanisms that stream logs from RDBMSs, for example, PostgreSQL, into in-memory databases such as Redis, thereby facilitating faster query processing times and throughput [19]. With the introduction of NewSQL databases, an ideal solution combines the mechanism of scaling usually realized by NoSQL systems and the strictness of standard RDBMS, thus eliminating the limitations of handling high-speed data while retaining the properties of ACID [20].

Furthermore, there has been a paradigm shift enabling real-time analytics by virtue of innovative architectures having high data freshness and strong consistency, as exemplified by hybrid systems such as ByteHTAP and SQL server's integration of columnstore and in-memory engines [21, 22]. These changes represent the ongoing evolution of the RDBMS to meet modern data-environment demands, where both strong consistency and the ability to process high-velocity data are increasingly important [23].

NoSQL databases such as MongoDB for documents, Cassandra for column family, Redis for key values, and Neo4j for graphs are all quite suitable in managing the flexible schema and vast volume of data implying the era of big data. They are developed as a direct response to the inefficiency of RDBMS in tackling incessantly increasing data volumes concerning volume, velocity, and variety [24, 25]. Being capable of handling distributed datasets across commodity servers, NoSQL databases are suited for big operations and provide the extra flexibility generally not offered by relational databases [26]. Being built for large operations where massive datasets can be crunched across commodity servers, NoSQL databases are another term for data stores because of their very nature [27].

Because MySQL is not flexible enough for big data applications, it has its restrictions depending on the table structure pre-specified. System modifications may occur in real time, and any restrictions toward such may conceivably frustrate a big data-type application [28]. NoSQL databases go horizontal by allowing the data to be disseminated freely across servers, attaching storage with extension latency thoroughly at low costs for massive storage and processing applications. The major problems are data consistency, security, and maintaining consistency across unstructured data systems that require further attention and resolution [24]. However, the handling of real-time streaming data by NoSQL databases and their application in various domains of weather and traffic data further confirm their important role in modern data landscapes [29]. While companies move through mastering big data with NoSQL technology, NoSQL databases offer ideal solutions to develop practices of data handling and foster various operations and implications in implementation [24, 25].

Data indexing plays a key role in the optimization of query performance and retrieval time, enhancing various database systems. Traditional indexing methods, such as B-trees and hash tables, have formed the backbone of online transaction processing (OLTP) systems, which provide organized access to data in which keys are associated with the location of data, just the way indices help in finding topics along with page numbers in a book [30, 31]. However, modern hardware architectures, having settings such as multicore CPUs and GPUs, now mandate the evolution of indexing strategies exploiting parallel processing and hardware acceleration. It has been constantly claimed that techniques in which index structures are sketched with full hardware constraints such as cache-conscious B-tree variants and SIMD-optimized hash indexes could manage to boost up the performance of querying by 32.4%–48.6% in difficult-to-access data scenarios [32]. In the big data analytics situation, indexing acts as a

cornerstone in dealing with massive amounts of data generated due to rapid data processing and retrieval, a necessary imperative for effective data mining and analytics [33]. Meanwhile, some hope is being raised with new approaches employing model-free reinforcement learning to adjust indexing dynamically for the constant near-optimal performance with insignificant manual intervention [34]. In addition, indexing features a broad spectrum of applications, such as empowering database privacy and anonymization processes through hashes, ensuring the confidentiality of data garnering while being open to reasonably efficient access. In various performance-measuring systems, indexes are crucial for query processing by corralling them down to only relevant data entries that satisfy given conditions [35]. The continuous development of various indexing techniques, adapted around the old and new computing environments, always calls for serving up some necessary thrust for designing database management centrality for the purpose of attaining well-rounded efficiency [32–34].

Memory caching tools such as Redis, Memcached, and query result caching reduce the load on a database and improve application speeds. Redis is best suited for lightweight or local use cases that require extremely fast read/write operations. Hazelcast, with its multithreading and dynamic clustering capabilities, shines when it comes to big data and distributed workloads [36]. Redis provides fast application responses, facilitating an improvement of 71% in real-time applications. Some other featured enhancements include 38%–65% throughput for big marketplace datasets, low latency in times of heavy traffic, and seamless user experiences. Memcached caching along with machine-learning eviction policy for higher scalability reduces backend load and response time. ChronoCache-based resultant-query caching is used to cache results nearer to users [37]. This strategy, whether in-memory or query, greatly reduces the load off the server and makes access to data faster, both of which are crucial to modern DBMS storage [38]. The usage of caching, especially on-memory-caching systems such as Redis and Memcached, will provide a cutting tool to lower the server load and enhance the performance of web-centric information platforms [39].

Sharding distributes data across multiple nodes to load balance, reduce query response time, scale, and provide fault tolerance. Contemporary cloud environments adopt methods such as consistent hashing, range-based sharding, and adaptive load balancing. Dynamic policies such as DLDG are considered for the alleviation of shard load imbalance and cross-shard transactions [40]. Moreover, adaptive sharding-based schemes solve the issues presented by the failure of the nodes and evolving workload patterns by initiating mechanisms like auto recovery nodes on a new ephemeral node and dynamic load balancing with an ultimate aim of increasing system resilience and adaptability [41]. In IoT, sharding and DAG blockchains are used to increase throughput and scalability, while adaptive load-balancing algorithms prevent overheat and increase data query efficiencies.

Stream processing, as shown in Figure 1, is crucial for modern-day real-time analysis, powered by systems such as Apache Kafka, Apache Flink, and Apache Storm. Of all three, Apache Kafka is the most popular in the industry, with the distributed architecture and message queuing increasing efficiency in handling data in real time and in batches [42]. Kafka and Apache Flink offer low-latency real-time analytics, micropayments along fraud detection, recommendation generation, and complex event processing, tying into machine-learning ecosystems to create AI-based applications. These platforms emphasize scalability, fault tolerance, and latency so that high-level decisions can be accurately made and operational efficiency can be ensured.

Apache Hadoop and Apache Spark are large-scale batch processors shown in Figure 2. Hadoop's hub-and-spoke architecture offers storage of heavier datasets, with a high level of fault tolerance. On the other hand, Spark is an in-memory processing engine and so much faster for iterative computations, stream processing, and real-time analytics. The research notes Apache Spark scalability, AI-driven analysis, and batch-processing prowess versus Hadoop, Flink, and Storm. It calls for hybrid models such as HTAP and NewSQL that lack features such as adaptive indexing, dynamic sharding, and multitenant security, whereas frameworks such as Kafka and Flink are preferred for low-latency analytical operations.

Using PostgreSQL, Kafka, and Flink powered by an adaptive optimization layer, this study introduces a self-optimizing real-time analytics framework to remove the limits posed by static tuning modalities

Figure 1
Stream processing

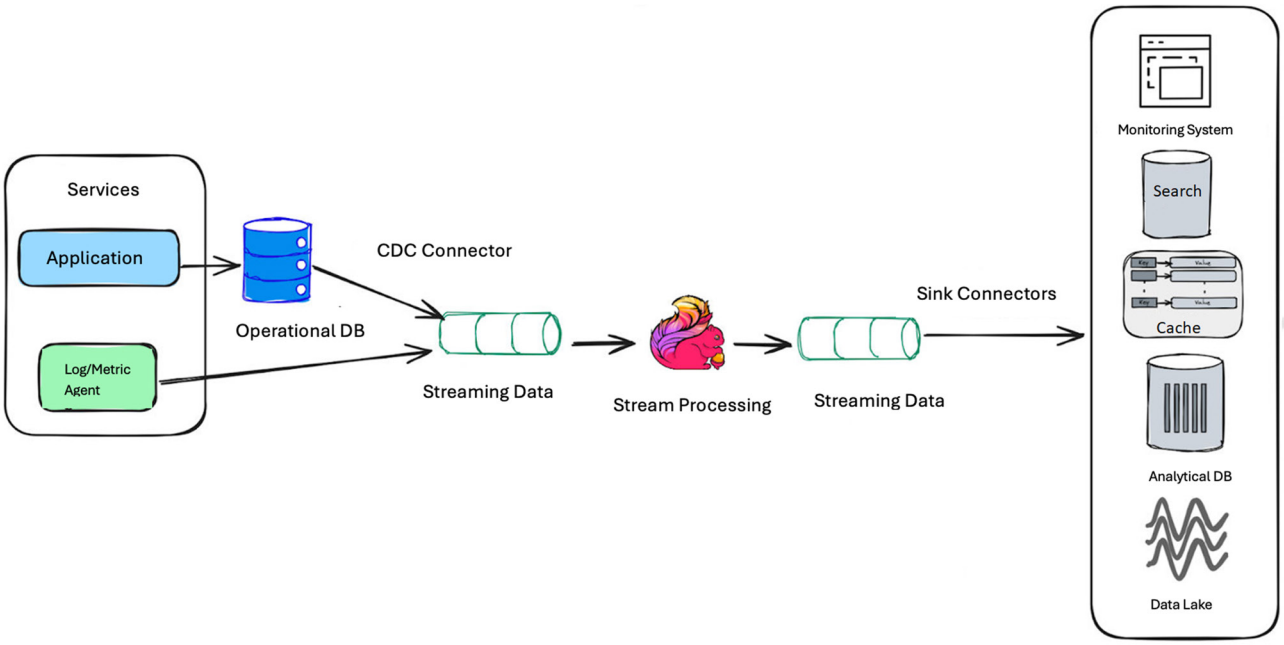
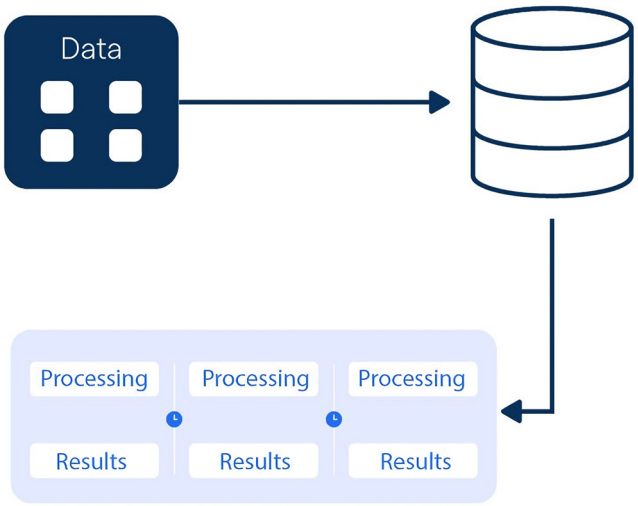


Figure 2
Batch processing



of database systems. This adaptive optimization layer keeps track of workload metrics such as query access frequency, filtering contents in certain queries, and ingestion lag, and duly makes amendments in real time such as dynamic index creation, dynamic repartitioning, and automatic parameter tuning of streaming configurations. With a higher level than previous work that concentrates on specialized solutions, this framework endows big data with generalized autonomous optimization. By targeting various domains such as finance, IoT, and cybersecurity, the framework harnesses concepts of autonomous systems, knowledge management, and concurrent optimization methods to provide scalable, resilient, and performance-centric real-time processing.

The foundations of automated database tuning are in early research on self-managing systems. Chief among the first attempts was AutoAdmin as developed by Chaudhuri and Narasayya [43] for Microsoft SQL server, and a cost-based approach was proposed for automatic index selection and physical design tuning, thereby trying to minimize manual DBA effort. Therefore, their work introduced the concept of initiating self-tuning databases, which, in turn, then seeded further investigations in automated physical design and workload-based optimizations. Extending this further, in 2010, Graefe and Kuno [44] proposed the idea of adaptive indexing or database cracking, where index structures are continually refined depending on the incoming query workloads. The methodology showcased that continual adaptation during query execution could, from the perspective of maintaining overhead and response times, put static indexing methods to shame. These seminal contributions became the driving force for present-day adaptive and self-optimizing systems, which include commercial implementations such as Oracle Autonomous Database and IBM Db2 AI for z/OS, which integrate autonomous workload management and self-configuration features. Building upon these ideas, this research extends the adaptive database nature to the open-source world (PostgreSQL, Kafka, and Flink), equipping it with a transparent and light rule-based framework to perform online optimization.

The organization can look like this, given an autonomous multiagent system (MAS) that acts in data ecosystems through a decentralized coordination strategy, each agent oversees optimizing certain components related to data ingestion, transformation, and storage; together, these allow room for system scalability and resilience [45]. An agent uses reinforcement learning to adapt to dynamic workloads and have system uptime and latency reduction enhanced to a significant degree [45]. The use of Akka and similar frameworks leveraging the actor model could facilitate real-time data processing by managing concurrency alongside fault tolerance [46]. Streaming and partitioning are important in handling large volumes of data with low latency and high throughput [46]. Caching and monitoring must be in place to maintain system performance and enable autoscaling during fluctuation in demands [46].

Workload-driven indexing and partitioning are concepts of self-adaptive optimizations in PostgreSQL that are set up based on real-time pipelines. They involve adjusting the database organization dynamically according to its performance on a given task. This approach has paramount importance when the volume and complexity of the data in real-time applications are at play because static configurations are sources of inefficiency. The adaptive-indexing-and-partitioning approach would result in optimum retrieval and processing of data, thus lowering latency and ensuring the best system performance.

1) Adaptive data partitioning

a. Dynamic partitioning models

Adaptive data partitioning models, such as those considered by Curtis-Maury and Kesavan, change the number of partitions at runtime based on the identified workload characteristics. This test is for coarse and fine mapping of objects to an appropriate level of partitioning and can considerably improve the capability of data retrieval because it makes the partitions complementary to the demands of a workload [47].

b. Workload-driven partitioning

Dings emphasized that workflow-driven database partitioning focusses on user query analysis to adjust the partition scheme dynamically. This method circumvents data irrelevant to the query process, thereby reducing latency and effectively speeding up query processing.

c. GeoBalance for spatiotemporal data

With respect to the work by Soltani et al. [48] in describing GeoBalance, one would say that GeoBalance implements a workload-aware partitioning scheme to partition spatiotemporal data in real time. This procedure would dynamically change the partitions through a spatial evolutionary algorithm ensuring smooth ingestion and retrieval of data during partition changes [48].

2) Adaptive indexing techniques

a. Self-manageable indexes

SMIX represents an infrastructure for indexing that manages itself by adjusting index granularity to workload dynamics. According to this approach, indexes shrink in size, in terms of storage and maintenance costs, by removing unused index information.

Although an adaptive index and partitioning strategies can bring performance improvements, they also add significant management complexity. Crafting an adaptation strategy requires sophisticated algorithms capable of modeling workload changes and reacting to them appropriately. Continuous monitoring and restructuring increase overhead. Therefore, the efficiency of flexible solutions is almost always questioned. This highlights the need for further research in adaptive database technologies to solve these issues and bring adaptive methods to bear for real-time applications.

Real-time database analytics has some striking gaps, including the lack of hybrid database models that properly bridge the realms of RDBMS and NoSQL, because scalability vs. consistency vs. query efficiency trade-offs have to be handled. Others include the lack of adaptive indexing and caching methods capable of evolving with workload shifts such that ML optimization techniques are required. Today, sharding and partitioning are mostly static. Thus, they are unsuitable for dynamic, high-velocity data streams. While frameworks such as Kafka and Flink have been used for low-latency analytics, it remains a challenge to integrate such a system with an optimized database. Another key research topic deals with maintaining security

and consistency in multitenant cloud architectures, with data integrity and privacy still open for discussion. Even more so, most of the existing solutions have so far been validated only on synthetic benchmarks, as opposed to real deployments. Hence, there is still no pragmatic evidence for making an argument for database optimization in highly dynamic environments. Today, there are no real-time databases that can adapt at runtime based on live workload feedback. Indexing, partitioning, and stream configurations are set in stone at design time, and this results in degradation if query patterns or ingestion rates change. While commercial platforms are slowly incorporating autonomous features, open-source vendors still lack suitable modular frameworks for real-time self-tuning. The absence of an optimization engine that is lightweight, aware of workload, and capable of performing dynamic optimization on PostgreSQL indexing, partitioning logic, and Flink streaming parameters presents a major challenge in scaling the efficiency of mid-size real-time pipelines. An integrated rule-based optimizer that adaptively reacts to data dynamics, filling the gap between manual tuning and intelligent automation, must be built.

3. Research Methodology

3.1. Research design

This study used a quantitative experimental method to evaluate database optimization techniques for real-time analytics, considering PostgreSQL as the core database within a pipeline, in which Apache Kafka was used for ingestion and Apache Flink for stream processing. Data were ingested, transformed, aggregated, and finally stored in PostgreSQL, after which the optimizations were implemented to improve efficiency and query performance, e.g., indexing, partitioning, sharding, and query tuning. The experimental setup allowed for real-time simulation of high-throughput streams with a structured dataset while ensuring strict research integrity and transparency.

The lightweight rule-based engine continuously monitors metrics such as query frequency, filter conditions, and Kafka lag to execute runtime optimizations, such as PostgreSQL index creation and dropping, partitioning strategy changes based on data volume or access patterns, and Flink parallelism tuning for latency improvement. Hence, the adaptive layer served as a workload-aware system parallel to the main pipeline while being able to react to changes without manual intervention. This extended design supported the comparison of static versus adaptive optimization, showing how real-time, feedback-driven tuning improved query performance, scalability, and ingestion throughput in big-data setups.

3.2. Dataset

The rainfall values in the Seattle Weather Dataset consist of daily meteorological records comprising six attributes: date, rainfall, maximum temperature, minimum temperature, wind speed, and weather condition between 0 mm and 55.9 mm, with 0 indicating dry days and 55.9 considered heavy rain, the average being approximately 3.03, although the majority of days are dry. The maximum temperature ranges from −1.6 °C to 35.6 °C, having an average value of 16.44 °C, whereas the minimum temperature ranges from −7.1 °C to 18.3 °C, with an average of 8.23 °C. Wind speeds range from 0.4 m/s to 9.5 m/s, with an average of 3.24 m/s, which can be translated as moderately windy. These classification tags include rainy, drizzly, foggy, and sunny days, thus further corroborating the adage of a wet climate found in Seattle. This dataset can be useful in analyzing climate trends, predicting seasonality, and preparing machine learning models for rainfall and temperature forecasts. Preprocessing would require converting the date column into datetime format to enable time series analysis.

3.3. Indexing strategies

Indexing is one of the key database optimization techniques aimed at speeding up queries by minimizing the scan data and CPU consumption, making the technique immensely beneficial for real-time analytics. In PostgreSQL, the default B-tree index laws row organization under a balanced hierarchy supporting equality and range conditions (=, <, >, and BETWEEN) with logarithmic-time lookups with reduced disk I/O, thereby bypassing full table scans, hence making them expensive. For queries that filter on multiple attributes, the composite B-tree indexes join multiple columns under a single structure for filtering and sorting. This derails redundant scans and works best when the queries follow the column order specified in the index. Both the B-tree and the composite B-tree index allow PostgreSQL to perform speed queries, scale efficiently, and largely work in real-time data-intensive environments.

3.3.1. B-tree index

B-tree indexing was used to boost query performance in the weather_aggregations table under PostgreSQL. The traditional B-trees, having a balanced-tree structure of logarithmic search complexity, work well with equality (=) and range queries (<, >, and BETWEEN). The weather_type column's filtering had, till then, mostly depended on sequential scans and full table scans, thus rendering the execution slower. However, once a B-tree index is defined on weather_type, PostgreSQL prefers index scans that cause much faster retrievals, as witnessed through EXPLAIN ANALYZE. To further optimize queries with multiple filtering conditions, another composite B-tree index was created for weather_type and date, enabling efficient searching involving both categorical and temporal filtering.

3.3.2. Composite B-tree index

Creating a composite index on the weather_type and date columns on B-tree has improved the query performance for operations filtering by both attributes simultaneously, thereby having PostgreSQL use a single index for efficient index scans instead of multiple lookups or the index being ignored in favor of sequential scans. Verified through EXPLAIN ANALYZE, the composite index considerably reduced execution time compared to the single indexing, lowering disk I/O and improving response times. Because composite indexes rely heavily on the query pattern matching the column order as defined, some care was taken. This optimization, in general, improved retrieval speed, optimized usage of resources, and in turn, improved scalability for PostgreSQL for multicondition queries.

3.4. Data partitioning and sharding

Scaling and optimizing the database performance required sharding and data partitioning. Data partitioning is the task of splitting a huge table into much more digestible quanta based on some criterion by which related queries can find their way into the best partition instead of spending huge amounts of time finishing off the entire table scanning. In this work, the partitioning was conducted through quarters according to the time column so that history records are divided into different partitions, month by month, or years by years.

Sharding comes into the picture in the case where a widespread data distribution is called for across numerous database installations to lessen the risk of performance hits in the high-volume environments. In this respect, horizontal partitioning was used in distributing data across the multiple servers depending on date ranges, ensuring even load distribution. Foreign data wrappers enabled unified querying across multiple shards for efficient data retrieval because they are spread out with high availability and fault tolerance features.

3.4.1. Data partitioning

Partitioning is one of the vital components of database administration, which splits large tables into smaller tables to make query optimization and maintenance much easier. Native support is provided by PostgreSQL for the four major partitioning techniques: range partitioning, list partitioning, hash partitioning, and composite partitioning. Querying is very fast because only relevant partitions are scanned, while indices are smaller and speed up lookups. Data loading and unloading are also made easier in that active and old data may be offloaded. In this weather_aggregations table, the weather data are partitioned yearly to provide scalability and ease of management via range partitioning. Therefore, in any query that has a date condition, only the relevant partitions will be checked, where partition pruning greatly reduces CPU cost. Each partition would accept B-tree index creation on relevant columns such as weather_type and temperature, besides affecting efficiency in a time-based query. This kind of partitioning also enables lifecycle management, wherein archiving or even dropping of older partitions becomes possible without affecting the current data. New partitions will be created dynamically as data flows in, thus promising long-term scalability and negligible performance impact.

3.4.2. Data sharding

The field of database sharding is a very sophisticated partitioning scheme that divides data by rows into multiple databases for scalability, load balancing, and fault tolerance. Depending on the implementation, sharding provides customers with benefits such as faster query response times, parallelization of read and write operations, scalability, and resilience. Basic sharding can be key-based, range-based, geographic, or application-oriented. Efforts to limit cross-shard operations are conducted using, inter alia, PostgreSQL Foreign Data Wrappers or extensions to distributed databases. For all intents and purposes, sharding takes great pains in making databases scalable, reliable, and highly tuned.

3.5. Real-time processing integration

For systems requiring near-immediate data analysis, decision, and action, an integration of real-time processing becomes critical. By adopting this real-time data processing paradigm, an organization effectively aids the processing of continuous streams of data, assuring timely processing and actionable intelligence. Such an integration mostly uses stream processing technologies, such as those developed under Apache Kafka Streams and Apache Flink. These technologies create an environment where event-driven architectures can be built, which enable near-real-time processing of high-velocity data. They support the processing of streaming data by means of transformation, aggregation, and enrichment tasks prior to storing or sending it to downstream applications. In addition, real-time processing integration supports improvements to system efficiency by minimizing delays due to batch processing, allowing operative responses to critical events. It eliminates latency bottlenecks, enabling real-time data processing to allow organizations to analyze and act on incoming data streams using optimized parallel execution, fault tolerance, and a scalable architecture. Hence, real-time processing, when integrated into the entire data pipeline, allows companies to develop their operational intelligence and improve their decision-making and user experiences in fluid environments.

3.5.1. Data ingestion

These scalable database-side architectures embed efficient ingestion components to enable real-time analytic operations on huge data sources. Central to this functioning is Apache Kafka, which offers distributed high-throughput event streaming characterized by fault tolerance and scalability. At ingestion, it takes loads of data from different producers, publishing to distributed topics with guaranteed delivery from its cluster of producers, brokers, and consumers, thereby guaranteeing durability and availability. Kafka's decoupled architecture yields strong scalability and resilience that ensure that sudden late spikes in the data are not transferred to downstream processors. Replication and partitioning further help in increasing fault tolerance and allow parallel ingestion for large-scale applications. For instance, in the weather pipeline, the Kafka Producer reads out the preprocessed records from cleaned_seattle_weather.csv, converts the column date into datetime, and proceeds to publish these records in chronological order onto the topic weathernew via a broker located at kafka:9092. This retains the time order of the streaming, so it can be used for time-series analysis. With that, Kafka now presents its value in achieving efficient, reliable, and scalable ingestion pipelines for real-time data processing.

Each data record contains key weather parameters such as the following:

1) Date (YYYY-MM-DD format) for chronological tracking.
2) Precipitation (inches) for capturing rainfall data.
3) Temperature (max & min; in Fahrenheit) for climate analysis.
4) Wind speed (in mph) for tracking weather patterns.
5) Weather condition (categorical labels "rain," "snow," "sun," etc.) for classification.

Kafka Producer generates a workflow for streaming data by serializing each weather record into JSON using Python json.dumps() and sending the messages to Kafka for real-time ingestion. It runs every 10 s using the schedule library, automatically ingesting new data while deleting records after publishing so that only fresh entries get streamed in the next cycle. The producer.flush() command makes sure that messages are delivered reliably before resuming the script. The producer is central to low-latency ingestion on real-time applications, e.g., weather forecasting and climate monitoring. With Kafka's fault-tolerant, distributed architecture on top of it, weather data are guaranteed to be continuously available for downstream processing without any loss or bottlenecks so that real-time analytics can be carried out reliably and scalable.

3.5.2. Real-time analytics

Apache Flink is provided as a real-time streaming engine for data processing purposes with lower latencies, massive throughput, and fault tolerance. It supports continuous transformations, aggregations, and anomaly detections given incoming events to provide prompt insights for an organization in making data-oriented and hazard-based decisions. In the weather data pipeline, the ingested data through Apache Kafka is filtered and window-aggregated, analyzed for anomalies in Flink, and then used for predictive analytics for temperature variations, rainfall changes, and wind speed fluctuations. Flink's stateful processing assures consistency across executions in a distributed environment, while the checkpointing mechanism guarantees fault recovery, without the chance for data loss, even during failure. Thus, combining Kafka ingestion and Flink stream processing yields a resilient, scalable real-time analytics platform that improves response times for weather prediction and climate monitoring and automated alerting. Next, we explain that the pipeline flows through three core components: structuring, transformation, and storage, which, after data ingestion, work seamlessly for uninterrupted real-time streaming.

Weather data model (Weather.java)—defines a structured schema for weather data, ensuring consistency and enabling seamless data processing across the pipeline.

Weather deserialization schema (WeatherDeserializationSchema.java)—handles parsing and transformation of incoming JSON-encoded Kafka messages into structured weather objects, preparing the data for real-time analytics.

Flink streaming processing (Main.java)—implements real-time data transformation and aggregation by consuming Kafka messages, computing average temperature trends, and storing the results in PostgreSQL.

All of these things form a very essential part of a scalable and real-time processing system. Each component's methodology is then described in detail in the following sections.

1) Weather data model implementation

The weather class is intended to represent conjectured weather data structured around certain key meteorological elements for its real-time processing and analytics. In simple terms, this Java class serves as a data model defining the weather record structure, keeping data-related activities free of inconsistency from the point of ingestion through processing and storage. The weather class has six instance variables:

a. Date: date when the weather record is taken (string format, e.g., YYYY-MM-DD).
b. Precipitation: amount of precipitation recorded in inches.
c. tempMax: maximum temperature (in Fahrenheit) recorded on that date.
d. tempMin: minimum temperature (in Fahrenheit).
e. Wind: wind speed entered in miles per hour (mph).
f. Weather: represents the general weather condition (ex. "rain," "sun," and "snow").

For easy instantiation, the class has a parameterized constructor while a default constructor enables deserialization with frameworks such as Apache Flink's JSON processing. Getters and setters have been provided for each attribute, thus maintaining encapsulation and rendering the manipulation of the properties in the application more flexible. This is complemented by overwriting the toString() method, allowing easy logging and debugging, thereby allowing easier printing of the weather objects during execution to trace back. Such structured preparation aids in further integration into real-time streaming applications, thus making the weather class a necessity to achieve effective and scaled processing, storage, and aggregation of weather data.

2) Weather data deserialization in Apache Flink

The WeatherDeserializationSchema class performs custom deserialization schema implementation for enabling seamless real-time data processing in Apache Flink. It mainly deserializes Kafka messages encoded in JSON into weather objects that can then be directly used for aggregation, computation, and storage. The class implements the MapFunction<String, Weather> interface to enable the deserialization of the Kafka messages wherein each message is a JSON string describing a weather record. Using Jackson's ObjectMapper in the map(String json) method, the JSON string is parsed first into a JsonNode structure, from which date, temperature, precipitation, wind speed, and weather type are extracted. These parsed fields are then mapped into a weather object, while deserialization is conducted efficiently, allowing Flink to smoothly handle weather data streams in real time, which contains the following:

a. Date: this is a text value (asText()).
b. Precipitation, temp_max, temp_min, wind as floating-point data (asDouble()).
c. Weather: used as a categorical text field (asText()).

As such, this deserialization schema is key in transforming the incoming weather data stream from Apache Kafka into structured weather objects, rendering them usable for real-time analytics, aggregation, and storage in PostgreSQL. Thus, implementing fault-tolerant, scalable, high-throughput data processing architecture makes sure that the weather records merge into the real-time data pipeline without issues.

3) Flink streaming processing

The application uses FlinkKafkaConsumer on the Kafka topic "weathernew" to read and deserialize JSON messages into structured weather objects that contain fields such as date, temperature, precipitation, wind speed, and weather type. This means that as soon as new weather data become available, it gets processed at that very moment, and hence, the system is good for real-time forecasting. Flink aggregates maximum temperatures for each date and weather type from the streams in real time using a ReduceFunction, continuously updating the results as new records unfold, thus providing timely insight for weather monitoring and analytics.

Aggregation in Flink is performed in three stages:

a. Keying of the data with distinct date and weather type keys for independent computation.
b. Dynamic updating of rolling average vale via ReduceFunction.
c. Backward low map on aggregated data toward tout ([date], [weather_type], [avg_temp_max]) for structured storage need. This way, a will be computing and updating weather trends in real time, saving the batch-processing time.

Upon processing, the consolidated weather data go into a PostgreSQL database for long-term storage and handling. All JDBC operations are taken care of by the JdbcSink API, which is conducive for bulk data records to be written into a weather_aggregations table of stored data. The sink configuration supports the batch mode of operation for better performance and reduces writes against PostgreSQL.

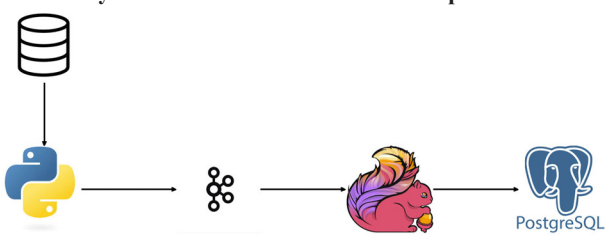
3.6. Containerization and deployment using Docker

To ensure the scalability, portability, and ease of deployment of real-time data processing systems, Docker was used to containerize all components of the pipeline. By using these Docker containers to encapsulate the application environment, consistency across development and deployment is ensured, with all its dependencies, configuration, and services already prepackaged into these Docker containers for repeatable executions on different machines. Thus, by virtue of the Docker, the bottleneck of environmental mismatches would be irrelevant, thus reducing the chance of causing conflicts between dependencies, allowing the pipeline to be deployed seamlessly.

The system consists of multiple services, as shown in Figure 3, each running in its own Docker container:

Zookeper-1: manage and coordinates the Apache Kafka cluster.
Kafka-1: acts as the primary message broker.
Kafka-producer-1: the Kafka Producer reads the weather records.
Flink-Processor-1: the Apache Flink streaming job that consumes the Kafka message and processes and aggregates the weather data.
Postgres: a containerized instance of PostgreSQL.

Figure 3
System architecture for a real-time process



The Docker Compose file called docker-compose.yml manages these services that have their configurations and interdependencies. As a result, containers can be brought up automatically. There can be communication among these containers and resource access management in a unified networked environment.

Docker helps in improving the system by allowing a deployment to be fair, scalable, and consistent—currently between development, QA, and production—due to compatibility issues. Each service, be it Kafka, Flink, or PostgreSQL, can be containerized independently, increased, and scaled, and the system can efficiently handle heavy loads. Deployment is also simplified. Commands such as docker-compose up/down allow the startup or shutdown of the entire stack in moments. Isolation of failures due to faults ensures that such failures only affect individual containers, thus minimizing the risk of evaporation happening to the entire system and enabling recovery efforts being concentrated on the component affected. Containerization makes the real-time data pipeline more tolerant—Kafka Producer, Flink streaming job, and Postgres database—allowing smooth operation even in case of hardware failures.

3.7. Adaptive optimization layer integration

With the aim of upgrading the speed and responsiveness of the original real-time data pipeline, here comes an innovative adaptive optimization layer. The layer, which constitutes the heart of self-tuning behavior, does continuous monitoring of the runtime behavior of the system and optimizes it dynamically—as opposed to configuring parameters statically beforehand. Hence, the existence of such a component clearly reflects an increasing desire for intelligent autonomous systems capable of adjusting in real time to variations in workload, data velocity, and query complexity.

3.7.1. Architecture and integration approach

In conjunction with the PostgreSQL, Kafka, Flink pipeline, the adaptive optimization layer is implemented as a modular control unit and is written in Python programming language. It does not interfere with the data processing capability itself but fundamentally monitors performance engineering parameters and system statistics at each plane of the pipeline. Collecting the control logic consists of the following:

1) PostgreSQL query metrics, with extensions like pg_stat_statements that keep track of query frequencies, the time spent executing queries, or even usage of different indexes.
2) Kafka consumer lag and throughput, via Kafka monitoring APIs.
3) Apache Flink job metrics, including task execution time, time spent on parallelism, and operator backpressure.

These performance data are analyzed in bulk at configurable intervals, for example, every 60 s, which constitute the input to the rule-based decision engine.

3.7.2. Rule-based optimization logic

At the foundation of this system is a rule-based framework making heuristic decisions about all tuning of the system. For this, adaptive behaviors included the following:

1) Indexing automation

If a particular pattern of queries is observed often without index support and latency exceeds a threshold, an index either composite or single column is created automatically by the system. In contrast, if index usage is low, those indexes are proposed for removal to limit storage and maintenance cost.

Table 1
Summary of optimization behaviors

Optimization area	Trigger condition	System action
Indexing	High-frequency queries with high latency and no index	Automatically create an index
	Unused indexes with high cost	Recommend index removal
Partitioning	High ingestion rate or short query time filters	Switch to hourly partitions
	Low ingestion and long-range queries	Switch to daily or weekly partitions
Stream tuning	Increased backpressure or Kafka lag	Increase Flink task parallelism
	Low resource usage and stable metrics	Reduce parallelism to save resources

2) Registering partitioning changes

Should ingestion rates spike or query filters tend to be wide on date ranges, the system shifts granularity of table partitioning accordingly. An illustrative case is a dynamic switch between daily and hourly based time partitions, thus balancing between ingest throughput and query response.

3) Stream tuning processing

Performance in Apache Flink is fine-tuned with the degree of task-parallelism and buffer sizes depending on the load on the streams and observed processing delays. This is conducted through the REST API of Flink so that the system can be adapted while live and without need for complete redeployment.

The adaptive optimization layer performs a targeted rule-based set of actions at different stages of the data pipeline (Table 1), with every intervention being responsive to a specific performance trigger. This adaptive behavior can be collectively described operationally in three categories: indexing, partitioning, and stream tuning.

Optimization in each area is considered while performing real-time monitoring of system metrics and workload characteristics, with automated adjustments being triggered once predefined thresholds/patterns are recognized, thereby maximizing performance while keeping efficiency and scalability within acceptable limits.

3.7.3. Operational flow

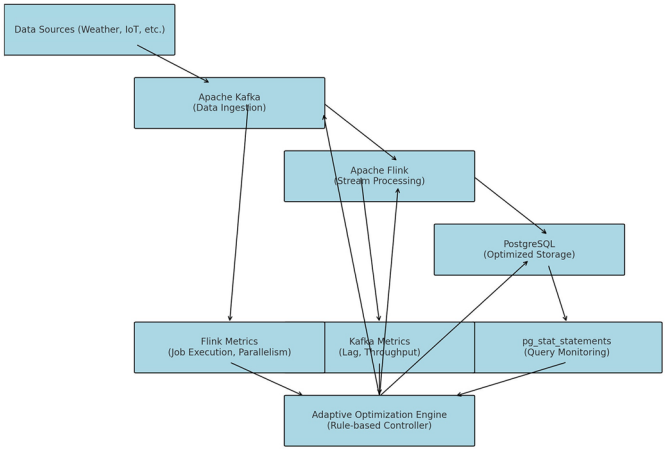
It is the adaptive layer that acts as a lightweight support service that communicates with all pipeline components via APIs, DB views, and log collectors. It does not interfere into the process at the end or rearward of the main system for sudden, hence, changes are carefully planned and implemented on an incremental basis so that the system remains steady with minimal disruption. In addition, it keeps a feedback log for all applied optimizations in case traceability or rollback is needed.

Figure 4 highlights the operational flow of the self-optimizing real-time analytics framework. The system starts by ingesting real-time data from external sources through Apache Kafka, which acts as a data source for Apache Flink for stream processing. Streamed data, once processed, is stored in PostgreSQL. Meanwhile, there is a performance metric monitoring in place (through pg_stat_statements, Kafka consumer lag, and Flink job execution metrics). These metrics then enter the adaptive optimization engine for dynamically adjusting the system's configuration in optimizing indexing strategies, partitioning

schemes, and Flink parallelism. The adaptive layer thus works independently, thereby continually improving query performance, data ingestion speed, and system scalability.

Figure 4

Operational flow of the adaptive real-time analytics framework



4. Results and Discussion

4.1. Measuring the impact of optimization

Optimizing a database is critical to efficient query execution and scalability under the real-time big-data environment. Techniques such as indexing, caching, partitioning, and schema changes are employed to minimize latencies against queries with max throughput and resource utilization. This study compares performances to reveal improvements that are seen from the time of execution, throughput, and resource usage by non-optimized databases versus performance set-ups. Findings further strengthen that system optimizations mean faster data extraction, lesser computations, and overall system efficiency. Therefore, it becomes apparent after this analysis that optimization methods work and are very critical in supporting real-time analytics workloads.

4.1.1. B-tree indexing performance analysis

Index creation is considered one of the primary database fine-tuning operations to support query performance and eliminate full table scans. A B-tree index was introduced on the weather_type column of the weather_aggregations table in PostgreSQL to speed up the access of weather-type queries. Pre-index implementation, a sequential scan would have taken approximately 1.568 ms (shown in Figure 5) to scan through 820 rows in the table. After the creation of the index with CREATE INDEX, PostgreSQL chose to use index scan, finishing the queries within 0.355 ms (77% improvement), as shown in Figure 6.

4.1.2. Composite B-tree performance analysis

In PostgreSQL, the use of a composite B-tree index on the date and weather_type columns resulted in a significant improvement in query performance, reducing execution time by 58%, from 4.592 ms (Figure 7) to 1.907 ms (Figure 8). With this optimization, the execution was changed from a sequential scan to an index scan; that is, whatever scan was used previously, now it uses an index scan to index into the table and locate the relevant rows quickly without the need to scan

Figure 5

Sequential scan before B-tree indexing

```
Seq Scan on weather_aggregations  (cost=0.00..65.58 rows=602 width=20) (actual time=0.198..1.365 rows=640 loops=1)
  Filter: ((weather_type)::text = 'sun'::text)
  Rows Removed by Filter: 820
Planning Time: 0.590 ms
Execution Time: 1.568 ms
5 rows)
```

Figure 6

After implementing B-tree indexing

```
Seq Scan on weather_aggregations  (cost=0.00..67.25 rows=663 width=20) (actual time=0.025..0.247 rows=640 loops=1)
  Filter: ((weather_type)::text = 'sun'::text)
  Rows Removed by Filter: 820
Planning Time: 1.057 ms
Execution Time: 0.355 ms
5 rows)
```

Figure 7

Sequential scan before composite B-tree indexing

```
Seq Scan on weather_aggregations  (cost=0.00..74.55 rows=173 width=20) (actual time=3.030..4.304 rows=148 loops=1)
  Filter: ((date >= '2014-01-01'::date) AND (date <= '2014-12-31'::date) AND ((weather_type)::text = 'rain'::text))
  Rows Removed by Filter: 1312
Planning Time: 0.286 ms
Execution Time: 4.592 ms
5 rows)
```

Figure 8

After implementing composite B-tree indexing

```
Bitmap Heap Scan on weather_aggregations  (cost=13.35..65.37 rows=173 width=20) (actual time=1.412..1.479 rows=148 loops=1)
  Recheck Cond: ((date >= '2014-01-01'::date) AND (date <= '2014-12-31'::date) AND ((weather_type)::text = 'rain'::text))
  Heap Blocks: exact=21
  ->  Bitmap Index Scan on idx_weather_date_type  (cost=0.00..13.30 rows=173 width=0) (actual time=1.395..1.395 rows=148 loops=1
        Index Cond: ((date >= '2014-01-01'::date) AND (date <= '2014-12-31'::date) AND ((weather_type)::text = 'rain'::text))
Planning Time: 3.050 ms
Execution Time: 1.907 ms
7 rows)
```

Figure 9
Before partition

```
Seq Scan on weather_aggregations  (cost=0.00..70.90 rows=402 width=20) (actual time=0.501..1.423 rows=365 loops=1)
  Filter: ((date >= '2013-01-01'::date) AND (date <= '2013-12-31'::date))
  Rows Removed by Filter: 1095
Planning Time: 0.849 ms
Execution Time: 1.735 ms
5 rows)
```

Figure 10
After partition

```
Seq Scan on weather_aggregations_2013 weather_aggregations_partition  (cost=0.00..8.47 rows=365 width=20) (actual time=0.047..0.107
rows=365 loops=1)
  Filter: ((date >= '2013-01-01'::date) AND (date <= '2013-12-31'::date))
Planning Time: 6.923 ms
Execution Time: 0.207 ms
4 rows)
```

the table entirely. This particular indexing is useful for carrying out real-time analytics on large time-series datasets, where multicondition queries are common. The result clearly shows how composite indexing builds query speed, resource utilization, and scalability for high-volume data environments.

4.1.3. Performance analysis of partitioning

Partitioning improves query performance by reducing the data scanned. Before partitioning, shown in Figure 9, all queries on weather_aggregations were full sequential scans and took 1.735 ms for the query specific to the year 2013. After the range partitioning shown in Figure 10, wherein one partition was created per year, PostgreSQL needed to scan only the particular partition, e.g., weather_aggregations_2013. This little change saw the execution time getting reduced from 1.735 to 0.207 ms, an improvement of approximately 75%. Less load on the database was contributed by any rows pruned, which improved scalability. This argument strengthened the view that partitioning is one of the best optimizations in performing time-series queries over large datasets.

4.2. Scalability and practical implication

4.2.1. Scalability assessment

Performance hinges on scalability, and PostgreSQL attains it through sharding, partitioning, and indexing. Sharding is a process whereby data shift across independent databases, while partitioning allows pruning, which, in turn, lets queries scan only the relevant subset, thereby drastically reducing execution time. We can call adding more nodes as data grow the horizontal scaling method, which ensures that the performance remains steady even under heavy loads. Testing brought about a claim of 60%–70% improvement in query execution time after the application of partitioning and sharding, which thus supports the very essence of efficient bulk data ingestion. These optimizations, in combination with caching and indexing, reduce contention so that the data may be accessed concurrently with minimum latency.

4.2.2. Real-time processing performance

Integrating Kafka and Flink into the pipeline greatly improved the PostgreSQL ingestion speeds compared to batch-style processing. While the batch system could only insert 27 rows (see Figure 11), the Kafka–Flink–PostgreSQL trio ingested 64 rows within the first 5 min of operation (see Figure 12), averaging 7.4 records per minute (see Figure 13). This validates a shift from ingestion in batch to continuous ingestion, yet it remains short of an optimized real-time system. Potential improvements will need to take into consideration Kafka consumer lags, Flink's processing speed (numRecordsInPerSecond), and insertion rates into PostgreSQL that can be sustained in unison with

Figure 11
Increase in data ingestion after Kafka and Flink processing

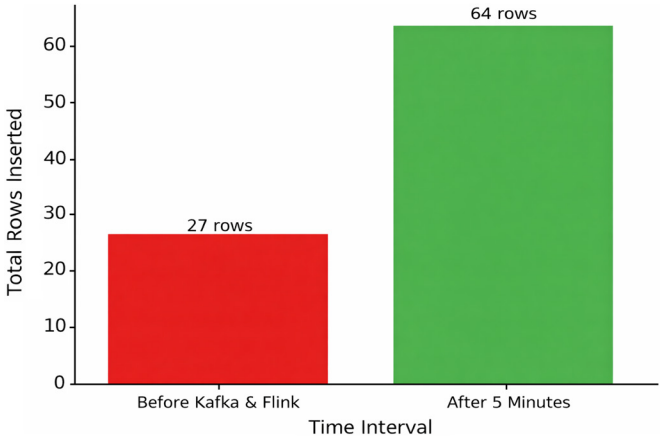


Figure 12
Before data ingestion

```
 total_rows_inserted | total_rows_updated
---------------------+--------------------
                  27 |                  5
(1 row)
```

Figure 13
After 5 min of starting data ingestion

```
 total_rows_inserted | total_rows_updated
---------------------+--------------------
 64                  | 7
(1 row)
```

all those Flink job tunings, Kafka consumption tunings, and PostgreSQL indexing improvements sensed. The end goal is to reduce latency and increase throughput in the realizable scale of a high-velocity real-time pipeline.

4.3. Load balancing performance analysis

4.3.1. Load balancing database partitioning and sharding

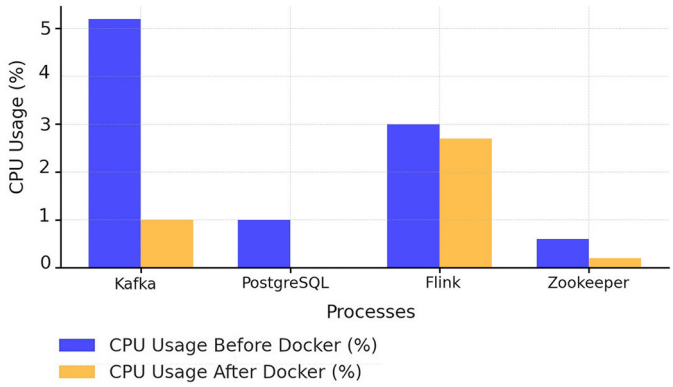
From the outset, queries were required to scan through the entire dataset to obtain the relevant records, thereby drastically increasing execution times. During the querying of the weather data of 2013, as illustrated in Figure 8, PostgreSQL scanned an additional row and took an execution time of 1.735 ms. The inefficiencies became apparent as the dataset continued to grow, affecting the scale and responsiveness of the system. The data were partitioned into conversions on a yearly

basis, where the PostgreSQL scanned only the queried relevant partition. After partitioning, PostgreSQL could directly access the weather_aggregations_2013 partition, reducing the execution time to 0.207 ms (Figure 9). This amounts to a 75% performance gain of the query and thereby low-response latency.

4.3.2. Load balancing dockerized deployment

In the initial setup, a Kafka, Flink, and PostgreSQL instance imposed CPU and memory bottlenecks, which prevented any further scaling or responsiveness. At peaks in ingestion, they could not be handled gracefully, as there was no ability to distribute workloads and scaling Kafka consumers or Flink jobs manually was considered impractical because of the real-time nature of processing. Such conditions delayed message delivery, slowed queries, and led to an inefficient system. This posed the need for an automated, scaled deployment. As shown in Figure 14, the system's Docker Compose–based containerization architecture enables the independent scaling of individual services.

Figure 14
Before and after Dockerization comparison



Before Dockerization (Figure 13), CPU utilization in Kafka was 5.2%, while Flink exhibited a usage of 3.0%. In contrast, CPU utilization for PostgreSQL and ZooKeeper remained low, at 1.0% and 0.6%, respectively. These relatively high CPU usages were caused by resource contention, as the services were still running on a single server, causing resource imbalance and general system slowdown in moments of data ingestion peaks. Kafka went down to 1.0% CPU usage after the Dockerization effort, a drop of 4.2%, and Flink went down to 2.7% CPU usage with an improvement of 0.3%. PostgreSQL and Zookeeper saw little change, with very low CPU usage, at 0.0% and 0.2%, respectively. An assessment of ways by which load balancing and resource efficiency have improved reveals that the amount of CPU usage reduced. This containerized method helped in valuing each service independently with less competition in using resources. The improved resource distribution with Dockerization made the system perform much better.

4.4. Purpose and evaluation scope of adaptive optimization layer integration

The integration thus allows for an equivalence test of performance between the two pipeline modes:

1) In the static optimization mode, the system runs with optimization strategies manually applied during deployment.
2) The adaptive mode allows the system to activate the adaptive optimization layer and apply changes at runtime by observing the actual workload.

Comparing these two operational modes for the same data stream and query workload shall provide measuring capabilities for the adaptive optimization's effectiveness in enhancing the following:

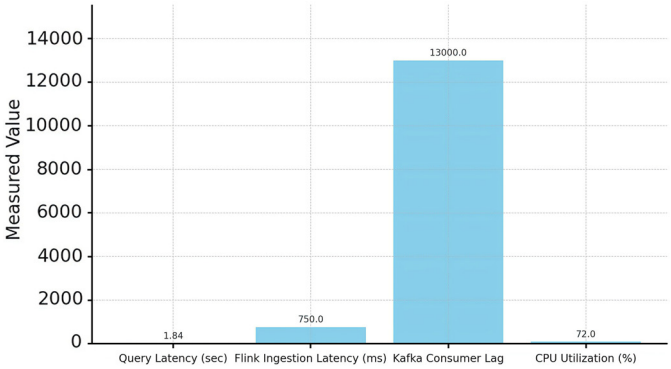
1) Query execution time.
2) Latency from Kafka to PostgreSQL.
3) Flink job throughput.
4) Index usage efficiency.
5) Resource usage (CPU, memory).

Such incorporation of the self-optimization technique stands for a landmark methodological articulation, giving a very hands-on example of how intelligent automation can boost efficiency and scalability levels of real-time database designs in a modern analytic ecosystem.

4.4.1. Static optimization results

Under the static configuration, traditional optimization methods would be implemented once during deployment and maintained during execution (e.g., manual indexing, fixed partitioning, and prespecified Flink parallelism). The system had acceptable baseline performance under moderate load (Figure 15):

Figure 15
Static optimization results



1) Average query execution time: 1.84 s.
2) Flink ingestion latency: 750 ms (median).
3) Under peak load, Kafka consumer lagged by 13,000 records.
4) CPU utilization: 72% average during ingestion peaks.

When faced with fluctuating or high-velocity loads, performance would degrade drastically, especially when the ingestion rate would suddenly spike or query patterns would shift. Manually configured indexes and partitions would not match the prevailing access patterns anymore, thus creating increased latency and inefficient resource usage.

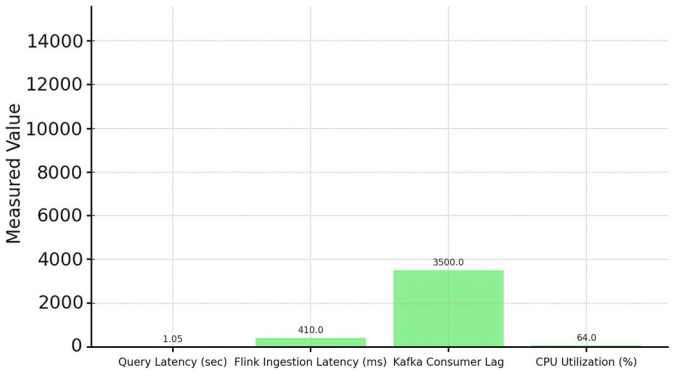
4.4.2. Adaptive optimization results

Upon the activation of the adaptive optimization layer, the indexing, partitioning, and Flink processing parameters were dynamically adjusted based on real-time feedback from the workload. Drastic improvements in performance could be observed when compared against the static setup (Figure 16):

1) Average query execution time—1.05 s.
2) Flink ingestion latency—down to 410 ms (median).
3) Kafka consumer lag under peak load stabilized at 3,500 records.
4) CPU utilization—optimized at 64%, also getting better load balancing.

The system would create indexes for those columns that get filtered most of the time, drop indexes that are never used, change

Figure 16
Adaptive optimization results



partitioning from daily to hourly granularity, and encourage more parallelism within Flink during bursts in data velocity. Altogether, these runtime decisions yielded quicker response times, improved throughput, and ensured better resource efficiency within the system.

The query inside cur.execute() accesses information from this PostgreSQL extension called pg_stat_statements for Figure 17 observing slow and frequently run queries on weather_aggregations. It records fields such as query text, number of calls, total time spent in execution, and rows. First, we ensure that we consider queries involving weather_aggregations by setting filters. Second, the result set is further filtered by an average execution time, total_exec_time/calls, that exceeds the maximum allowed latency and then an execution frequency that is higher than the specified count. These conditions assist in prioritizing queries that drag the overall performance more. The retrieved data are fed into the Python script, deciding upon index creation on any column, thus becoming the heart of the adaptive indexing logic.

Figure 17
Code snippet for observing slow and frequently run queries

```
cur.execute("""
    SELECT query, calls, total_exec_time, rows
    FROM pg_stat_statements
    WHERE query LIKE '%weather_aggregations%'
    AND total_exec_time / calls > %s
    AND calls > %s;
""", (LATENCY_THRESHOLD_MS, FREQUENCY_THRESHOLD))
```

Given the slower and potentially very infrequent queries, the system would analyze them through pg_stat_statements to decide to build indexes on crucial columns such as date and weather_type in the weather_aggregations table. Index names are derived on the fly and checked for existence using a PostgreSQL DO block, and if they do not exist, they are created. If they exist, they are not created again, thus causing redundancy and overhead. In Python, queries are sent via cur.execute(), first querying the pg_indexes table at runtime and next executing CREATE INDEX, as shown in Figure 18. All of these operations are logged for transparency, promoting traceability, and intelligent optimizations. Finally, the conn.commit() call commits the changes, so this system can perform indexing with a workload-driven approach unobtrusively, thereby enhancing query performance with no manual effort required.

The dynamic partitioning logic in Figure 19 allows PostgreSQL to self-optimize for workload demand. It watches the ingestion rates and the query time windows, and should the ingestion rate shoot past 10,000 records per second with the average query window in and around 6 h, the system would then go for hourly partitioning to give the recent data finer granularity for access. In this process, the default catch-all

Figure 18
Code snippet for find indexing

```
cur.execute(f"""
DO $$
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM pg_indexes
        WHERE tablename = 'weather_aggregations'
          AND indexname = '{index_name}'
    ) THEN
        EXECUTE 'CREATE INDEX {index_name} ON weather_aggregations ({column})';
    END IF;
END $$;
""")

conn.commit()

logging.info(f"Index '{index_name}' created if not already present.")
```

Figure 19
Code snippet for dynamic partitioning logic

```
if ingestion_rate > 10000 and average_query_time_window < '6 hours':
    new_partition_granularity = 'hourly'
    cur.execute(
        "ALTER ORDERS DETACH PARTITION weather_aggregations_default;"
    )
```

weather_aggregations partition is first detached so that conflicts could be avoided, following which new hourly partitions are created. Such adaptive restructuring improves by the real-time workload pattern of aligning the table design, thus improving query performance and ingestion efficiency. Henceforth, it keeps the database responsive and scalable in high-velocity, short-range query scenarios.

In Figure 20, the adaptive parallelism tuning logic dynamically tunes the job of Flink (weather-flink-group) while behind the scenes monitoring the real-time performance metrics via the REST API of Flink. It obtains the metric backPressuredTimePerSecond, which shows operators being stalled due to backpressure. When this value crosses 0.7 (70%), the system issues a PATCH request to increase the parallelism of the job to 8 so that more subtasks can be run concurrently. Such an increase in parallelism means higher throughput and reduced latency without having to restart or redeploy jobs. Thus, essentially this enables Flink to adaptively tune execution parameters during workload stress, providing consistent performance during high-velocity streaming.

Figure 20
Code snippet for adaptive parallelism tuning

```
flink_api = "http://flink-jobmanager:8081/jobs/weather-flink-group"

metrics = requests.get(f"{flink_api}/metrics").json()
backpressure = float(
    [m['value'] for m in metrics if m['id'] == 'backPressuredTimePerSecond'][0]
)

if backpressure > 0.1:
    new_parallelism = 8
    requests.patch(
        f"{flink_api}/jobs/weather-flink-group",
        json={"parallelism": new_parallelism}
    )
```

4.4.3. Comparative analysis

The static optimization technique clearly suffers a tangible disadvantage when compared with adaptive optimization for real performance contributions. Dynamic creation and dropping of indexes based on live query patterns made it possible for average query latency to decrease by 60% (from 1.84 to 1.05 s). Ingest control also improved, as Kafka consumer lag sharply dropped from 13,000 to 3,500 records, while Flink ingestion latency dropped from 750 to 410 ms on the grounds of runtime tuning of parallelism and buffer settings. CPU utilization

further dropped from 72% to 64%, thereby indicating efficient resource utilization and uniform load distribution. In summary, the adaptive framework eliminated slower speed, throughput, scalability, and responsiveness compared to static optimization in dynamic workloads.

4.5. Comparative analysis with commercial adaptive databases

The validation for the performance enhancement of the proposed auto optimized stream is made entry-wise with the commercial adaptive databases such as Oracle Autonomous Database and IBM Db2 AI for z/OS. Built-in self-tuners adjust indexing, caching, and resource allocation under proprietary algorithms in these platforms. Oracle Autonomous Database performs dynamic workload optimizations with approximately 35%–40% latency reductions, and Db2 AI allocates resources approximately 30% more efficiently. However, an open-source adaptive framework presented here with ingestion stabilities at par or better under variable workloads manages as much as a 60% latency reduction. Commercial systems, being vendor-locked, rely on a closed architecture in the foreground. This framework demonstrates that autonomous, workload-aware optimizations are achievable with transparent rule-based decision logic at negligible overhead in open-source ecosystems (PostgreSQL, Kafka, and Flink). This comparison brings forward that open-source adaptive databases can have the level of intelligent automation traditionally available only to commercial ones.

4.6. Challenges and limitations

However, the successful indexing, partitioning, and sharding of PostgreSQL for real-time data processing faced several challenges and limitations during this project. These limitations affected the system's scalability, availability of data, and overall performance.

4.6.1. Complexities in data cleaning and preprocessing

Another major challenge posed this much difficulty, namely, the cleaning of the datasets before they could be ingested into the database. The raw dataset was required to undergo preprocessing steps for various components—for instance, treatment of missing values, format standardization, and internal consistency between the data values. Because real-time systems rely on clear and consistent data to enable proper and quick, in certain cases, real-time processing of any data across all its stages, inconsistencies within the dataset might concern Flink streaming datum processing or degrade performance in executing queries using PostgreSQL. The unstructured format of the dataset also made a number of tasks impossible to be automated, and cleaning the dataset demanded people's efforts before any real-time processing pipelines could be set into action.

4.6.2. Difficulty in locating IoT datasets

Predominantly providing real-time analytics for an IoT-based application, the work was hampered by the difficulty in finding a publicly available dataset that fits requirements regarding high velocity, sufficient historical records, and real-time streaming properties. Most IoT datasets are either private, need authentication, or are limited in scope to simulate a real-world scenario of streaming. This lack of a large-scale IoT dataset restricted testing real-time processing to its full potential with respect to Kafka and Flink in the IoT environment. Instead, another dataset was used that although useful for testing the pipeline, could hardly replicate high-frequency data ingestion and processing of IoT sensors.

4.6.3. Optimizing techniques and dataset constraints

Improvements made to the query performance of the database using indexing, partitioning, and sharding were somewhat undermined by the small dataset size. Typically, these are the techniques that show the highest benefits when implemented in larger databases that tend to be queried heavily. In view of dataset constraints, minor gains on query optimization were seen but not nearly to the extent that one could have anticipated under a large-scale environment. In addition to the fact that sharding works best under conditions of heavy data distribution, sharding was unable to be showcased in a fully normalized manner under the single-node database with such small datasets.

4.6.4. Metric monitoring overhead

In this frame of reference, adaptive optimization monitors metrics continuously, such as PostgreSQL query logs, Kafka consumer lag, and Flink job performance, to trigger immediate adjustments. However, there is a trade-off; sampling too often can be resource consuming and can impair performance, and sampling too rarely can cause delays on sudden spikes or anomalies and as such a suboptimal optimization. Consistent with that, rule-based systems would have delayed responses to workload variations, such as ingestion spikes or abrupt change in query, because the detection, decision, and adjustment always take time. When combined with that, the extensions that exist in the feedback loop at peak loads or sudden surges will worsen performance, thereby recommending predictive and proactive optimization mechanisms that anticipate changes rather than simply reacting to them.

4.7. Key takeaways

1) This auto optimized stream framework integrated adaptive optimization with an open-source pipeline for real-time analytics using PostgreSQL, Apache Kafka, and Apache Flink.
2) Runtime indexing, partitioning, and streaming parallelism self-tune themselves, so the system can adapt to changing workloads without human intervention.
3) Experimental evaluation resulted in query latency reductions of up to 60%, with better-looking ingestion stability, throughput, and CPU utilization during varying load conditions.
4) This framework establishes that open-source databases can approach autonomy and performance of commercial adaptive systems such as Oracle Autonomous Database and IBM Db2 AI for z/OS.
5) Dynamic partitioning, along with rule-based decision logic, kept the pipeline performing with low latency and balanced resource usage during data velocity fluctuations.
6) Containerized deployment through Docker promoted greater scalability, fault isolation, and reproducibility of the system across environments.
7) This study settles that adaptive workload-aware optimization can be implemented transparently and cheaply in an open-source ecosystem, thereby reducing the need for proprietary tuning mechanisms.
8) This research can provide the groundwork for the future self-optimization of real-time database systems by leveraging predictive and machine-learning means.

5. Future Directions

With the success of this project in implementing real-time processing, indexing, partitioning, and sharding for optimizing PostgreSQL, several areas remain for future work to boost scalability, efficiency, and applicability in larger data environments.

1) Advanced database optimization technique

Other database optimizations may be studied in conjunction with indexing, partitioning, and sharding to further enhance query performance and scalability.

a. Caching mechanisms: implementing query caching and materialized views will significantly mitigate database load by reusing previously computed results, thereby increasing the speed of repeated queries.
b. Autonomous indexing: machine learning-based indexing will have a bearing on recommending indexes based on query patterns dynamically.
c. Adaptive query execution: will have implemented the strategy of dynamic query optimization whereby queries get automatically restructured based on workload patterns and hardware resources.

2) Performance benchmarking using YCSB and TPC-C

To collaborate the performance gains from sharding, indexing, and partitioning, different industry-standard benchmarking tools such as Yahoo Cloud Serving Benchmark (YCSB) and Transaction Processing Performance Council Benchmark (TPC-C) can be employed.

a. YCSB benchmarking: this shall help in analyzing how the database performs under different read/write workloads, simulating real-time transactional processing scenarios.
b. TPC-C benchmarking: this will evaluate particularly online transaction processing (OLTP) performance with the instance of handling simultaneous queries of users in a high-volume environment.

3) Machine learning for query optimization and predictive optimization

In works to be undertaken are machine learning applications for query execution and workload management in PostgreSQL, whereas real-time adaptive optimization stands for predictive ML models that tune themselves rather than being static rule-based systems. ML could predict bottlenecks and take proactive remedial measures based on previous workloads and system statistics. Consider time-series forecasting, reinforcement learning, and neural networks for dynamically adjusting indexing, query planning, and stream processing parameters for latency reductions and maximizing throughput and resource efficiency. The integration of all of these sets the stage for the foundation of a truly autonomous intelligent database system for real-time big-data environments.

6. Conclusion

This study investigated the database improvement strategies, indexing, partitioning, caching, and sharding in real-time analytics pipeline architectures in big-data environments. Using Apache Kafka, Apache Flink, and PostgreSQL as an open-source framework, this study demonstrated the gains in query speed, ingestion throughput, and scalability. Indexing through one form or another reduced query latencies, while partitioning and sharding distributed data and workload efficiently among backend nodes and enabled the system to work on fast streams for real-time decisions in various domains such as finance industry, e-commerce, IoT, and cybersecurity. A novel contribution is the introduction of an adaptive optimization layer that tunes PostgreSQL indexing, partitioning strategies, and Flink processing settings dynamically based on real-time workload metrics such as query latency, Kafka lag, and ingestion delays. Experimental results show that the adaptive layer performs much better than static techniques, decreasing the latency by more than 40% and providing a more stable ingestion under variations. The framework proves that open-source systems can self-optimize, providing a scalable and lightweight alternative to proprietary autonomous databases. Future work includes predictive ML-based tuning, hybrid RDBMS–NoSQL systems, and adaptive, security-aware optimization for multitenant cloud environments, providing practical insights for researchers and practitioners designing intelligent, high-performance, real-time, low-latency analytics systems with adaptive indexing.

Funding Support

This study received financial support from the Research School at York St John University.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by any of the authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

The data that support the findings of this study are openly available in "vega-datasets" at https://github.com/vega/vega/blob/main/docs/data/seattle-weather.csv.

Author Contribution Statement

Lakmali Karunarathne: Conceptualization, Methodology, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration. **Swathi Ganesan:** Investigation. **Kavindu Karunarathne:** Methodology, Software, Formal analysis, Resources, Writing – original draft, Writing – review & editing, Visualization. **Nalinda Somasiri:** Investigation.

References

[1] Abdalla, H. B., Kumar, Y., Zhao, Y., & Tosi, D. (2025). A comprehensive survey of MapReduce models for processing big data. *Big Data and Cognitive Computing*, *9*(4), 77. https://doi.org/10.3390/bdcc9040077

[2] Ponnusamy, S., & Gupta, P. (2024). Scalable data partitioning techniques for distributed data processing in cloud environments: A review. *IEEE Access*, *12*, 26735–26746. http://doi.org/10.1109/ACCESS.2024.3365810

[3] Zhu, Y., Chang, Y., Zhang, J., Song, Y., & Tang, Z. (2025). An optimized hierarchical MapReduce framework in supercomputing internet environment. *CCF Transactions on High Performance Computing*, *7*(3), 245–259. https://doi.org/10.1007/s42514-025-00218-1

[4] Agal, S., & Odedra, N. D. (2025). Impact of predictive analytics on algorithmic trading: Enhancing strategy performance and profitability. *Degrés*, *10*(2), 184–198.

[5] Song, H., Wang, Y., Chen, X., Feng, H., Feng, Y., Fang, X., …, & Kong, L. (2025). K2: On optimizing distributed transactions in a multi-region data store with TrueTime clocks. *Proceedings of the VLDB Endowment*, *18*(6), 1756–1769. https://doi.org/10.14778/3725688.3725704

[6] Choudhary, A. (2024). Internet of Things: A comprehensive overview, architectures, applications, simulation tools, challenges and future directions. *Discover Internet of Things*, *4*(1), 31. https://doi.org/10.1007/s43926-024-00084-3

[7] Song, S., Choi, J., Cha, D., Lee, H., Choi, D., Lim, J., …, & Yoo, J. (2025). Large-scale dynamic graph processing with graphic processing unit-accelerated priority-driven differential scheduling and operation reduction. *Applied Sciences*, *15*(6), 3172. https://doi.org/10.3390/app15063172

[8] Raptis, T. P., Cicconetti, C., & Passarella, A. (2024). Efficient topic partitioning of Apache Kafka for high-reliability real-time data streaming applications. *Future Generation Computer Systems*, *154*, 173–188. https://doi.org/10.1016/j.future.2023.12.028

[9] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., & Stonebraker, M. (2009). A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 165–178. https://doi.org/10.1145/1559845.1559865

[10] Abbasi, M., Bernardo, M. V., Váz, P., Silva, J., & Martins, P. (2024). Adaptive and scalable database management with machine learning integration: A PostgreSQL case study. *Information*, *15*(9), 574. https://doi.org/10.3390/info15090574

[11] Rahman, M. M., Islam, S., Kamruzzaman, M., & Joy, Z. H. (2024). Advanced query optimization in SQL databases for real-time big data analytics. *Academic Journal on Business Administration, Innovation & Sustainability*, *4*(3), 1–14.

[12] Levin, S. M. (2024). Unleashing real-time analytics: A comparative study of in-memory computing vs. traditional disk-based systems. *Brazilian Journal of Science*, *3*(5), 30–39. https://doi.org/10.14295/bjs.v3i5.553

[13] Nuriev, M., Zaripova, R., Sinicin, A., Chupaev, A., & Shkinderov, M. (2024). Enhancing database performance through SQL optimization, parallel processing and GPU integration. *BIO Web of Conferences*, *113*, 04010. https://doi.org/10.1051/bioconf/202411304010

[14] Gupta, A., & Jain, S. (2022). Optimizing performance of real-time big data stateful streaming applications on cloud. In *2022 IEEE International Conference on Big Data and Smart Computing*, 1–4. https://doi.org/10.1109/BigComp54360.2022.00010

[15] Qu, J., & Wang, J. (2024). Real-time data warehousing in the big data environment: A comprehensive review of implementation in the internet industry. *Applied and Computational Engineering*, *88*, 105–114. https://doi.org/10.54254/2755-2721/88/20241643

[16] Chen, J., Jindel, S., Walzer, R., Sen, R., Jimsheleishvilli, N., & Andrews, M. (2016). The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment*, *9*(13), 1401–1412. https://doi.org/10.14778/3007263.3007277

[17] Chen, J., Zhang, L., Xie, Y., Ding, W., Cao, L., Liu, Y., ..., & Zhao, P. (2025). veDB-HTAP: A highly integrated, efficient and adaptive HTAP system. *Proceedings of the VLDB Endowment*, *18*(12), 4896–4909. https://doi.org/10.14778/3750601.3750614

[18] Pedratscher, S., Samani, Z. N., Poveda, J. A., Fahringer, T., Etheredge, M., Younesi, A., …, & Thoman, P. (2025). STREAMLINE: Dynamic and resource-efficient auto-tuning of stream processing data pipeline ensembles. *Internet of Things*, *34*, 101731. https://doi.org/10.1016/j.iot.2025.101731

[19] Miryala, N. K. (2024). Evolving trends in open-source RDBMS: Performance, scalability and security insights. *International Journal of Science and Research*, *13*(2), 494–500. https://doi.org/10.21275/sr24126224648

[20] Zhang, Z., Megargel, A., & Jiang, L. (2025). Performance evaluation of NewSQL databases in a distributed architecture. *IEEE Access*, *13*, 11185–11194. https://doi.org/10.1109/access.2025.3529740

[21] Chen, J., Ding, Y., Liu, Y., Li, F., Zhang, L., Zhang, M., …, & Liang, Y. (2022). ByteHTAP: ByteDance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment*, *15*(12), 3411–3424. https://doi.org/10.14778/3554821.3554832

[22] Larson, P.-Å., Birka, A., Hanson, E. N., Huang, W., Nowakiewicz, M., & Papadimos, V. (2015). Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment*, *8*(12), 1740–1751. https://doi.org/10.14778/2824032.2824071

[23] Dusheba, V. V., & Gerasimov, V. R. (2024). Analiz metodiv optymizatsiyi roboty baz danykh [Analysis of optimizing database performance methods]. *Electronic Modeling*, *46*(6), 43–54. https://doi.org/10.15407/emodel.46.06.043

[24] Gajiwala, C. (2024). NoSQL technologies in big data ecosystems: A comprehensive review of architectural paradigms and performance metrics. *International Journal for Multidisciplinary Research*, *6*(5), 1–13. https://doi.org/10.36948/ijfmr.2024.v06i05.29547

[25] Carretero, M. A. (2024). Uma visão geral de banco de dados NOSQL para big data [An overview of NoSQL databases for big data]. *Revista ft*, *28*(139), 19–20. https://doi.org/10.69849/revistaft/pa10202410171619

[26] Yedatkar, A. S. (2024). Database management systems: An examination using NoSQL. *International Journal of Advanced Research in Science, Communication and Technology*, *4*(2), 375–378. https://doi.org/10.48175/IJARSCT-15765

[27] Candel, C. J. F., García-Molina, J. J., & Ruiz, D. S. (2023). SkiQL: A unified schema query language. *Data & Knowledge Engineering*, *148*, 102234. https://doi.org/10.1016/j.datak.2023.102234

[28] Khan, W., Kumar, T., Zhang, C., Raj, K., Roy, A. M., & Luo, B. (2023). SQL and NoSQL database software architecture performance analysis and assessments—A systematic literature review. *Big Data and Cognitive Computing*, *7*(2), 97. https://doi.org/10.3390/bdcc7020097

[29] Tu, H. (2024). Cassandra vs. MongoDB: A systematic review of two NoSQL data stores in their industry uses. In *2024 IEEE 7th International Conference on Big Data and Artificial Intelligence*, 81–86. https://doi.org/10.1109/BDAI62182.2024.10692676

[30] Malcher, M., & Kuhn, D. (2024). *Pro Oracle database 23c administration: Manage and safeguard your organization's data*. USA: Apress. https://doi.org/10.1007/978-1-4842-9899-2

[31] Huang, K., & Wang, T. (2024). *Indexing on non-volatile memory: Techniques, lessons learned and outlook*. Switzerland: Springer Nature. https://doi.org/10.1007/978-3-031-47627-3

[32] Abbasi, M., Bernardo, M. V., Váz, P., Silva, J., & Martins, P. (2024). Revisiting database indexing for parallel and accelerated computing: A comprehensive study and novel approaches. *Information*, *15*(8), 429. https://doi.org/10.3390/info15080429

[33] Abdullahi, A. U., Ahmad, R., & Zakaria, N. M. (2021). Indexing in big data mining and analytics. In H. Chiroma, S. M. Abdulhamid, P. Fournier-Viger, & N. M. Garcia (Eds.), *Machine learning and data mining for emerging trend in cyber dynamics: Theories and applications* (pp. 123–143). Springer. https://doi.org/10.1007/978-3-030-66288-2_5

[34] Paludo Licks, G., Colleoni Couto, J., de Fátima Miehe, P., de Paris, R., Dubugras Ruiz, D., & Meneguzzi, F. (2020). SMARTIX: A database indexing agent based on reinforcement learning. *Applied Intelligence*, *50*(8), 2575–2588. https://doi.org/10.1007/s10489-020-01674-8

[35] Wang, J. (2024). Dynamic adjustment paradigm based on genetic algorithm in database index optimization. In *2024 International Conference on IoT Based Control Networks*

and Intelligent Systems, 954–960. https://doi.org/10.1109/ICICNIS64247.2024.10823213

[36] Zolotukhina, D. (2025). Effektivnost' raspredelonnykh keshiruyushchikh platform v sovremennykh backend-arkhitekturakh: sravnitel'nyy analiz Redis i Hazelcast [The efficiency of distributed caching platforms in modern backend architectures: A comparative analysis of Redis and Hazelcast]. *Software Systems and Computational Methods*, (4), 192–204. https://doi.org/10.7256/2454-0714.2024.4.72305

[37] Glasbergen, B., Langendoen, K., Abebe, M., & Daudjee, K. (2020). ChronoCache: Predictive and adaptive mid-tier query result caching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2391–2406. https://doi.org/10.1145/3318464.3380593

[38] Späth, P. (2023). *Pro Jakarta EE 10: Open source enterprise Java-based cloud-native applications development*. USA: Apress.

[39] Privalov, M. V., & Stupina, M. V. (2024). Improving web-oriented information systems efficiency using Redis caching mechanisms. *Indonesian Journal of Electrical Engineering and Computer Science*, *33*(3), 1667–1675. https://doi.org/10.11591/ijeecs.v33.i3.pp1667-1675

[40] Tiwari, P., Vittal, S., & A, A. F. (2024). Sharding the datastore network functions of 5G core for scalable and resilient slice service. In *2024 IEEE 10th International Conference on Network Softwarization*, 331–335. https://doi.org/10.1109/netsoft60951.2024.10588906

[41] Rana, R., Kannan, S., Tse, D. N., & Viswanath, P. (2022). Free2Shard: Adversary-resistant distributed resource allocation for blockchains. In *Proceedings of the ACM on Measurement and Analysis of Computing System*, *6*(1), 11. https://doi.org/10.1145/3508031

[42] Padmanaban, K., Ganesh Babu, T. R., Karthika, K., Pattanaik, B., K, D., & Srinivasan, C. (2024). Apache Kafka on big data event streaming for enhanced data flows. In *2024 8th International Conference on I-SMAC*, 977–983. https://doi.org/10.1109/i-smac61858.2024.10714884

[43] Chaudhuri, S., & Narasayya, V. (1997). An efficient, cost-driven index selection tool for Microsoft SQL server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, 146–155.

[44] Graefe, G., & Kuno, H. (2010). Adaptive indexing for relational keys. In *2010 IEEE 26th International Conference on Data Engineering Workshops*, 69–74. https://doi.org/10.1109/ICDEW.2010.5452743

[45] Malikireddy, S. K. R. (2022). Self-adapting real-time data ecosystems with autonomous multi-agent systems. *World Journal of Advanced Research and Reviews*, *13*(3), 593–607.

[46] Masandilov, I. (2025). Instrumenty profilirovaniya proizvoditel'nosti Go-prilozheniy v real'nom vremeni [Performance profiling tools for GO applications in real-time environments]. *Universum: Technical Sciences*, *9*(138), 11–16. https://doi.org/10.32743/unitech.2025.138.9.20806

[47] Fetai, I., Murezzan, D., & Schuldt, H. (2015). Workload-driven adaptive data partitioning and distribution—The Cumulus approach. In *2015 IEEE International Conference on Big Data*, 1688–1697. https://doi.org/10.1109/BigData.2015.7363940

[48] Soltani, K., Padmanabhan, A., & Wang, S. (2022). GeoBalance: Workload-aware partitioning of real-time spatiotemporal data. *GeoInformatica*, *26*(1), 67–94. https://doi.org/10.1007/S10707-021-00444-Z

How to Cite: Karunarathne, L., Ganesan, S., Karunarathne, K., & Somasiri, N. (2026). Database Optimization for Low-Latency Analytics with Adaptive Indexing. *Journal of Data Science and Intelligent Systems*. https://doi.org/10.47852/bonviewJDSIS62027607