Ministry of Education and Science of Ukraine
Ternopil Ivan Puluj National Technical University

Faculty of Computer Information System and Software Engineering
(full name of faculty)
Department of Computer Science
(full name of department)

QUALIFYING PAPER

For the degree of

Bachelor
(degree name)

topic: Parallel Processing for Real – Time Stream Analytics

Submitted by: student IV course, group ICH-43
specialty 122 Computer science
(шифр і назва спеціальності)

Emmanuel Yaw Mac-Gatus
(signature) (surname and initials)

Supervisor Roman Zolotyi
(signature) (surname and initials)

Standards verified by
(signature) (surname and initials)

Head of Department Bodnarchuk I.O.
(signature) (surname and initials)

Reviewer
(signature) (surname and initials)

Ternopil
2026

Ministry of Education and Science of Ukraine
Ternopil Ivan Puluj National Technical University

Faculty Faculty of Computer Information System and Software Engineering
(full name of faculty)

Department Department of Computer Science
(full name of department)

APPROVED BY
Head of Department

_____ Bodnarchuk I.O.
(signature) (surname and initials)

« » 20___

ASSIGNMENT
for QUALIFYING PAPER

for the degree of Bachelor
(degree name)

specialty 122 Computer science
(code and name of the specialty)

student Emmanuel Yaw Mac-Gatus
(surname, name, patronymic)

1. Paper topic Parallel Processing for Real – Time Stream Analytics

Paper supervisor Zolotyi R.Z., PhD
(surname, name, patronymic, scientific degree, academic rank)

Approved by university order as of «____» _____ ____ № __

2. Student's paper submission deadline

3. Initial data for the paper Literature sources about architecture, principles of operation and development of information systems.

4. Paper contents (list of issues to be developed)

5. List of graphic material (with exact number of required drawings, slides)

6. Advisors of paper chapters

Chapter	Advisor's surname, initials and position	Signature, date	
		assignment was given by	assignment was received by
Life safety, basics of labor protection			

7. Date of receiving the assignment 20.01.2025

TIME SCHEDULE

LN	Paper stages	Paper stages deadlines	Notes
1	Analysis of the task for qualifying work. Selection		*Completed*
	and work with literary sources.		
2	Writing chapter 1		*Completed*
3	Writing chapter 2		*Completed*
4	Writing chapter 3		*Completed*
5	Writing chapter 4		*Completed*
6	Standartization control		*Completed*
7	Plagiarism check		*Completed*
8	Preliminary defense of qualifying paper		*Completed*
9	Defense of qualifying paper		

Student _____ Emmanuel Yaw Mac-Gatus
(signature) (surname and initials)

Paper supervisor _____ Zolotyi R.Z.
(signature) (surname and initials)

ANNOTATION

Parallel Processing for Real – Time Stream Analytics // Diploma thesis Bachelor degree // Emmanuel Yaw Mac-Gatus // Ternopil' Ivan Puluj National Technical University, Faculty of Computer Information System and Software Engineering, Department of Computer Science // Ternopil', 2026 // P. __, Fig. – __, Tables – __, Annexes – __, References – __.

Keywords: Real-time stream analytics, parallel processing, distributed systems, data streams, low-latency computing.

Parallel processing helps overcome limitations by distributing data and computations across multiple processing units, allowing everything to run simultaneously. This approach enables stream analytics systems to handle higher volumes of data, respond more quickly, and scale more effectively. As a result, they are ideally suited for continuous, unbounded data streams. This paper offers a thorough review of methods for parallel processing in real-time stream analytics, exploring key architectural designs, stream-processing models, and parallelization strategies that improve the efficiency and scalability of live data analysis.

This paper also explores the everyday challenges faced when implementing parallel stream processing. These include coordinating tasks, maintaining consistent data states, tolerating faults, balancing workloads, and managing data skew. If not properly handled, these issues can seriously affect how well the system performs and how reliably it operates. By examining current solutions and emerging trends in stream processing frameworks and hardware acceleration, the study highlights the growing importance of parallel processing for real-time data analysis. The results show that effective parallelization is crucial for supporting modern, data-driven applications across various fields, including smart cities, finance, healthcare, and industrial automation.

CONTENT

INTRODUCTION

The rapid growth of data-generating technologies has changed how information is created and used in many fields. Devices such as those in the Internet of Things, social media platforms, financial systems, and sensor networks constantly generate large volumes of real-time data [1]. To extract meaningful insights, immediate processing is essential. Delays, even brief ones, diminish value—especially when monitoring for fraud, ensuring smooth traffic flow, or maintaining system reliability. This is where challenges arise: traditional methods weren't built for this. They work through batch jobs, handling tasks sequentially, which made sense years ago but falls apart when you're drowning in data that demands instant analysis [2].

Parallel processing helps overcome limitations by distributing data and computations across multiple processing units, allowing everything to run simultaneously. This approach enables stream analytics systems to handle higher volumes of data, respond more quickly, and scale more effectively. As a result, they are ideally suited for continuous, unbounded data streams [3]. This paper offers a thorough review of methods for parallel processing in real-time stream analytics, exploring key architectural designs, stream-processing models, and parallelization strategies that improve the efficiency and scalability of live data analysis [4].

This paper also explores the everyday challenges faced when implementing parallel stream processing. These include coordinating tasks, maintaining consistent data states, tolerating faults, balancing workloads, and managing data skew. If not properly handled, these issues can seriously affect how well the system performs and how reliably it operates [5]. By examining current solutions and emerging trends in stream processing frameworks and hardware acceleration, the study highlights the growing importance of parallel processing for real-time data analysis. The results show that effective parallelization is crucial for

supporting modern, data-driven applications across various fields, including smart cities, finance, healthcare, and industrial automation [6].

1. ANALYSIS OF THE SUBJECT AREA AND STATEMENT OF THE TASK

1.1. Background and Motivation

The continuous progress in digital technologies has profoundly changed how we create, share, and look at data. In the past, data was mainly gathered in separate groups, stored in databases, and handled based on fixed plans. These datasets were typically finite, systematically organized, and comparatively small in volume. In contrast, contemporary systems frequently produce continuous data streams, exemplifying an unceasing flow of information generated over time. A data stream is a real-time, constant flow of data points that arrive in sequence and often require immediate processing or rigorous temporal management.

These ongoing data streams come from many different sources. You might find sensor networks monitoring the environment or industrial processes, wearable and smart devices, Internet of Things (IoT) platforms, online transaction and payment systems, web server and application logs, or social media sites. The data from these sources usually has three main traits: it comes in quickly, it’s large in volume, and it keeps coming without stopping. "High velocity" means data is generated and sent very fast. "Large volume" refers to the enormous, often limitless amount of data. "Continuous arrival" means the data flow has no clear end. Because of these traits, storing all incoming data before analyzing it is often not possible, so traditional batch processing methods don’t work well here.

To tackle these issues, the concept of real-time stream analytics has emerged. This approach involves analyzing, transforming, and deriving insights from streaming data as it arrives, with minimal delay between data arrival and the resulting insights. Unlike batch analytics, which looks at past data, real-time analytics focuses on immediate response. This is crucial in situations where data loses its value quickly. For example, in financial systems, real-time analytics helps

spot fraudulent transactions early; in intelligent transportation, it supports live traffic flow and congestion control; in stock markets, it enables quick trend detection; and in healthcare, it allows continuous monitoring of patients' vital signs. In these cases, delays can lead to financial losses, lower system efficiency, or even endanger human safety.

Despite its advantages, real-time stream analytics introduces significant computational complexity. As data streams grow in speed and scale, sequential processing models, where data items are processed one after another on a single processing unit, become increasingly inefficient. Such models struggle to meet strict latency and throughput requirements, leading to processing backlogs and delayed responses. This limitation becomes more severe as modern applications demand real-time analysis of millions of events per second.

To address these challenges, parallel processing has become a key technique in stream analytics systems. Parallel processing is a computational approach in which multiple processing units execute tasks simultaneously to solve problems more efficiently. In stream analytics, it involves splitting incoming data streams and analytical tasks across multiple processors, cores, or distributed machines. By enabling concurrent execution, parallel processing greatly enhances throughput, lowers processing latency, and improves system scalability.

As a result, it plays a critical role in meeting the performance demands of modern real-time analytics applications.

As more industries rely on making quick, real-time decisions, it's essential to explore how parallel processing can help analyze data streams effectively. Understanding how to use parallelism effectively, along with the challenges it entails, is key to building real-time data systems that are efficient, reliable, and scalable.

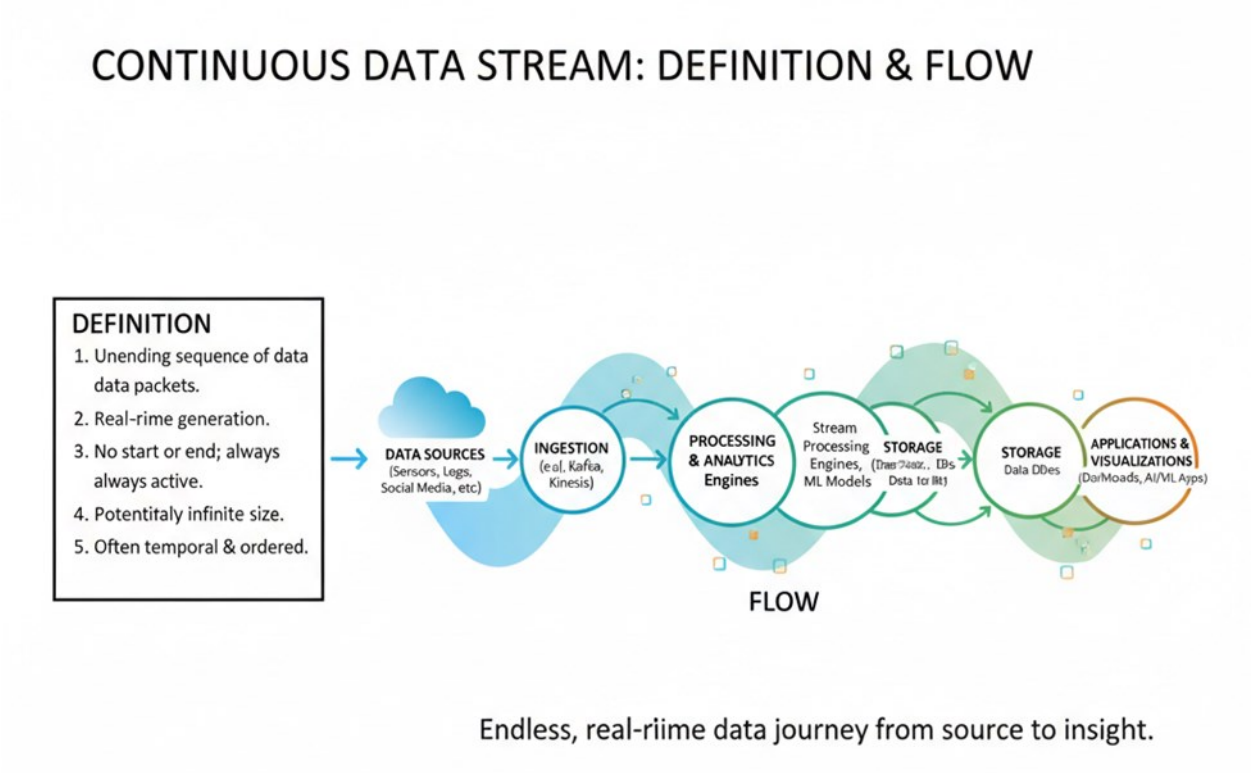

Figure 1.1 - Definition and Flow of Continuous Data Streams

Real-Time Stream Analytics Overview

Building on the previous discussion, real-time stream analytics marks a shift in how data is processed and analyzed in modern systems. Unlike batch analytics, which collects data over a set period for later processing, stream analytics processes data as it arrives. This approach enables systems to produce insights continuously, rather than waiting for the entire dataset.

The primary goal of real-time stream analytics is to facilitate rapid decision-making. Because data can quickly lose its relevance, delayed analysis often proves ineffective. Consequently, stream analytics systems are designed to process events rapidly, typically delivering results within milliseconds or seconds. This distinguishes stream analytics from traditional data processing by prioritizing speed, responsiveness, and system efficiency.

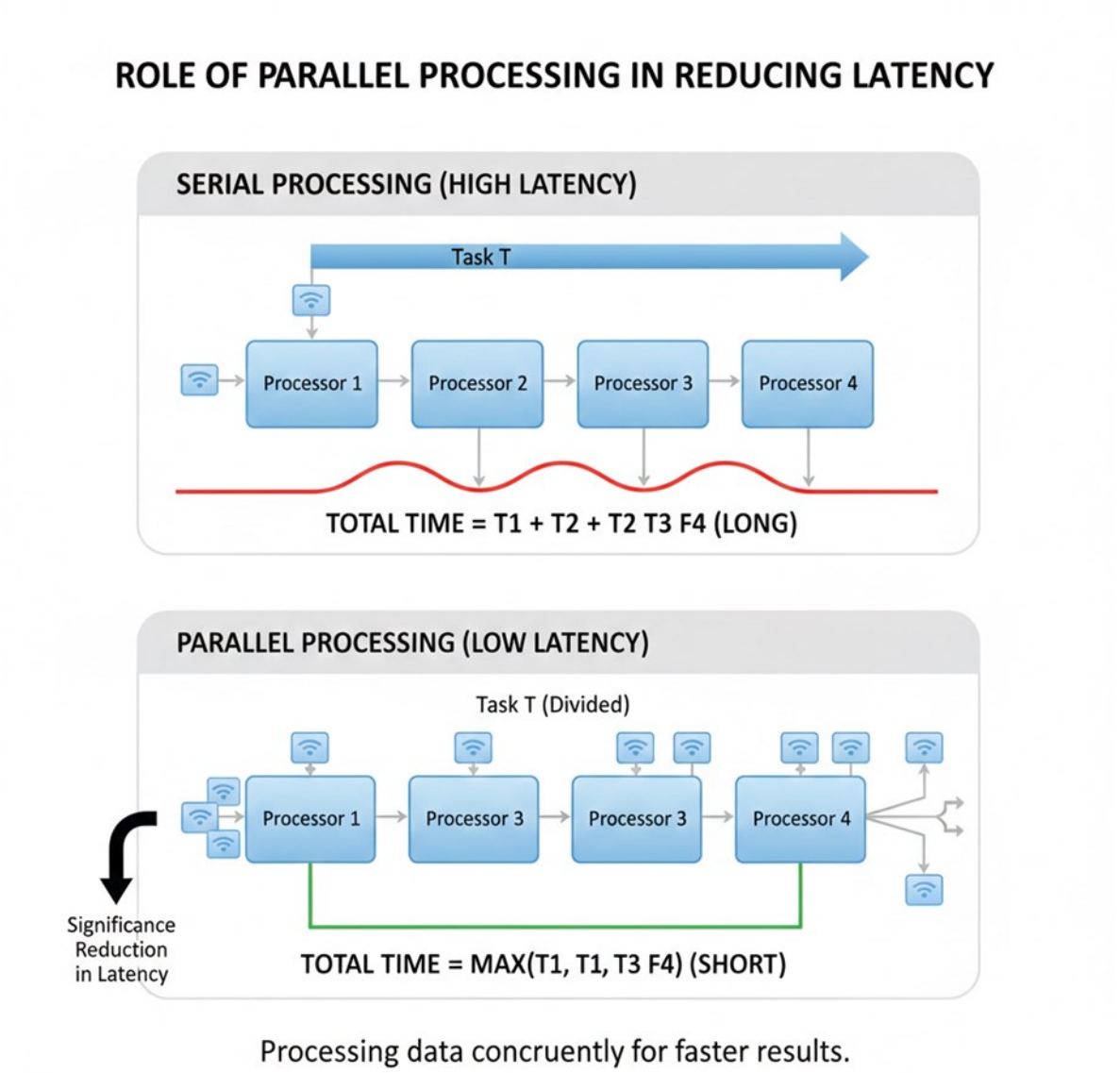

Figure 1.2 – Role of Parallel Processing in Reducing Latency

Real-time stream analytics systems have some key features. They keep processing incoming data nonstop, so they need to run continuously. Their ability to handle data quickly is essential because even minor delays can affect performance and decision accuracy. These systems also often track context across multiple events, which helps spot patterns, trends, or anomalies. Scalability is essential too, since data volumes can suddenly grow.

These features really challenge computer resources and how systems are built. Just using one machine and processing tasks one after another usually isn't

enough. That's why many modern stream analytics tools are moving toward parallel and distributed computing models, helping them run faster and scale more easily.

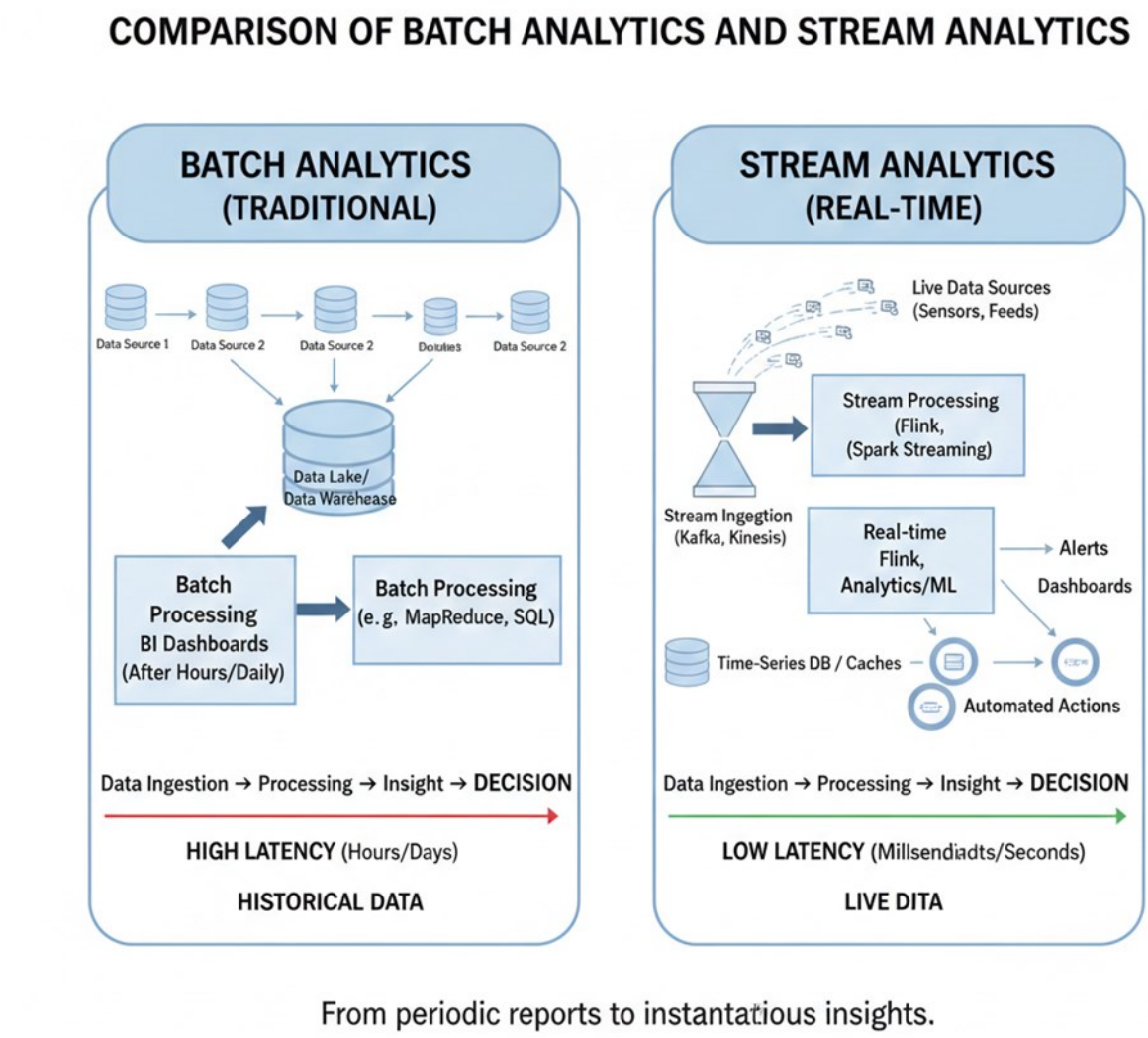

Figure 1.3 - Comparison of Batch Analytics and Stream Analytics

Role of Parallel Processing

To meet the performance demands of real-time stream analytics, parallel processing plays a central and enabling role. Rather than executing operations

sequentially, parallel processing allows multiple computations to run concurrently by distributing workloads across multiple processing units. In streaming environments, this approach enables systems to handle large volumes of incoming events without sacrificing responsiveness.

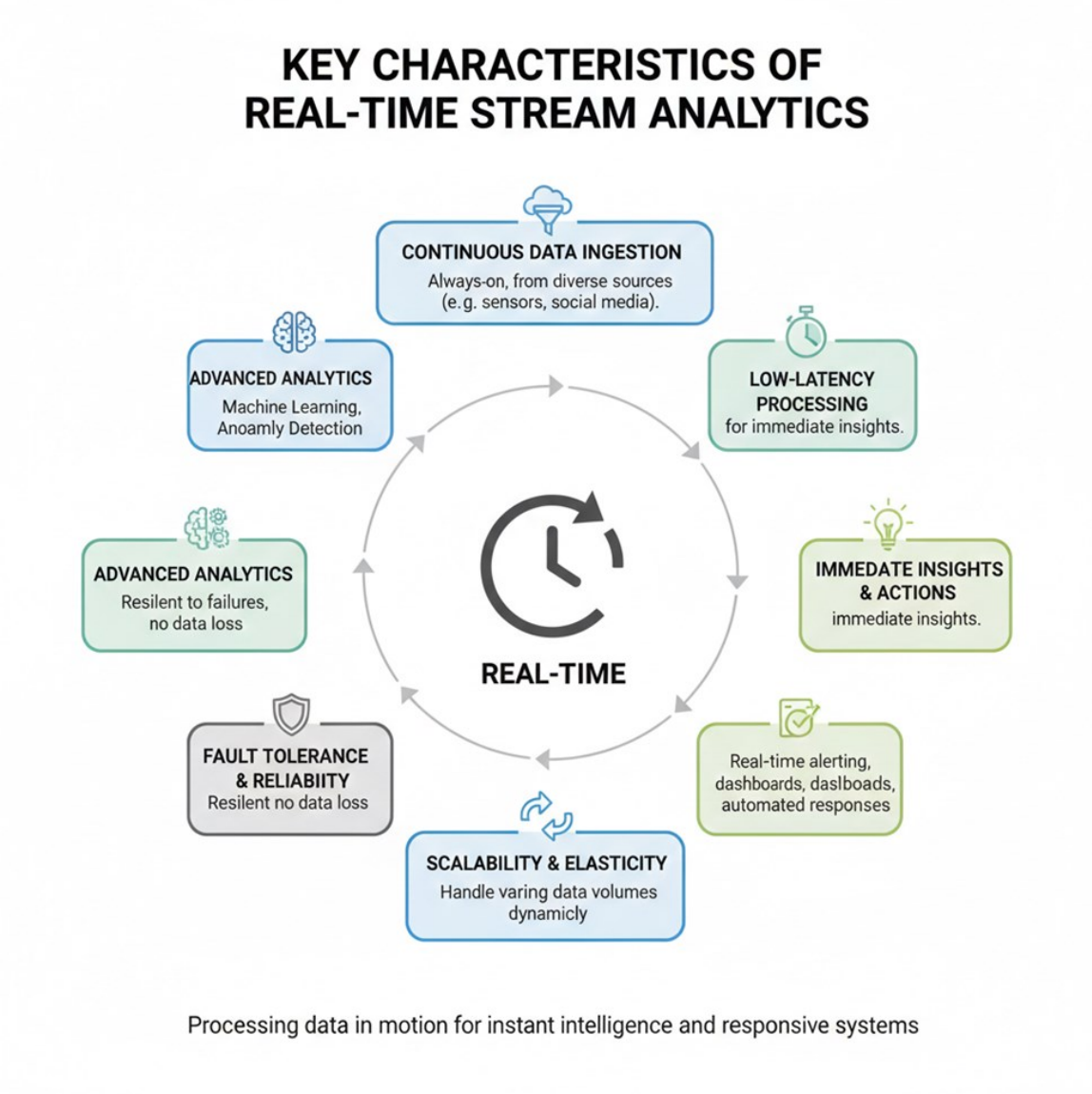

Figure 1.4 - Key Characteristics of Real-Time Stream Analytics

In real-time stream analytics, parallelism can be implemented at multiple levels of the system architecture. A prevalent method is data-level parallelism, in which incoming data streams are partitioned for independent processing. This

facilitates the concurrent analysis of multiple events, thereby substantially enhancing throughput. Additionally, task-level parallelism involves executing different analytical operations or processing stages simultaneously. Pipeline parallelism extends this concept by enabling data to traverse a series of processing stages that operate concurrently, thereby optimizing overall system efficiency.

By leveraging these forms of parallelism, stream processing systems can scale horizontally by adding more processing resources as data rates increase. This flexibility allows systems to adapt dynamically to changing workloads while maintaining consistent performance across varying conditions. As a result, parallel processing is not merely an optimization technique but a fundamental requirement for achieving real-time performance in modern stream analytics systems.

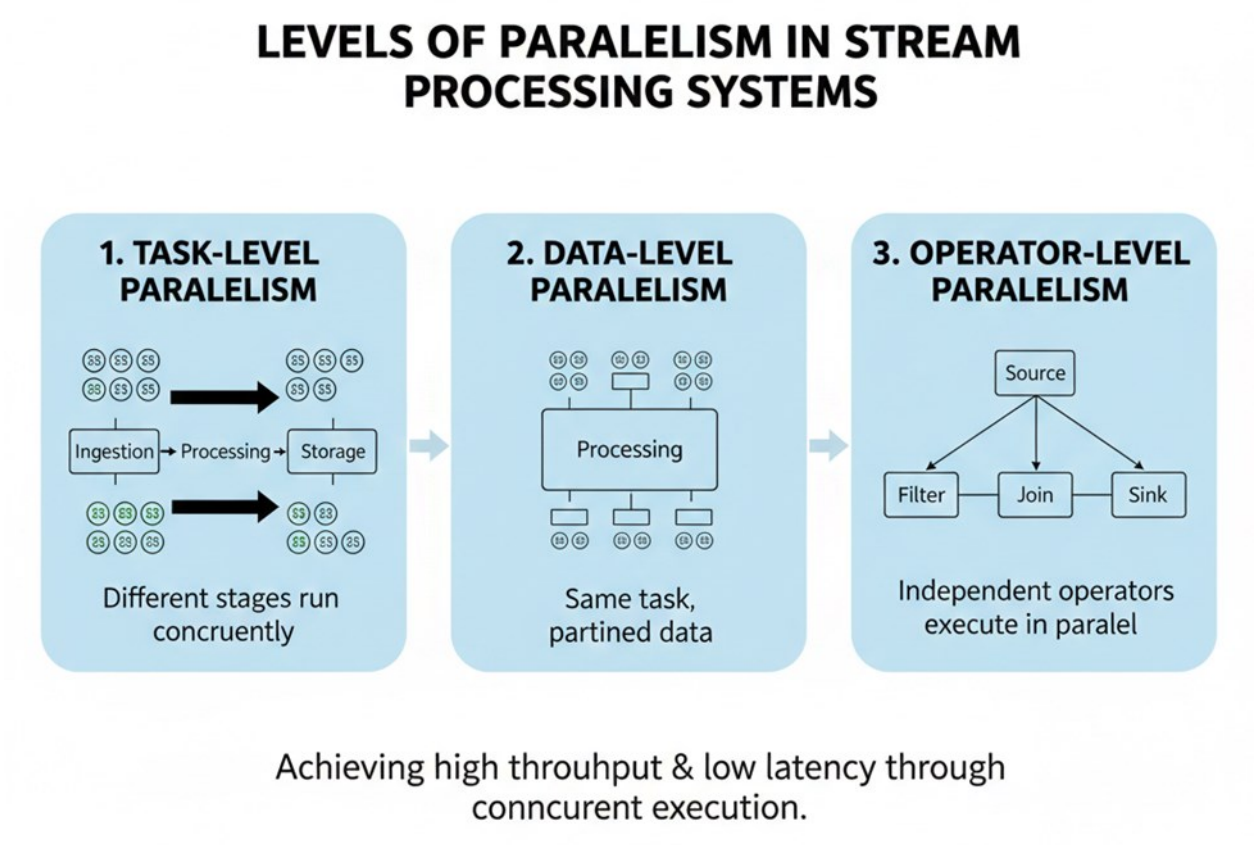

Figure 1.5 - Levels of Parallelism in Stream Processing Systems

Challenges in Parallel Stream Processing

While running tasks in parallel can significantly boost performance, it also introduces tricky technical challenges that require careful attention. One major issue is handling shared state, since many stream analytics apps rely on it across different tasks. Keeping everything consistent and correct can be quite challenging. If the state isn’t managed properly or is poorly designed, it can lead to wrong results or even make the system unstable.

Synchronization overhead is a significant concern because coordinating parallel tasks often involves synchronization mechanisms that, if not optimized, can decrease performance gains. Too much synchronization can cause delays that negate the advantages of parallel execution. Moreover, data skew—where some data partitions carry much more workload than others—can create load imbalance among processing units. This imbalance hampers overall system efficiency and restricts scalability.

Fault tolerance is crucial in systems that handle multiple streams of data simultaneously. In big-scale setups, things like hardware breaking down, network issues, and software bugs happen more often than you'd like. That's why these systems need to bounce back from problems quickly, ensuring no data is lost and everything keeps running smoothly. Finding ways to make these systems both highly efficient and really reliable is an ongoing and exciting area of research.

Objectives and Scope of the Paper

The main goal of this paper is to explore and explain how parallel processing techniques are used in real-time stream analytics systems. We want to improve understanding of how parallelism enables the quick, scalable processing of continuous data streams. Specifically, this paper aims to:

1. Describe the basic ideas behind analyzing live data streams and handling multiple tasks at the same time.

2. Describe how systems are designed to handle tasks by breaking them into smaller parts and working on them simultaneously across multiple machines.

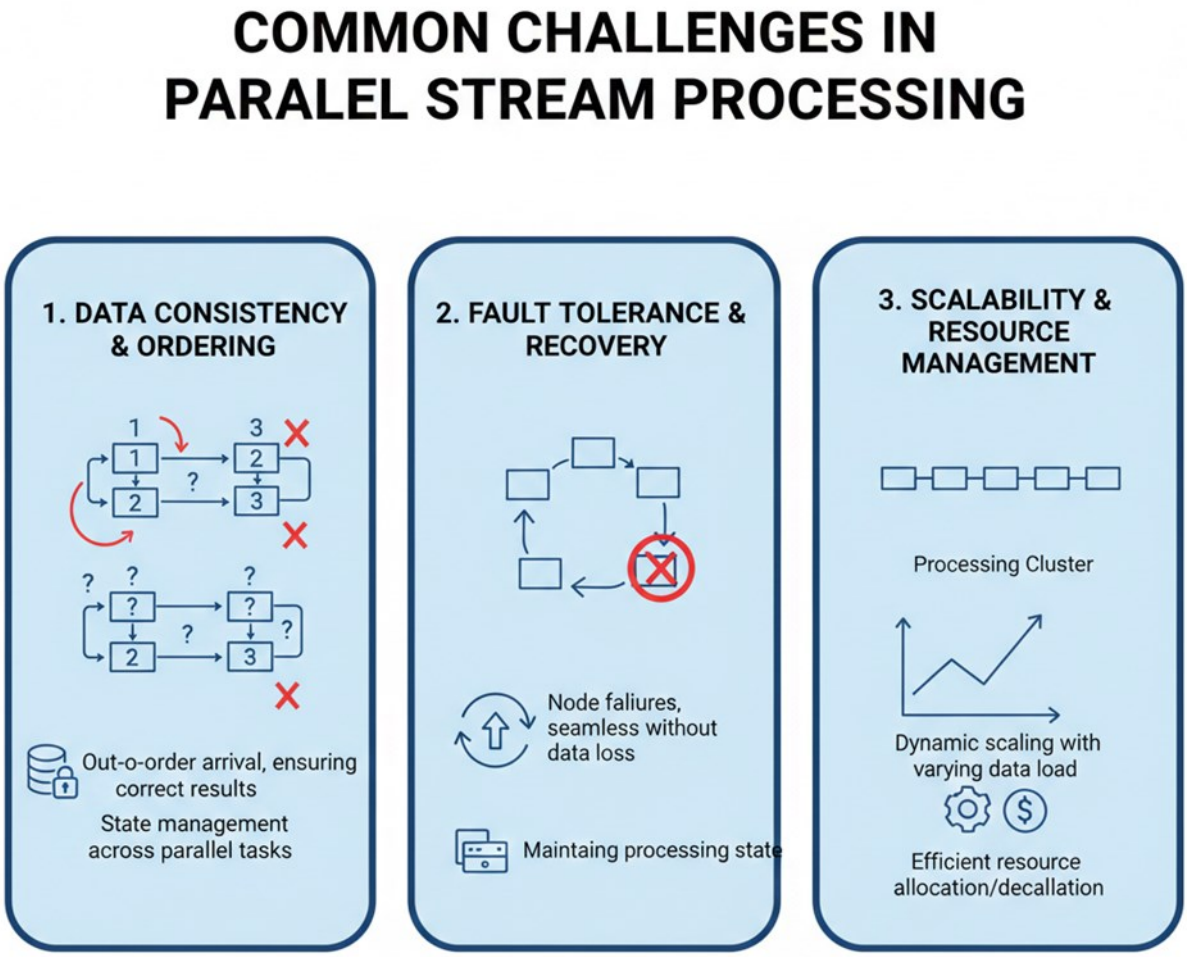

Figure 1.6 - Common Challenges in Parallel Stream Processing

3. Examine the models and techniques used for parallel processing in modern stream analytics platforms.

4. Identify the main challenges, limitations, and compromises involved in processing data with parallel streams.

5. Explore new trends and future research paths in the field.

This paper primarily explores ideas, system designs, and processing methods rather than delving into the nitty-gritty details of implementation. This way, the

discussion can be relevant and applicable across a variety of platforms and applications.

Organization of the Paper

The remainder of this paper is organized to provide a logical and progressive exploration of the topic. Section II reviews foundational concepts and related work in real-time stream analytics and parallel processing. Section III discusses system architectures for real-time stream analytics. Section IV examines parallel processing models and techniques in detail. Section V explores algorithms and frameworks commonly used in parallel stream processing systems. Section VI discusses key challenges and limitations. Section VII presents discussion and future research trends, and Section VIII concludes the paper.

1.2. Background and related work

Fundamentals of Stream Analytics

Stream analytics is a data processing paradigm that focuses on the real-time or near-real-time analysis of continuously generated data. Unlike traditional data analytics systems that operate on fixed, stored datasets, stream analytics systems are designed to process data that arrives constantly and may have no predefined end. This unbounded nature of streaming data requires systems to analyze information as it flows through the system rather than waiting for complete datasets to be collected.

At its core, the primary objective of stream analytics is to extract meaningful insights, patterns, or anomalies from data as soon as it is generated. These insights may include detecting abnormal behavior, identifying trends, triggering alerts, or supporting automated decision-making. Because streaming data is often time-sensitive, its usefulness can diminish rapidly if analysis is delayed. As a result, stream analytics emphasizes immediacy, responsiveness, and efficiency.

Streaming data is commonly described using the well-known “three V’s”: velocity, volume, and variety. Velocity refers to the speed at which data is generated and transmitted, often at very high rates. Volume refers to the potentially massive, continuously growing amount of data produced over time. Variety indicates the heterogeneous nature of streaming data, which may include structured, semi-structured, and unstructured formats originating from diverse sources. Together, these characteristics distinguish stream analytics from traditional data processing approaches and introduce unique computational challenges.

In real-time stream analytics, data is typically processed either event-by-event or in small batches. This incremental processing model allows applications to respond quickly to changes in the data stream. As a result, stream analytics is well-suited for time-critical domains such as financial trading systems, cybersecurity monitoring, industrial automation, smart grids, and intelligent transportation systems. In these environments, the ability to analyze and act on data in real time can provide significant operational and strategic advantages.

Stream Processing Models

To enable real-time analysis of streaming data, different processing approaches have been developed and are commonly used in practice. These approaches dictate how incoming data is grouped, handled, and processed within a system. The two most popular methods are processing data one event at a time and processing it in small batches.

Event-at-a-time processing, also called true streaming, means handling each new event as it arrives. This approach offers very low latency, making it ideal for tasks that require quick responses, such as fraud detection, security breaches, or real-time alerts. But processing one event at a time can also create extra work, especially when it comes to tracking data or managing multiple tasks simultaneously. Updating stored information and syncing everything can slow things down if not done carefully.

Micro-batch processing groups incoming events into small batches that are processed together over short time frames. This approach reduces workload by spreading computation across multiple events, boosting overall speed. Although it introduces a slight delay compared to processing each event alone, it still delivers results fast enough for many uses. Plus, it simplifies error handling and state management, which is why many modern stream processing systems favor this approach.

Both processing methods really benefit from parallel techniques. Splitting data or tasks across multiple processing units enables systems to handle high input volumes without excessive delay. Deciding whether to process one event at a time or in small batches usually depends on the application's needs, especially when balancing speed, efficiency, and system complexity.

Overview of Parallel Processing Concepts

Parallel processing is a fundamental computing paradigm in which multiple processing elements execute tasks simultaneously to solve problems more efficiently. In the context of stream analytics, parallel processing is essential for meeting real-time performance requirements and achieving scalability under high data rates.

Parallelism in stream analytics can be applied in several forms. Data parallelism involves partitioning incoming data streams into independent subsets that can be processed concurrently using the same operations. This approach is convenient when events are independent or can be grouped by keys such as user identifiers or sensor IDs. Task parallelism, on the other hand, focuses on executing different operations or analytical tasks in parallel. This model is useful when a stream analytics application consists of multiple independent processing stages.

Pipeline parallelism is another essential form of parallelism, in which different stages of a processing pipeline operate concurrently on distinct data items. As data flows through the pipeline, each stage performs a specific function, such as filtering, aggregation, or pattern detection. By overlapping execution

across stages, pipeline parallelism improves overall throughput and resource utilization.

In practice, modern stream analytics systems often combine multiple forms of parallelism to maximize performance. The effective use of parallel processing allows systems to scale horizontally, adapt to fluctuating workloads, and maintain low-latency processing even under heavy data loads.

Distributed Stream Processing Systems

Most real-time stream analytics platforms are implemented as distributed systems due to the scale and complexity of modern data streams. In distributed stream processing systems, computational tasks are executed across multiple machines connected through a network. This distributed architecture enables horizontal scalability, allowing systems to handle increasing data rates by adding more processing nodes.

Distributed stream processing systems typically consist of several key components. Data ingestion layers are responsible for collecting, buffering, and distributing incoming data streams. Processing engines execute analytical operations in parallel across multiple nodes. State management modules maintain intermediate results required for stateful processing, such as windowed aggregations or pattern detection. Finally, output sinks store processed results or forward them to external systems for further analysis or action.

Parallel processing is fundamental to the operation of distributed stream processing systems. By distributing computation and data across multiple machines, these systems can efficiently utilize available resources and achieve high throughput. However, distributed execution also introduces challenges related to coordination, communication overhead, and fault tolerance, which must be carefully addressed through system design.

Evolution from Batch Processing to Stream Processing

Early data analytics systems were primarily designed around batch processing models. In batch-oriented systems, data is collected over time, stored

persistently, and processed periodically. While this approach is practical for historical analysis and reporting, it is poorly suited for applications that require immediate insights.

The limitations of batch processing, particularly its inability to provide timely responses, led to the development of stream processing systems capable of handling continuous data flows. Early stream processing systems were often limited in scalability and reliability, making them challenging to deploy in large-scale environments. However, advances in distributed computing, cloud infrastructure, and parallel processing techniques have significantly improved the robustness and scalability of modern stream analytics platforms.

The transition from batch to stream processing reflects a broader shift in computing toward real-time, data-driven decision-making. Today, many systems adopt hybrid architectures that combine batch and stream processing to support both historical analysis and real-time insights. Parallel processing has played a crucial role in enabling this transition by providing the computational power to process large-scale data streams efficiently.

Related Work in Parallel Stream Analytics

A substantial body of research has investigated parallel processing techniques for stream analytics. Early work in this area focused on parallel query processing and operator parallelization, drawing inspiration from parallel database systems. These studies explored how analytical operators could be executed concurrently to improve performance.

Subsequent research introduced window-based processing models and stateful stream operators, enabling more complex analytics, such as aggregations, joins, and time-window-based pattern detection. As stream analytics systems grew in scale, research attention shifted toward scalability, fault tolerance, and low-latency processing.

More recent studies have proposed techniques such as dynamic load balancing, adaptive parallelism, and efficient state management to address

challenges in parallel stream processing. Researchers have also examined trade-offs between latency and throughput in different processing models, highlighting the importance of selecting appropriate parallelization strategies based on application requirements. This body of related work provides a strong foundation for understanding modern parallel stream analytics systems.

Summary

This section presents a detailed overview of the fundamental concepts and related work in real-time stream analytics and parallel processing. It discussed the nature of streaming data, stream processing models, parallel processing paradigms, and distributed system architectures. The evolution from batch-oriented systems to real-time stream processing was also examined, along with key research contributions in parallel stream analytics. These foundational concepts provide the necessary background for analyzing system architectures and parallel processing techniques, which are explored in the subsequent sections.

2. SYSTEM ARCHITECTURE FOR REAL-TIME STREAM ANALYTICS

2.1. Architectural Overview

A real-time stream analytics system is fundamentally designed to ingest, process, and analyze continuous flows of data with minimal delay between data arrival and result generation. Unlike traditional data processing architectures, which are typically centralized and batch-oriented, real-time stream analytics architectures must support continuous operation, rapid response times, and dynamic scalability. These requirements make architectural design a critical factor in the effectiveness of any stream analytics platform.

At a high level, real-time stream analytics systems adopt a modular, distributed architecture. Instead of relying on a single centralized processing unit, computation and data management responsibilities are distributed across multiple components and nodes. This design enables the system to handle high-throughput workloads, tolerate failures, and efficiently exploit parallel processing. Each architectural component is responsible for a specific function, and together they form a pipeline that supports continuous data flow.

A typical real-time stream analytics architecture consists of several key components: data sources, a stream ingestion layer, a parallel processing engine, state management mechanisms, window-based processing modules, and output and integration layers. These components are interconnected and operate concurrently, enabling the system to process large-scale streaming data in real time. The effectiveness of the overall architecture depends on how well these components are coordinated and how efficiently parallelism is applied across the system.

Data Sources and Stream Generation

Data sources represent the origin of streaming data in real-time analytics systems. These sources may include physical sensors deployed in industrial or environmental settings, smart and mobile devices, web servers and application

logs, financial transaction platforms, social media feeds, and Internet of Things (IoT) infrastructures. Each source generates data continuously, often at high speed and with varying levels of reliability.

One of the defining challenges associated with data sources is their heterogeneity. Streaming data may be structured, semi-structured, or unstructured, and it may arrive in different formats, sizes, and frequencies. Additionally, data generation rates can fluctuate significantly due to user behavior, environmental conditions, or system events. As a result, real-time stream analytics systems must be designed to handle unpredictable workloads without compromising performance or stability.

To address these challenges, modern architectures incorporate mechanisms that allow multiple data streams to be ingested simultaneously. Parallelism often begins at the data source level, where independent streams are treated as separate input channels. This approach enables the system to scale horizontally and prevents delays caused by bottlenecks at a single input point. Efficient handling of data sources is therefore a foundational requirement for achieving real-time performance.

Stream Ingestion Layer

The stream ingestion layer serves as the interface between data sources and the stream processing engine. Its primary role is to collect incoming data, buffer it, and distribute it to downstream processing components. Because it operates at the front of the processing pipeline, the ingestion layer must handle high input rates while maintaining reliability and low latency.

Scalability is a critical requirement for the ingestion layer, as data arrival rates can spike due to bursts of activity. To support scalability, ingestion systems typically use parallel ingestion mechanisms that partition incoming data streams and distribute them across multiple ingestion nodes. This prevents overload on individual nodes and ensures balanced utilization of system resources.

In addition to data collection and distribution, the ingestion layer often provides important system-level features. These include buffering to absorb short-term spikes in data volume, ordering mechanisms to preserve event sequence where required, and backpressure handling to regulate data flow when downstream components become congested. Fault tolerance is also a key concern, and ingestion systems are often designed to recover quickly from failures without losing data. Through these capabilities, the ingestion layer plays a vital role in maintaining the stability and efficiency of the overall architecture.

Parallel Processing Engine

The parallel processing engine is the core computational component of a real-time stream analytics system. It is responsible for executing analytical operations on incoming data streams using parallel and distributed computing techniques. This engine performs tasks such as filtering, transformation, aggregation, correlation, and pattern detection, all under strict latency constraints.

Processing logic within the engine is typically represented as a directed graph of operators, where each operator performs a specific computation on the data. Data flows from one operator to the next, forming a processing pipeline. Parallelism can be applied at multiple levels within this engine to maximize performance and scalability.

One common approach is operator parallelism, in which multiple instances of the same operator execute concurrently across different partitions of the data stream. Task parallelism allows different operators within the processing graph to execute simultaneously on separate processing nodes. Pipeline parallelism enables different stages of the processing pipeline to operate concurrently on different data items. By combining these forms of parallelism, the processing engine can efficiently handle large-scale data streams while maintaining low latency.

Effective scheduling and resource allocation are essential to the performance of the parallel processing engine. Tasks must be assigned to processing nodes to balance the workload and minimize communication overhead. When properly

designed, the parallel processing engine enables real-time analytics systems to scale efficiently and respond rapidly to incoming data.

State Management in Parallel Systems

State management is a critical architectural concern in real-time stream analytics, particularly for applications that require stateful processing. Stateful operations include aggregations, joins, pattern detection, and window-based computations, all of which rely on maintaining intermediate results across multiple events.

In parallel and distributed environments, managing state becomes significantly more complex. The state must be partitioned and distributed across multiple processing nodes to enable parallel execution while minimizing access latency. Many systems store state locally on processing nodes to improve performance, but this approach requires mechanisms to ensure consistency and correctness.

Checkpointing is a commonly used technique for state management in parallel systems. Periodic snapshots of the system state are saved to stable storage, allowing the system to recover from failures without losing progress. In the event of a failure, the system can restore state from the most recent checkpoint and resume processing. Designing efficient, minimally disruptive checkpointing mechanisms is a major architectural challenge.

Synchronization overhead is another critical consideration. Excessive synchronization among parallel tasks can reduce performance gains. As a result, modern architectures aim to minimize coordination while still ensuring accurate and consistent results. Effective state management is therefore essential for balancing performance, correctness, and fault tolerance in real-time stream analytics systems.

Window-Based Processing Architecture

Window-based processing is a widely used architectural technique in real-time stream analytics for analyzing subsets of data over specific intervals. Rather

than processing an entire unbounded stream at once, windowing divides the stream into manageable segments based on time or event count. Common window types include tumbling, sliding, and casement windows.

Parallel processing significantly enhances window-based analytics by enabling multiple windows to be processed concurrently. Data can be partitioned by keys or window boundaries, enabling windowed computations to be distributed across multiple processing nodes. This approach improves scalability and reduces processing latency.

However, window-based processing introduces additional architectural challenges. Late-arriving data that arrives after a window has closed must be handled carefully to avoid incorrect results. Window alignment across parallel tasks is also critical to ensure consistent computations. Architectural support for event-time processing, watermarking, and window coordination is often required to address these challenges while maintaining real-time performance.

Output and Integration Layer

The output and integration layer is responsible for delivering processed results to downstream systems or end users. Outputs may be written to databases, transmitted to dashboards, forwarded to messaging systems, or used to trigger automated actions such as alerts or control signals. Because real-time analytics often supports time-sensitive decisions, this layer must deliver data with low latency and reliability.

Parallelism in the output layer ensures that result generation and transmission do not become bottlenecks. Multiple output streams can be handled concurrently, allowing the system to maintain high throughput even when delivering results to various destinations. Integration with external systems also requires flexibility, as different applications may use other data formats and have varying delivery requirements.

An efficient design of the output layer ensures that insights from the analytics engine are delivered promptly and reliably, completing the end-to-end real-time processing pipeline.

Scalability and Fault Tolerance

Scalability and fault tolerance are essential properties of real-time stream analytics architectures. As data volumes and processing demands increase, systems must scale without significant redesign. Parallel processing enables horizontal scalability by allowing additional processing nodes to be added dynamically.

Fault tolerance is equally important, as failures are inevitable in distributed environments. Mechanisms such as task replication, checkpointing, and automatic recovery ensure the system continues operating despite hardware or software failures. These mechanisms are closely integrated with parallel processing models to minimize downtime and data loss.

A well-designed architecture balances scalability, performance, and reliability, ensuring continuous operation under varying workloads and failure conditions.

Summary

This section presents a comprehensive examination of system architectures for real-time stream analytics. It discussed the roles of data sources, ingestion layers, parallel processing engines, state management mechanisms, window-based processing, and output integration. The section emphasized how parallel processing underpins scalability, low latency, and fault tolerance across the entire architecture. Understanding these architectural principles provides a strong foundation for analyzing parallel processing techniques and models, which are explored in the subsequent sections.

2.2. Parallel processing techniques for real-time stream analytics

Overview of Parallelism in Stream Processing

Parallel processing underpins modern real-time stream analytics systems. As discussed in earlier sections, streaming data arrives continuously and often at very high speeds, making sequential processing approaches insufficient for meeting strict latency and throughput requirements. Parallelism enables stream analytics systems to divide workloads into smaller units that can be processed simultaneously across multiple computing resources, such as processor cores, machines, or distributed clusters.

In the context of stream processing, parallelism must be carefully designed to account for several unique factors, unlike batch processing, which processes finite, static datasets; stream processing processes unbounded, evolving data. Processing logic often depends on a state that is continuously updated as new events arrive. Additionally, many applications operate under tight time constraints, requiring results to be produced within milliseconds or seconds. These characteristics make the design of parallel stream processing techniques more complex than those used in traditional data analytics.

Effective parallel processing in stream analytics requires balancing multiple objectives, including low latency, high throughput, scalability, and correctness. Tasks must be coordinated to ensure that parallel execution does not compromise data consistency or processing guarantees. As a result, stream processing frameworks adopt specialized parallelization techniques that are tailored to the characteristics of streaming workloads.

Data Parallelism

Data parallelism is one of the most widely adopted parallel processing techniques in real-time stream analytics. In this approach, the incoming data stream is divided into multiple partitions, often called substreams, which can be processed independently and concurrently. Each partition is assigned to a parallel instance of

the same processing logic, allowing the system to handle high data rates efficiently.

Partitioning in data parallelism is commonly based on keys extracted from incoming events. Examples of such keys include user identifiers, sensor IDs, geographic regions, or transaction types. By grouping events with the same key into the same partition, systems can ensure that related data is processed together, which is particularly important for stateful operations such as aggregations and joins.

Data parallelism is highly effective for stateless operations, such as filtering, mapping, and transformation, since these operations do not require shared state across partitions. It can also be applied to stateful operations when state is partitioned in a way that aligns with the data. This alignment allows each parallel task to manage its own portion of the state independently, reducing synchronization overhead.

Despite its advantages, data parallelism introduces challenges related to workload distribution. One common issue is data skew, where some partitions receive significantly more data than others. This imbalance can cause specific processing tasks to become overloaded while others remain underutilized, reducing overall system efficiency. To address this problem, stream analytics systems often employ techniques such as dynamic repartitioning, load-aware scheduling, and adaptive key assignment. These mechanisms help distribute workload more evenly and maintain stable performance under varying data distributions.

Task Parallelism

Task parallelism focuses on executing different processing tasks or operators concurrently within a stream analytics pipeline. Instead of dividing data among identical operations, task parallelism allows distinct operations to run in parallel on separate computing resources. This approach is beneficial in complex analytics workflows that consist of multiple independent or loosely coupled tasks.

In a typical stream-processing application, tasks may include data ingestion, preprocessing, filtering, aggregation, enrichment, machine-learning inference, and output generation. Task parallelism enables these operations to run in parallel, reducing end-to-end processing latency. By overlapping computations across different pipeline stages, systems can better utilize available resources and improve overall responsiveness.

Task parallelism is especially beneficial when different tasks have varying computational requirements. For example, lightweight filtering operations can execute in parallel with more computationally intensive analytics tasks. However, implementing task parallelism requires careful coordination to manage task dependencies. Data produced by one task must be correctly routed to downstream tasks, and synchronization mechanisms may be necessary to ensure that results are generated in the correct order when ordering guarantees are needed.

Designing effective task-parallel stream processing pipelines involves balancing concurrency with coordination overhead. Excessive synchronization can reduce performance gains, while insufficient coordination may lead to incorrect or inconsistent results. As a result, task parallelism is often combined with other parallelization techniques to achieve optimal performance.

Pipeline Parallelism

Pipeline parallelism is a specialized form of task parallelism in which data flows through a sequence of processing stages that operate concurrently. Each stage in the pipeline performs a specific function, and different stages process different data items simultaneously. As one data item moves to the next stage, subsequent items can enter earlier stages, keeping the pipeline continuously active.

In real-time stream analytics, pipeline parallelism improves resource utilization by ensuring that all processing stages run concurrently. This approach is efficient when processing stages have similar computational complexity and can be balanced evenly across resources. Pipeline parallelism reduces idle time and increases throughput by overlapping execution across stages.

However, pipeline parallelism introduces important latency considerations. The overall latency of a data item depends on the slowest stage in the pipeline, often called the bottleneck. If one stage requires significantly more processing time than others, it can limit the performance of the entire pipeline. Consequently, careful performance tuning and load balancing are needed to maximize the benefits of pipeline parallelism.

Pipeline parallelism is commonly used in stream analytics applications that involve multi-stage processing, such as data cleansing, feature extraction, analysis, and result generation. When properly designed, it enables systems to achieve both high throughput and low latency.

Window-Level Parallelism

Window-based processing is a fundamental concept in real-time stream analytics, enabling computations over finite subsets of an otherwise unbounded data stream. Window-level parallelism exploits the independence of these subsets to allow parallel execution of window-based computations.

In many cases, windows can be processed independently, making them well-suited for parallelization. For example, tumbling windows that do not overlap can be assigned to different processing tasks and computed concurrently without coordination. This approach enables efficient scaling of time-based aggregations and summaries.

Sliding windows and session windows present additional challenges, as they may overlap or depend on event timing. Overlapping windows require careful coordination to manage shared data and avoid redundant computation. To address this, stream processing systems often use incremental computation techniques, where results are updated as new events arrive rather than recomputed from scratch. Parallel processing frameworks also employ window partitioning strategies to distribute window computations across multiple tasks while maintaining correctness.

Window-level parallelism is significant for applications such as trend analysis, anomaly detection, and time-series analytics, where timely insights are critical. By enabling multiple windows to be processed concurrently, systems can deliver real-time results even under high data rates.

Operator Parallelism

Operator parallelism refers to replicating processing operators so that multiple instances of the same operator can execute concurrently across different data partitions. Each operator instance performs identical computations but operates on a distinct subset of the input stream. This technique enables fine-grained scalability and is a key mechanism for handling large-scale streaming workloads.

In practice, operator parallelism is often combined with data parallelism. Data is first partitioned into multiple streams, and each partition is assigned to a separate operator instance. As data volume increases, additional operator instances can be dynamically deployed to distribute the workload evenly.

Managing state in operator-parallel systems presents significant challenges, particularly for stateful operators. Each operator instance must maintain its own state, and systems must ensure that state updates are applied correctly and consistently. Synchronization and coordination mechanisms are required to prevent inconsistencies while minimizing performance overhead.

Operator parallelism is a powerful technique for achieving horizontal scalability, but its effectiveness depends on careful design of state management, data partitioning, and fault tolerance mechanisms.

Hybrid Parallel Processing Models

In real-world applications, no single parallel processing technique is sufficient to meet all performance and scalability requirements. As a result, modern real-time stream analytics systems employ hybrid parallel processing models that combine multiple forms of parallelism. For example, a system may use data

parallelism to partition incoming streams, pipeline parallelism to structure processing stages, and window-level parallelism to support time-based analytics.

Hybrid models provide flexibility, enabling systems to adapt to varying workloads and application requirements. By combining different parallelization strategies, systems can optimize performance across multiple dimensions, including latency, throughput, and resource utilization. However, hybrid models also increase system complexity, as they require sophisticated scheduling, coordination, and monitoring mechanisms.

Designing effective hybrid parallel processing models involves careful consideration of application characteristics, workload patterns, and system constraints. When implemented correctly, hybrid models offer the best balance between performance and scalability in real-time stream analytics.

Performance Considerations

While parallel processing provides substantial performance benefits, it also introduces overhead that can limit its effectiveness if not carefully managed. Communication between parallel tasks, synchronization delays, and state management overhead can all reduce the gains achieved through parallelization.

Performance optimization strategies in stream analytics systems focus on minimizing unnecessary communication, optimizing data partitioning schemes, and reducing synchronization overhead. In-memory processing techniques are widely used to reduce data access latency and improve throughput. Monitoring and adaptive tuning mechanisms enable systems to adjust parallelism levels in response to changing workloads dynamically.

Achieving optimal performance requires continuous evaluation and tuning of parallel processing strategies. Systems must balance the benefits of increased parallelism with the costs of coordination and resource contention.

Summary

This section presents a detailed examination of parallel processing techniques used in real-time stream analytics. It discussed data parallelism, task

parallelism, pipeline parallelism, window-level parallelism, operator parallelism, and hybrid parallel processing models. The section also highlighted key performance considerations and trade-offs associated with parallel execution. Together, these techniques form the foundation of scalable and efficient real-time stream analytics systems. Understanding their strengths and limitations is essential for designing high-performance analytics solutions that meet modern real-time data processing demands.

3. ALGORITHMS AND FRAMEWORKS FOR PARALLEL STREAM PROCESSING

3.1. Stream Processing Algorithms

Real-time stream analytics algorithms differ fundamentally from traditional batch-processing algorithms due to the continuous and unbounded nature of streaming data. In batch analytics, algorithms operate on static datasets with finite size, often allowing multiple passes over the data. In contrast, streaming data arrives continuously and must be processed incrementally, often under strict latency constraints. As a result, stream processing algorithms must be designed to operate with limited memory, process events in real time, and produce timely results that support immediate decision-making.

Parallel processing is central to the scalability and efficiency of these algorithms. By leveraging multiple processing units simultaneously, streaming algorithms can maintain high throughput and low latency even when dealing with massive data volumes. Parallel execution is critical when applications require processing millions of events per second, such as in financial trading platforms, smart city monitoring systems, or industrial IoT networks.

Streaming algorithms are broadly categorized into four main classes: filtering, aggregation, pattern detection, and machine learning–based algorithms. Each class addresses specific types of analytics requirements and benefits from parallel execution in distinct ways. Filtering algorithms handle selection tasks, aggregation algorithms summarize or condense data, pattern detection algorithms identify sequences or correlations within the stream, and machine learning algorithms provide predictive or adaptive analytics. Understanding these classes is crucial for designing high-performance, parallel stream analytics systems.

Parallel Filtering and Transformation Algorithms

Filtering and transformation are fundamental building blocks in stream analytics pipelines. Filtering algorithms select events that meet specific criteria, such as network packets containing anomalies or financial transactions exceeding a threshold. Transformation algorithms, on the other hand, convert raw data into meaningful formats or enrich events with additional context, such as converting timestamps, extracting features, or adding geographical metadata.

These operations are typically stateless, meaning that each event can be processed independently without relying on prior events. This property makes them particularly well-suited for data parallelism. In parallel stream processing, the incoming data stream can be partitioned, with multiple parallel tasks independently filtering or transforming their assigned partitions.

Operator replication is another technique used to enhance parallel filtering and transformation. Multiple instances of the same operator can run concurrently across separate data partitions, enabling systems to handle high event rates without introducing latency. For example, in an online recommendation system, filtering user interactions by relevance and transforming them into feature vectors can be distributed across multiple parallel tasks, enabling personalized recommendations to be generated in real time.

Despite their stateless nature, parallel filtering and transformation algorithms must still manage practical challenges such as load balancing and resource utilization. Uneven arrival rates of data can cause specific processing tasks to become overloaded while others remain underutilized. Adaptive partitioning and dynamic scheduling are therefore commonly implemented to maintain consistent throughput.

Parallel Aggregation Algorithms

Aggregation algorithms compute summary statistics over streaming data, including counts, sums, averages, minimum and maximum values, and more complex metrics such as quantiles and histograms. These algorithms are inherently

stateful, as they maintain intermediate results over time or within specific event windows.

Parallel aggregation is achieved primarily through data partitioning. Incoming events are partitioned according to keys, such as user IDs, device identifiers, or sensor types, and local aggregates are maintained within each partition. Once partial aggregates are computed, they are merged to produce global results. This hierarchical approach reduces communication overhead and allows aggregation tasks to scale efficiently across multiple processing units.

However, parallel aggregation introduces several challenges. Maintaining state consistency across partitions is critical to ensure correct results. Window alignment, where aggregates are computed over time-based or count-based windows, must also be carefully managed, especially when windows overlap or events arrive late. Additionally, fault-tolerance mechanisms such as checkpointing must be integrated to enable recovery from node failures without losing intermediate results.

A practical example is traffic monitoring, where sensors along a highway continuously count vehicles. By partitioning data by sensor location and computing local aggregates in parallel, the system can provide timely congestion analysis across the entire network. The combination of regional and global aggregation ensures both scalability and accuracy.

Pattern Detection and Complex Event Processing

Pattern-detection algorithms identify sequences or combinations of events that match predefined criteria. These algorithms are crucial in applications such as fraud detection, cybersecurity monitoring, industrial equipment failure prediction, and automated alerting systems. Pattern detection often requires maintaining temporal and logical relationships between events, making it more complex than simple filtering or aggregation.

Parallel processing enhances pattern detection by dividing the stream into segments that can be analyzed concurrently. Task parallelism is frequently used to

assign different detection rules to separate processing units, while window-level parallelism allows temporal patterns to be detected within specific intervals. This combination ensures timely detection even under high event rates.

Complex Event Processing (CEP) systems extend pattern detection by enabling more sophisticated event correlation, temporal reasoning, and hierarchical pattern recognition. CEP frameworks leverage parallelism to execute multiple pattern-matching tasks concurrently and maintain state across distributed nodes. A significant challenge in CEP is ensuring correct event ordering and synchronization across parallel tasks, as misaligned events can lead to false positives or missed detections.

For instance, in financial fraud detection, sequences of suspicious transactions across multiple accounts must be monitored in real time. Parallel pattern detection allows the system to evaluate various transaction sequences simultaneously, ensuring rapid identification of potential fraudulent behavior.

Machine Learning Algorithms for Streaming Data

Machine learning (ML) algorithms are increasingly applied to streaming data for real-time prediction, classification, anomaly detection, and adaptive decision-making. Unlike traditional batch ML, streaming ML algorithms must continuously update models as data distributions evolve, often without revisiting past events. This incremental nature makes them well-suited for parallel execution.

Parallelism in streaming ML can be implemented in several ways:

1. Model Partitioning – Different components of a model, such as layers of a neural network, are distributed across multiple processing units for concurrent execution.

2. Parallel Feature Extraction – Features are computed in parallel from raw events before feeding them into the learning model.

3. Incremental Learning – Model parameters are updated concurrently across partitions of the stream, enabling the system to adapt to changing data patterns.

Despite these advantages, parallel machine learning in streaming environments introduces challenges. Ensuring model consistency across parallel updates, minimizing communication overhead, and controlling prediction latency are critical concerns. For example, in real-time recommendation engines, inconsistent model updates can result in incorrect recommendations, while excessive synchronization delays can increase response time.

Emerging research continues to explore novel parallelization strategies for streaming ML, including federated learning across distributed nodes, approximate model updates, and asynchronous parallel training techniques. These strategies aim to maximize throughput while maintaining prediction accuracy and model stability.

Stream Processing Frameworks

To simplify the development and deployment of real-time stream analytics applications, several stream processing frameworks have been introduced. These frameworks provide abstractions for defining processing pipelines, managing parallel execution, handling fault tolerance, and supporting stateful operations.

Key features of modern frameworks include:

- Support for parallel and distributed execution: Frameworks automatically partition data and distribute tasks across multiple nodes.
- Built-in state management and checkpointing: Stateful operators are supported with mechanisms for consistent state updates and recovery.
- Window-based processing capabilities: Time- or count-based windows are natively supported, with optimizations for overlapping or sliding windows.
- Scalability and fault tolerance mechanisms: Systems can dynamically scale out to additional nodes and recover from failures without data loss.

Popular stream processing frameworks include Apache Flink, Apache Spark Structured Streaming, Apache Storm, and Apache Samza. Each framework provides unique features and optimizations tailored to different application scenarios. For instance, Apache Flink offers low-latency, exactly-once state

consistency, making it suitable for financial or industrial monitoring applications. At the same time, Spark Structured Streaming provides strong integration with batch analytics pipelines and large-scale data warehouses.

Frameworks abstract much of the complexity of parallel processing, allowing developers to focus on application-specific logic rather than low-level system concerns such as resource scheduling, checkpointing, or operator replication.

Algorithm–Framework Interaction

The effectiveness of parallel stream processing depends heavily on the interaction between the algorithms and the underlying framework. Algorithms must be designed to exploit the framework's parallelism. In contrast, frameworks must provide flexible, efficient execution models that accommodate stateful, stateless, and hybrid processing tasks.

For example, stateless algorithms such as filtering or transformation can be easily scaled using simple data parallelism. In contrast, stateful algorithms such as aggregation or pattern detection require careful alignment of state partitions with data partitions to ensure correctness. Frameworks that support dynamic load balancing, operator replication, and stateful checkpointing make it easier to implement parallel algorithms efficiently.

Understanding this interaction is crucial for achieving high-performance analytics. Poor alignment between algorithm design and framework capabilities can lead to underutilized resources, increased latency, or incorrect results. Conversely, well-integrated algorithms and frameworks can deliver near-linear scalability and robust, fault-tolerant processing.

Summary

This section provides an in-depth examination of algorithms and frameworks used in parallel real-time stream analytics. It covered key classes of algorithms,

including filtering, transformation, aggregation, pattern detection, and machine learning, and explained how parallelism enhances their performance.

Additionally, it discussed popular stream processing frameworks and the critical interplay between algorithm design and framework capabilities. Together, these elements form the backbone of scalable, low-latency, and reliable real-time analytics systems, enabling applications across diverse domains such as finance, healthcare, smart cities, cybersecurity, and industrial automation.

3.2. Challenges and limitations of parallel stream analytics

Real-time stream analytics systems powered by parallel processing are critical for modern data-driven applications, yet they are not without significant challenges. Parallelism enables high throughput, scalability, and low-latency processing, but it also introduces complexities that can impact system performance, reliability, and manageability. In this section, we examine these challenges in detail, highlighting key limitations and considerations for designing robust real-time analytics platforms.

Scalability Challenges

Scalability is one of the principal motivations for applying parallel processing to real-time stream analytics. In theory, adding more processing nodes should proportionally increase system throughput. In practice, however, achieving linear scalability is rarely straightforward. Several factors limit performance as the system grows:

1. Coordination Overhead: As the number of nodes or cores increases, the need to coordinate processing across them becomes more significant. Task scheduling, load balancing, and state synchronization introduce communication overhead that can negate the benefits of adding resources.

2. Communication Latency: Distributed stream processing systems require data to be exchanged between nodes. Network latency, serialization/deserialization costs, and protocol overhead can slow down data

movement, particularly when events must traverse multiple nodes for stateful computations.

3. Resource Contention: Multiple parallel tasks may compete for shared resources, such as CPU, memory, disk, or network bandwidth. High contention can cause processing delays, leading to missed deadlines in time-sensitive applications.

A particularly challenging issue is data skew, in which specific partitions of the input stream receive disproportionately large volumes of data. This uneven distribution can cause specific processing tasks to become bottlenecks, while other tasks remain underutilized. For example, in a social media analytics application, popular hashtags or topics may generate a flood of events for a small subset of partitions, overwhelming their corresponding processing nodes. Addressing data skew requires adaptive strategies, such as dynamic stream repartitioning, load-aware scheduling, or predictive partitioning based on historical data trends. However, implementing these solutions adds complexity to system design and may introduce additional overhead, which itself can affect performance.

Another consideration is horizontal scalability. While distributed systems can theoretically scale out by adding more nodes, the cost, coordination, and management overhead grow with system size. Cloud-based solutions offer elastic scaling, but dynamic node provisioning introduces transient performance variability and may require careful orchestration to maintain processing guarantees.

Latency Constraints

Many real-time stream analytics applications are latency-sensitive, requiring results within milliseconds or seconds of event arrival. Examples include fraud detection in financial systems, emergency alerts in smart cities, anomaly detection in industrial equipment, and recommendation systems for e-commerce platforms. Meeting these stringent latency requirements is a key challenge in parallel processing environments.

Parallelism can reduce processing latency by enabling the simultaneous execution of multiple tasks. However, it also introduces latency due to communication and synchronization overheads. In distributed settings, data must be transmitted between nodes, serialized, deserialized, and coordinated among parallel tasks. Each of these steps adds to the overall end-to-end latency.

Moreover, task dependencies can exacerbate latency. Certain computations depend on the results of prior tasks or on state maintained across nodes. Synchronization mechanisms, such as barriers, locks, or consensus protocols, ensure correctness but may delay downstream processing.

Network topology, inter-node bandwidth, and congestion also affect latency. In geographically distributed deployments, the physical distance between nodes can introduce additional delays, making it difficult to maintain consistent low-latency processing across the system.

Optimizing latency in parallel stream analytics requires careful system design. Strategies include minimizing inter-task communication, using in-memory processing, optimizing task placement based on network proximity, and employing incremental computation techniques that reduce the need to recompute results from scratch. However, achieving a balance between latency and throughput remains a fundamental challenge, as optimizations that reduce latency may increase resource usage or reduce overall system efficiency.

State Management Complexity

Stateful processing is at the heart of many real-time analytics tasks, including aggregations, joins, pattern detection, and window-based computations. Maintaining and managing this state in a parallel, distributed environment is a complex, error-prone task.

State in parallel systems must satisfy several requirements:

1. Consistency: Updates to shared state across parallel tasks must be consistent, even when events are processed out of order or nodes fail.

2. Fault Tolerance: Systems must recover state correctly after failures to avoid incorrect analytics results.

3. Efficiency: Maintaining state should not impose significant overhead that degrades throughput or increases latency.

To achieve these goals, stream analytics frameworks often employ checkpointing and state partitioning techniques. State is periodically saved to persistent storage, enabling recovery in case of node failures. However, frequent checkpoints introduce performance overhead, while infrequent checkpoints increase recovery time and the risk of data loss.

Distributed state management also requires careful partitioning to ensure that parallel tasks can efficiently access and update the relevant portions of state. Poorly designed state partitioning can lead to bottlenecks and increased synchronization costs. For example, in a real-time recommendation system, user session data must be partitioned such that updates and lookups can occur in parallel without conflicts.

Late-arriving events and out-of-order processing further complicate state management. In windowed computations, events may arrive after the window has been partially processed, requiring updates to previously computed results. Handling these scenarios efficiently while maintaining correctness is an ongoing research challenge in stream analytics.

Fault Tolerance and Reliability

In distributed parallel systems, failures are inevitable. Hardware faults, software errors, network outages, and resource exhaustion can all disrupt stream processing. Real-time analytics systems must tolerate such failures without data loss or significant downtime.

Fault tolerance mechanisms, such as operator replication, checkpointing, and log-based recovery, are widely used to maintain reliability. In operator replication, multiple instances of a processing task are run concurrently, allowing another instance to take over if one fails. Checkpointing periodically saves the system

state, enabling recovery without recomputing from scratch. Log-based recovery involves storing event streams in durable storage so that processing can resume from the last consistent point.

Despite their effectiveness, these mechanisms introduce performance overhead. Maintaining replicas consumes additional resources, checkpointing interrupts processing, and log replay can increase latency. Striking a balance between fault tolerance and performance is a key challenge for designers of parallel stream analytics systems.

Applications with strict reliability requirements, such as financial transaction monitoring or autonomous vehicle systems, require high-availability guarantees. Ensuring minimal downtime in these systems adds further complexity to system architecture and resource management.

Synchronization and Consistency Issues

Executing tasks simultaneously requires careful coordination to ensure accurate results. Methods like locks, barriers, and agreement procedures help make sure that shared information is updated correctly and that related calculations happen in the right sequence.

The choice of consistency approach greatly influences how well the system performs and how complex it is. Ensuring strong consistency makes sure everything is correct but can lead to higher delays and more coordination work. On the other hand, more relaxed consistency methods, like eventual consistency, tend to boost performance but might result in brief periods of inconsistency.

For example, in a distributed system that combines data from multiple sources, updates to counters need to be shared among all the connected nodes. Ensuring that every node shows the same value at all times requires ongoing communication and coordination, which can slow down performance. Allowing nodes to temporarily have different values and to synchronize later can increase speed, but might lead to short periods of inaccuracy.

Designers must carefully choose consistency models based on application requirements, balancing correctness, latency, and system complexity.

Resource Management and Cost

Parallel stream analytics systems are typically deployed on distributed infrastructure, including cloud clusters, on-premises servers, or hybrid environments. While additional nodes and cores improve scalability, they also incur operational costs, including computing resources, energy consumption, and maintenance.

Efficient resource management is crucial to balance performance and cost. Over-provisioning ensures low latency and high throughput but increases expenses, while under-provisioning may lead to performance degradation, missed deadlines, or dropped events.

Adjusting the number of active processing nodes based on workload changes helps manage resource demands and maintain performance. However, setting up these adjustments can make operations more complicated. For instance, in cloud environments, scaling up too quickly can increase costs without providing significant performance improvements, while scaling too slowly might cause delays during traffic spikes.

Resource management also relates to handling unexpected issues. Creating copies and saving system states requires extra memory and processing work, so careful planning is needed to keep the system affordable while ensuring it stays reliable.

Additional Challenges

Beyond the primary limitations discussed, several other challenges affect parallel stream analytics systems:

1. Data Heterogeneity: Streams often originate from diverse sources with different formats, units, and schemas. Transforming and normalizing these heterogeneous streams in parallel can introduce additional complexity.

2. Backpressure Handling: Systems need to deal with situations where they can't process data fast enough. Backpressure signals are sent upstream to control the flow, but managing these signals efficiently in environments with many processes requires careful planning.

3. Security and Privacy: Streaming systems often handle sensitive information, such as financial transactions or personal health records. Protecting this data through secure processing methods, including encryption, access controls, and privacy measures, makes things more complicated.

4. Monitoring and Debugging: Observing and diagnosing performance or correctness issues in parallel, distributed systems is challenging due to their dynamic and non-deterministic behavior.

Summary

This section has examined the main challenges and limitations of analyzing data simultaneously across multiple streams, highlighting that while working in parallel allows for growth, quick responses, and handling large amounts of data, it also adds complexity in various ways. Growing systems can face difficulties due to coordinating different parts, uneven data distribution, and competition for resources. Reducing delays requires careful tuning of communication and processing steps. Managing state, ensuring reliability, keeping processes synchronized, and maintaining accuracy are additional hurdles in building dependable systems. Finally, managing resources and controlling costs are important factors in practical implementations.

Understanding these limitations is essential for designing effective stream processing architectures, selecting appropriate parallel processing techniques, and evaluating existing solutions. Addressing these challenges remains an active area of research, with ongoing efforts to improve adaptive scheduling, state management, consistency models, and fault-tolerant frameworks.

3.3. Statistical analysis of large data sets

In the previous sections, the results and graphs of a regular dataset were demonstrated, the dataset was collected according to the technical specifications. The results of machine learning methods did not meet the needs of the customer. So it was decided to collect the data in a different way, as a result, a large dataset was obtained1815696 – expired. You must first connect to the librarydask and allocate the amount of memory that is needed, in this case 8 GB. The results are shown in Figure 3.1. The data are shown in Fig. 3.2.

```
client = Client(n_workers=2, threads_per_worker=1, memory_limit='4GB',processes=False)
client
```

Client

- **Scheduler:** inproc://172.28.0.2/55/22

Cluster

- **Workers:** 2
- **Cores:** 2
- **Memory:** 8.00 GB

Figure 3.1 – Result of connecting to the library

	0	1	2	3	4
x1	14136.000000	21895.000000	16245.000000	21811.000000	20322.000000
x2	8773.000000	10789.000000	7822.000000	14270.000000	12093.000000
x3	0.937134	0.526545	0.422646	0.755819	0.670338
x4	0.929526	0.434938	0.774747	0.822779	0.536706
x5	15.771608	11.431559	10.652850	10.800585	15.789350
x6	3.125166	3.127802	3.445335	2.178386	4.792278
x7	0.922892	0.518313	0.405828	0.742631	0.658859
x8	0.744948	0.331276	0.693582	0.674253	0.404934
x9	15.707006	11.395277	10.624442	10.748601	15.743230
x10	-0.359916	0.599410	1.087976	-0.206522	1.299158
x11	230.160848	103.589098	69.569617	173.912968	147.385963
x12	181.955516	96.973361	170.712982	167.314556	112.300623
x13	74.270311	283.590717	253.445205	146.779673	214.344640
x14	479.626285	226.649362	255.879338	377.384050	296.904523
x15	0.000000	0.000000	0.000000	0.000000	0.000000
x16	0.000000	0.000000	0.000000	0.000000	0.000000
target1	111.000000	111.000000	111.000000	111.000000	111.000000
target2	121.000000	121.000000	121.000000	121.000000	121.000000

Figure 3.2 – Building a new data set

After the data was collected and displayed, the next step was the primary analysis, that is, to understand that the data does not have gaps or other characters. First, we will check the list in percentage terms by the proportion of missing records for each feature. The result is shown in Fig.3.3.

```
x1 - 0.0%
x2 - 0.0%
x3 - 0.0%
x4 - 0.0%
x5 - 0.0%
x6 - 0.0%
x7 - 0.0%
x8 - 0.0%
x9 - 0.0%
x10 - 0.0%
x11 - 0.0%
x12 - 0.0%
x13 - 0.0%
x14 - 0.0%
x15 - 0.0%
x16 - 0.0%
target1 - 0.0%
target2 - 0.0%
```

Figure 3.3 – Checking for missing sample values

Next, we will check for text values. The result is shown in Fig. 3.4.

```
x1         0
x2         0
x3         0
x4         0
x5         0
x6         0
x7         0
x8         0
x9         0
x10        0
x11        0
x12        0
x13        0
x14        0
x15        0
x16        0
target1    0
target2    0
```

Figure 3.4 – Non-standard missing values

The next step was to create a target sample distribution graph, blue.– workout, orange–testing, green common. How similar the target variables are to each other. The density plot is shown in Fig. 3.5.

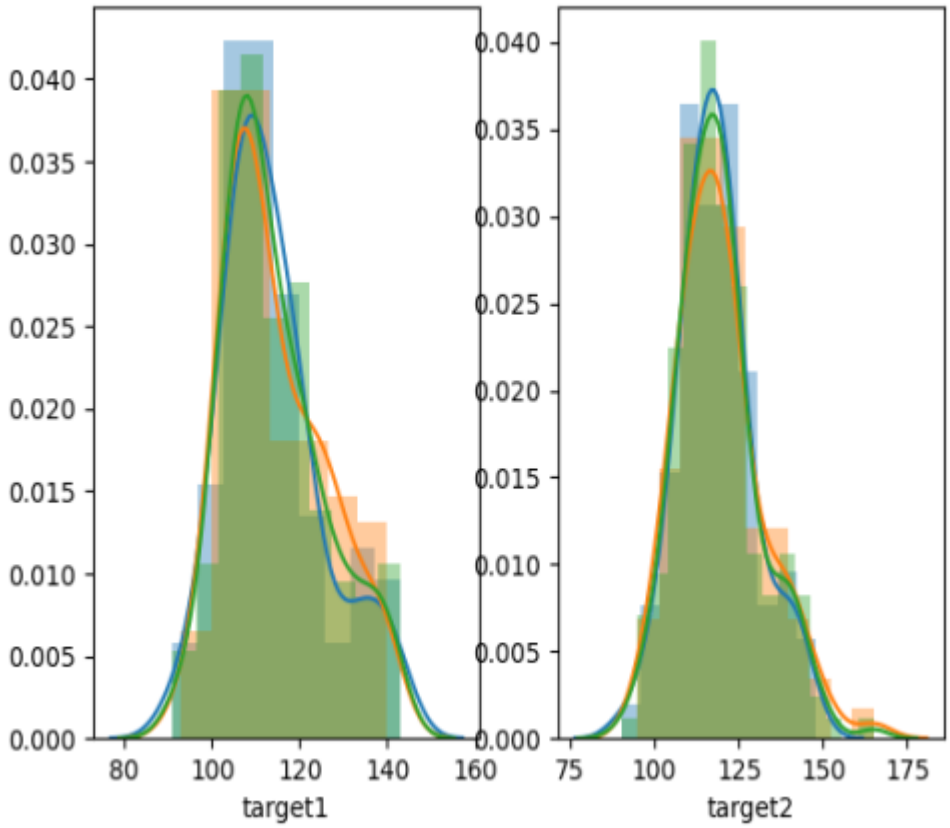

Figure 3.5 – Density plot of the distribution of target variables

The scatter plot shown in Figure 3.6 shows us how dependent the data is on each other, as well as the spread of the data. This plot can help us determine whether we need to clean the data with other methods, or apply a different approach to the data.

In order to make sure how often certain values occur, we construct a frequency diagram. Each column of the histogram shows the frequency of the sample value falling within the value interval – the higher the bar, the more likely the corresponding indicator values are. The histogram is shown in Figure 3.7.

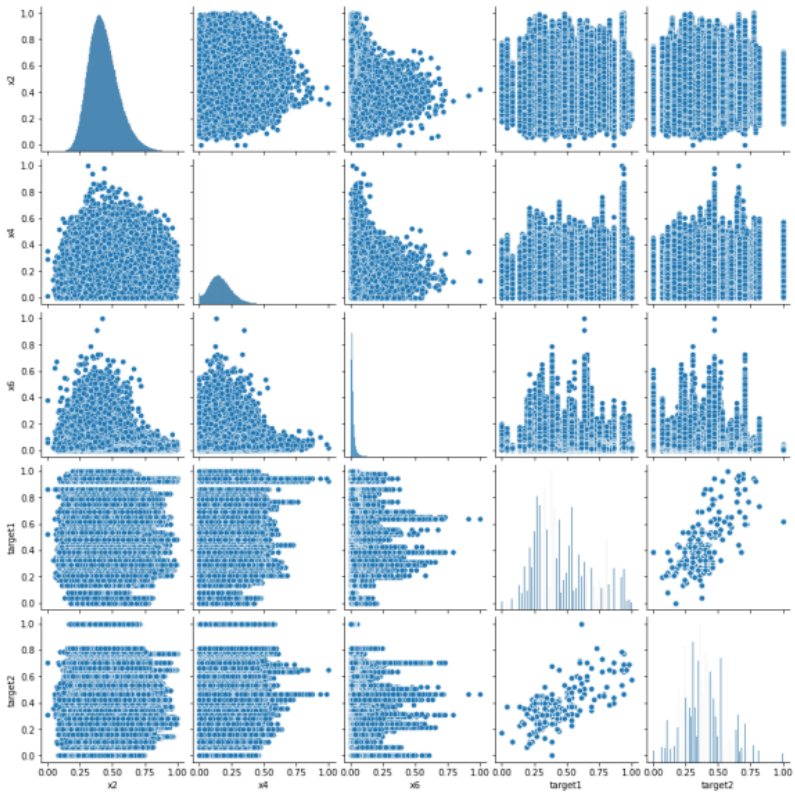

Figure 3.6 – Scatter plots of dependent and explanatory variables

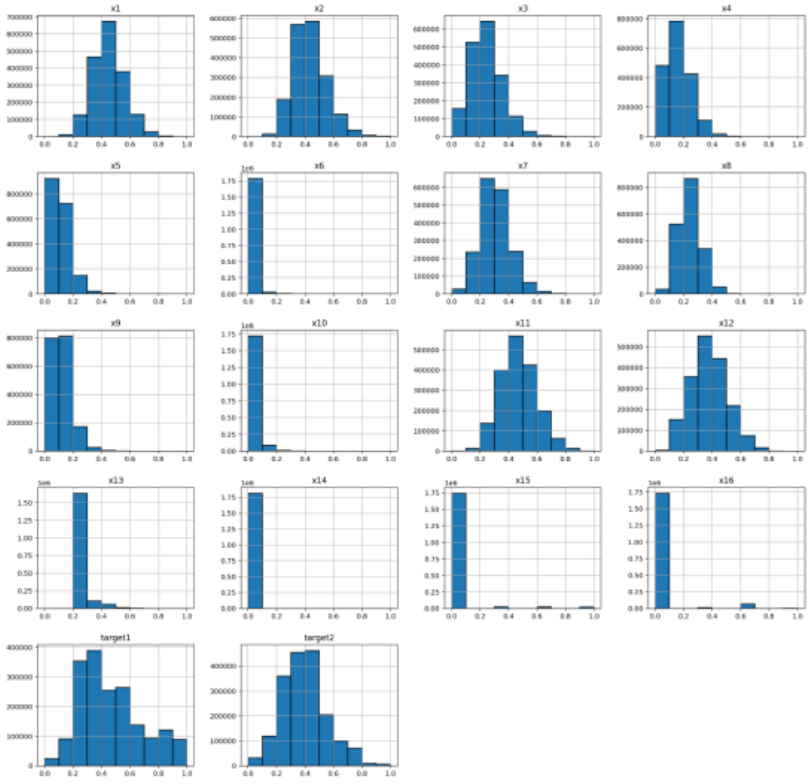

Figure 3.7–Data distribution histograms

The next step was to use a correlation map. Here we can see how the parameters (features) depend on each other. Based on the previous graphs, our assumptions are correct, where 1 in the previous graph was the perimeter distribution. The map is shown in Fig.3.8.

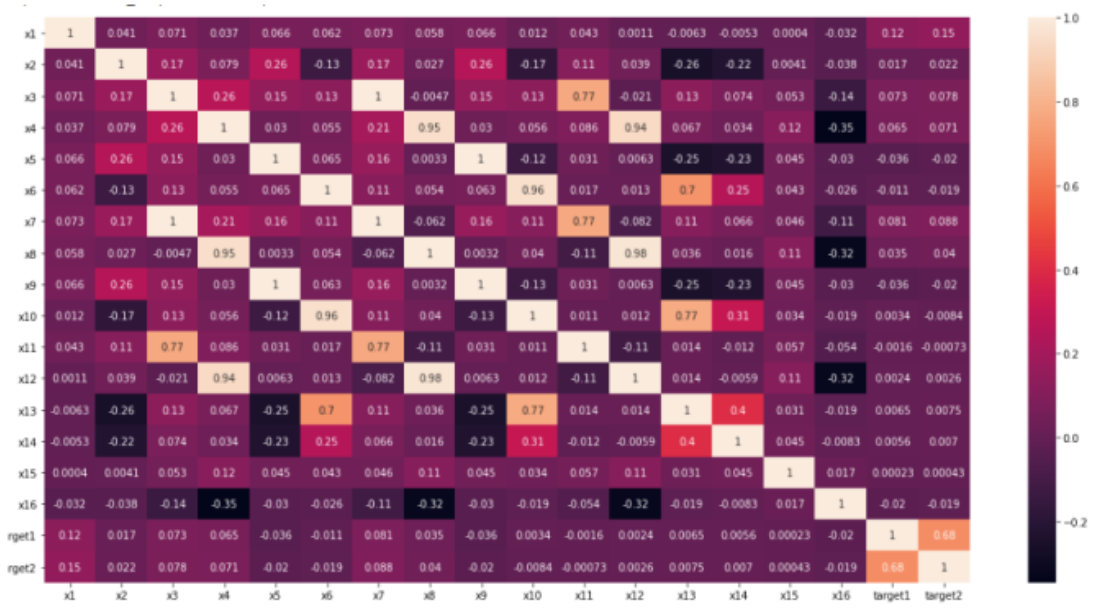

Figure 3.8 – Correlation map – matrix

After the correlation map, it was decided to remove those variables that have a very high correlation, i.e. they do not carry any informative value. The following variables were removed: x7, x5, x8, x10, x14, x15, x16.

After removing the data, we obtained a sample that looks like the one shown in Fig. 3.9.

Data has two types such as integer and floating point numbers. To make them look the same, standardization has been applied. The idea behind standardization is that it transforms the data so that its distribution will have a mean of 0 and a standard deviation of 1. Given the distribution of the data, each value in the data set will be subtracted from the sample mean and then divided by the standard deviation of the entire data set. Mathematical explanation:

	x2	x3	x4	x6	x9	x11	x12	x13	target1	target2
0	8773	0.937134	0.929526	3.125166	15.707006	230.160848	181.955516	74.270311	111	121
1	10789	0.526545	0.434938	3.127802	11.395277	103.589098	96.973361	283.590717	111	121
2	7822	0.422646	0.774747	3.445335	10.624442	69.569617	170.712982	253.445205	111	121
3	14270	0.755819	0.822779	2.178386	10.748601	173.912968	167.314556	146.779673	111	121
4	12093	0.670338	0.536706	4.792278	15.743230	147.385963	112.300623	214.344640	111	121
...	...	...	...	...	...	...	...	...	...	...
1815691	13855	0.444579	0.944804	16.943530	9.929977	61.825721	158.095613	3942.412923	106	105
1815692	8183	0.771922	0.674117	8.652344	2.028285	156.530470	106.108712	2434.390274	106	105
1815693	11037	0.626786	0.886155	7.217694	14.590242	114.204442	141.366510	1036.541148	106	105
1815694	8055	0.493443	0.724412	13.149114	8.804448	80.185968	122.784254	3032.202558	106	105
1815695	7353	0.353857	1.341039	9.910459	3.079820	29.159254	217.796809	2656.283451	106	105

1815696 rows × 10 columns

Figure 3.9 – Data after removing non-informative variables

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{3.1}$$

where and are given as the minimum and maximum allowable values, for default $min = 0$, $max = 1$. Standardized data are shown in Fig. 3.10.

x1	0.375985	0.641868	0.448256	0.638990	0.587965	0.581214	0.649853	0.605990
x2	0.267968	0.329546	0.238920	0.435872	0.369376	0.234002	0.282843	0.285226
x3	0.355502	0.181139	0.137017	0.278504	0.242203	0.124837	0.183350	0.265253
x4	0.253043	0.102171	0.205828	0.220480	0.133215	0.120272	0.159996	0.184107
x5	0.086409	0.062479	0.058186	0.059000	0.086507	0.086044	0.194558	0.052087
x6	0.009036	0.009044	0.009992	0.006208	0.014015	0.012862	0.044172	0.014173
x7	0.418574	0.257907	0.213236	0.346988	0.313721	0.204234	0.252021	0.332577
x8	0.321932	0.212488	0.308343	0.303229	0.231976	0.235338	0.263424	0.273603
x9	0.096200	0.072690	0.068487	0.069164	0.096397	0.096044	0.203307	0.062341
x10	0.057737	0.060438	0.061813	0.058169	0.062408	0.061361	0.078584	0.066458
x11	0.741563	0.429539	0.345674	0.602901	0.537507	0.330298	0.410736	0.567871
x12	0.535364	0.321671	0.507093	0.498548	0.360212	0.366903	0.417575	0.438845
x13	0.252685	0.265898	0.263995	0.257262	0.261527	0.260576	0.284900	0.272603
x14	0.002307	0.001779	0.001840	0.002094	0.001926	0.001961	0.001328	0.001592
x15	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
x16	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
target1	0.384615	0.384615	0.384615	0.384615	0.384615	0.384615	0.384615	0.384615
target2	0.413333	0.413333	0.413333	0.413333	0.413333	0.413333	0.413333	0.413333

Figure 3.10 – Standardized data

3.4. Results of machine learning models

Next, we build machine learning models. Since we have come to the point where we have a regression problem, the main task is to predict the target variables. We will build the same models that were built on past data. Since the unsatisfactory results made it clear that there is not enough data, the models are retrained. Therefore, it was decided to use such methods as: linear regression, random forest, decision tree and determine the best predictive model. All methods will be considered from the point of view of regression. The following metrics of predictive quality were used, as well as the coefficient of determination R^2:

– coefficient R^2:

$$R^2 = 1 - \frac{u}{v} = (y - y')^2 \quad (3.2)$$

where u –residual sum of squares;

v - total sum of squares.

– root mean square error:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad (3.3)$$

– mean absolute error:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|x_i - x| \quad (3.4)$$

– average absolute percentage error:

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \quad (3.5)$$

– average percentage error:

$$MPE = \frac{100\%}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{y_i}\right) \quad (3.6)$$

Results of the error of the predictive quality, as well as the coefficient of determination R^2, given for models such as, linear regression, trees solutions, random forest. The results are shown in Table 3.1.

In addition to visualizing the predictive quality, regression trees were constructed graphically for both target variables. It is graphically that we can understand how the distribution is going. The tree for target1 is shown in Fig. 3.15. The tree for target2 is shown in Fig. 3.16. As you can see, we take a subset of the data and decide how best to divide the subset.

Our initial subset was the entire data set, and we divided it according to the rule X <= 0.258 . Then for each subset we performed additional splitting until they could correctly predict the target variable without adhering to the constraint on the depth of the tree.

Table 3.1 - Results of forecast error of the methods used

Method	Error forecast qualities	Target 1 (target variable)		Target 2 (target variable)	
		train	test	train	test
Linear regression	MSE	0.02	0.04	0.01	0.02
	MAE	0.13	0.17	0.9	0.11
	MAPE	14.20	16.67	14.28	16.41
	SCORE	0.13	0.10	0.13	0.13
Decision tree	MSE	0.01	0.02	0.0	0.01
	MAE	0.06	0.11	0.03	0.07
	MAPE	7.49	10.63	3.15	6.55
	SCORE	0.76	0.53	0.75	0.58
Random forest	MSE	0.02	0.04	0.01	0.03
	MAE	0.01	0.03	0.02	0.04
	MAPE	2.54	3.46	1.0	1.96
	SCORE	0.90	0.88	0.93	0.91

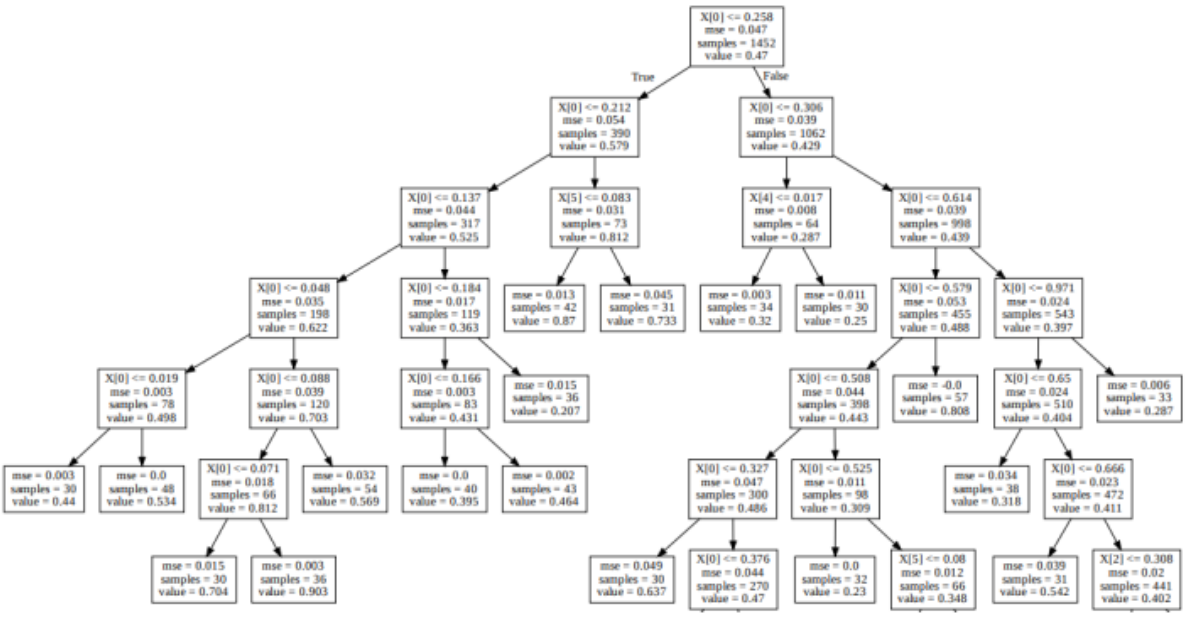

Figure 3.14 – Tree for target variable (target1)

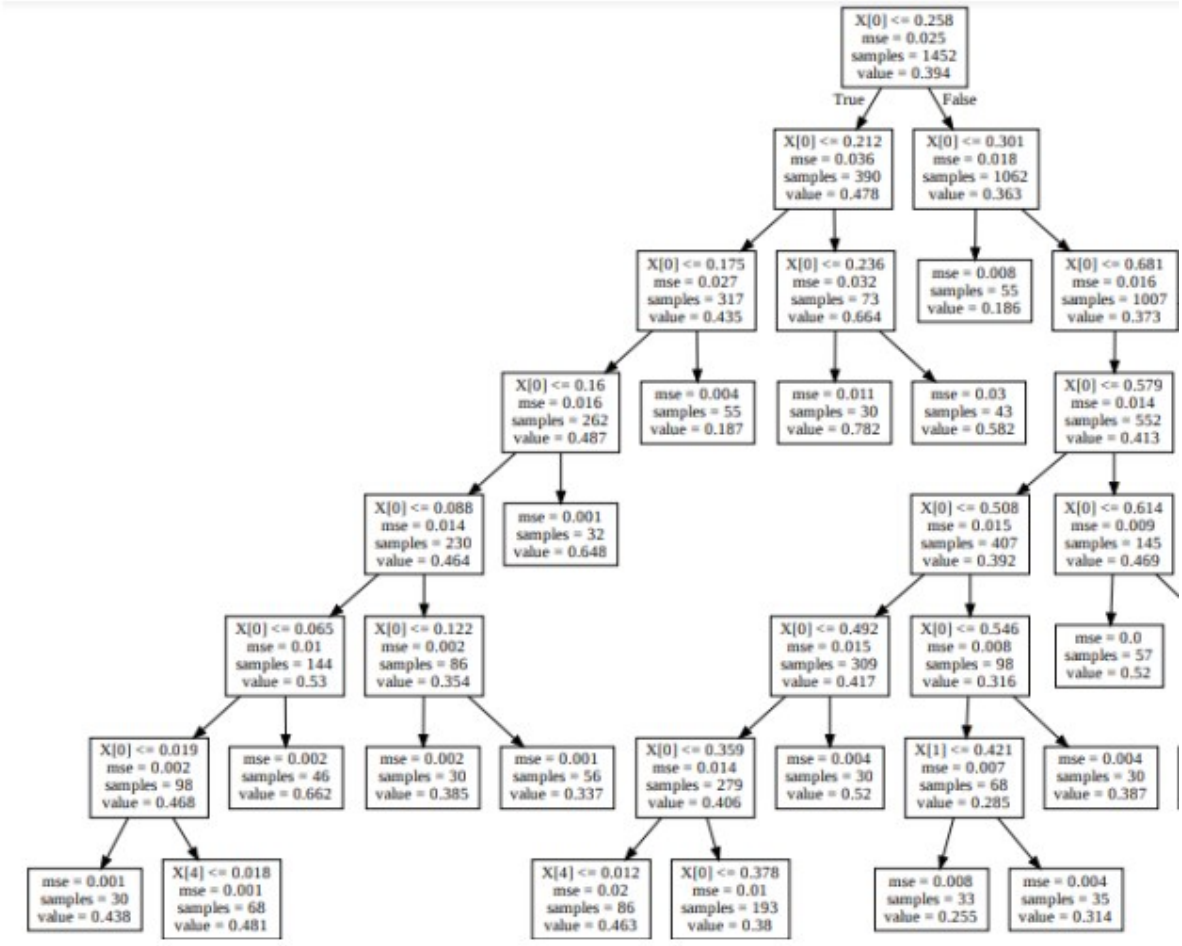

Figure 3.15 – Tree for target variable (target2)

Fig. 3.16 visualizes the results of the methods. The graphs show the actual and forecast data. It can be seen that the random forest model predicts the target variables much better, since the forecast values are close to the actual ones along the bisector, and this is also visible in the forecast quality estimates that were shown in Table 3.1. These results meet the needs of the customer and will be transferred to him.

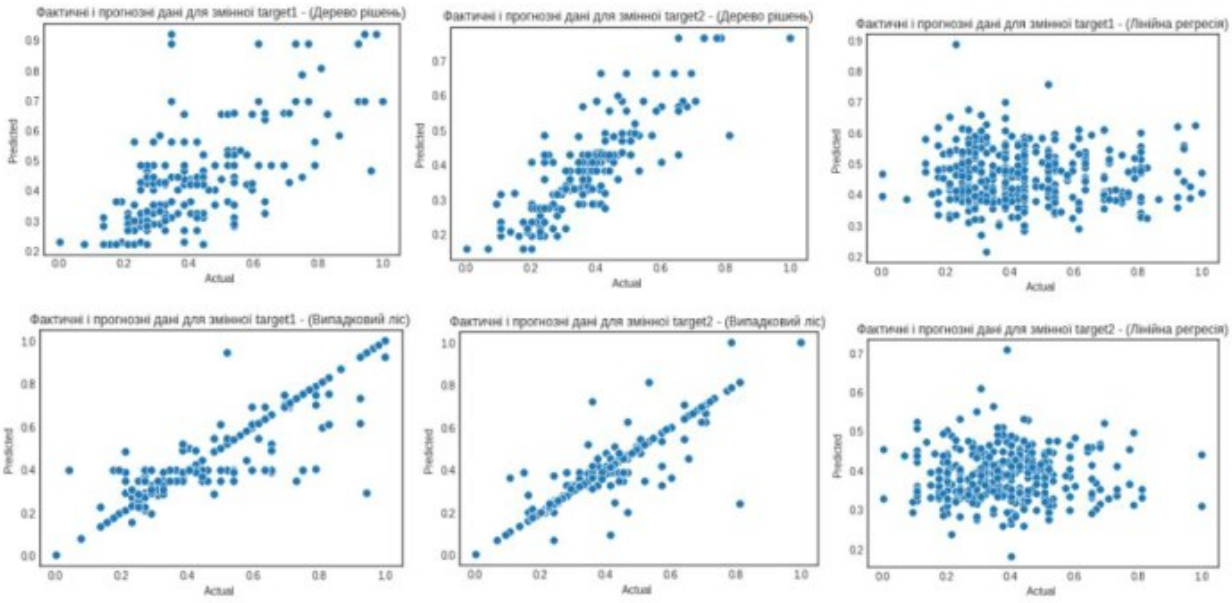

Figure 3.16 – Results of forecast and actual data

All the results that were demonstrated, they say that the most effective was the random forest machine learning model. After that, a convenient program code was developed so that the customer could easily operate this system. In order to predict other data, it is necessary to download the data to the root directory, and then run the program code through the command line of the operating system. All results will be received and stored in the excel file in the form shown in Fig. 3.18.

A	B	C	D
target1	target2	pred_tar_1	pred_tar_2
106	105	0,2404	0,3753
102	102	0,6273	0,3805
116	111	0,3092	0,3641
106	108	0,3114	0,3801
108	109	0,4808	0,3727
111	122	0,486	0,3853
119	107	0,4448	0,4105
106	105	0,4846	0,4313
107	106	0,4308	0,3845
103	103	0,4229	0,4903
140	125	0,2837	0,3288
106	105	0,4297	0,3395

Figure 3.18–Output result

3.5. Discussion and future trends

Discussion of Parallel Stream Analytics

The rapid evolution of data-generating technologies has made real-time stream analytics a cornerstone of modern information systems. The analysis presented in the preceding sections highlights that parallel processing is essential for enabling real-time stream analytics at scale. By distributing computations across multiple processing units, these systems can handle high-volume, high-velocity data streams while maintaining stringent latency requirements.

Parallel stream processing allows multiple events to be processed concurrently, improving throughput and ensuring that results are delivered in near real-time. For example, consider an intelligent traffic management system in a metropolitan area. Sensors embedded in roads and traffic signals generate continuous streams of data, including vehicle counts, speeds, and congestion levels. Without parallel processing, the sequential handling of these streams would delay insights, rendering the system ineffective at managing live traffic flows. By leveraging data-level and task-level parallelism, the system can process multiple streams simultaneously, detect congestion patterns in real time, and issue adaptive traffic signals.

However, the effectiveness of parallel processing depends heavily on system design choices. Several factors influence performance, scalability, and resource efficiency:

1. Data Partitioning Strategies: How incoming streams are divided among processing nodes significantly affects workload distribution. Poor partitioning can lead to data skew, where some nodes are overloaded while others remain underutilized, resulting in bottlenecks. Effective partitioning strategies must account for both event distribution and the nature of computations being performed.

2. Parallel Execution Models: Whether a system relies on data parallelism, task parallelism, pipeline parallelism, or a hybrid approach affects throughput, latency, and system flexibility. Systems optimized for stateless operations may underperform when handling stateful analytics unless their execution model efficiently handles shared state.

3. State Management Techniques: Stateful stream processing, such as computing moving averages or detecting patterns across multiple events, introduces complexity. Systems must manage state consistently across parallel tasks while supporting fault tolerance and recovery. Inefficient state management can degrade performance and increase processing latency.

4. Resource Allocation and Scheduling: How computational resources are allocated and tasks are scheduled across nodes directly impacts system efficiency. Over- or under-provisioning resources can either waste infrastructure or compromise real-time performance.

An important point is that no single method of parallel processing works best for everything. Different jobs have their own needs for speed, capacity, accuracy, and reliability. For example, a system that detects fraud in stock trading needs to respond quickly and give accurate results, since delays or mistakes could cost money. On the other hand, a product recommendation system for an online shopping site might focus on handling many requests at once, even if it causes small delays, as this won't affect the customer's experience much because of this, systems that analyze data in real-time need to be built to be adaptable and flexible so they can meet the specific needs of each task.

Trade-offs in System Design

Designing effective parallel real-time stream analytics systems involves navigating a complex landscape of trade-offs between competing objectives. These trade-offs manifest across several dimensions:

1. Throughput vs. Latency: Increasing parallelism generally improves throughput by enabling more events to be processed simultaneously. However,

additional parallel tasks introduce communication overhead, synchronization delays, and coordination complexity, which may increase overall latency. For time-sensitive applications, balancing throughput with minimal latency is critical.

2. Consistency vs. Performance: Strong consistency guarantees correctness across all parallel tasks but can require extensive coordination and synchronization, which may slow down processing. Eventual consistency models improve performance by reducing synchronization but may temporarily produce inconsistent or approximate results. Designers must assess the acceptable level of consistency for the application domain. For example, in healthcare monitoring systems, strong consistency is critical for patient safety, whereas in social media trend analysis, temporary inconsistencies may be tolerable.

3. Fault Tolerance vs. Resource Utilization: Replication, checkpointing, and other fault-tolerance mechanisms consume computational resources and network bandwidth. While these mechanisms improve system reliability, they increase operational costs and may reduce effective throughput. Optimizing this trade-off involves carefully selecting checkpoint intervals, replication strategies, and recovery protocols to balance reliability with performance.

4. Complexity vs. Maintainability: Hybrid parallel models and adaptive algorithms enhance performance but increase system complexity. Highly complex systems may be more challenging to maintain, debug, and scale, especially in distributed environments. System designers must consider long-term maintainability alongside immediate performance improvements.

5. Centralized vs. Edge Processing: Deploying processing closer to data sources (edge computing) reduces network latency but may limit the available computational resources compared to cloud-based centralized systems. Conversely, centralized systems provide virtually unlimited resources but can introduce higher latency due to data transmission delays.

Effectively navigating these trade-offs requires profiling workloads, understanding application requirements, and dynamically adapting system

configurations. Adaptive scheduling, dynamic repartitioning, and elastic resource management are key strategies for optimizing these trade-offs in real-time deployments.

Emerging Trends in Parallel Stream Processing

The field of parallel real-time stream analytics is rapidly evolving, driven by advances in hardware, software, and computing paradigms. Several key trends are shaping the future of this domain:

1. Edge and Fog Computing Integration

Edge computing refers to processing data closer to the source rather than sending it to a centralized cloud or data center. In stream analytics, this reduces network overhead, minimizes latency, and enables faster decision-making. Edge-based parallel processing partitions computation between edge devices and central servers, allowing local event filtering, aggregation, or pattern detection before transmitting results upstream.

For example, in autonomous vehicles, onboard edge devices process sensor data in real time to make immediate driving decisions, while aggregated information is sent to central cloud servers for fleet-wide analytics. Fog computing, an intermediate layer between edge and cloud, further enhances this architecture by distributing computation hierarchically.

2. Adaptive and Elastic Parallelism

Modern stream analytics systems increasingly adopt adaptive parallelism, dynamically adjusting the degree of parallelism in response to workload fluctuations. This elasticity ensures that resources are used efficiently while maintaining performance across variable input rates. Techniques such as dynamic operator scaling, load-aware task scheduling, and stream auto-repartitioning enable systems to respond to sudden spikes or drops in data volume.

Adaptive parallelism is particularly beneficial in applications such as social media monitoring or e-commerce analytics, where traffic patterns can be highly

unpredictable. By automatically reallocating resources and scaling processing tasks, systems can maintain low latency without over-provisioning infrastructure.

3. Hardware Acceleration

Advances in hardware are influencing the design and optimization of parallel stream analytics systems. Multi-core CPUs, GPUs, and specialized accelerators such as Field Programmable Gate Arrays (FPGAs) and Tensor Processing Units (TPUs) provide high-throughput parallel computation capabilities.

Parallel stream analytics frameworks increasingly leverage these accelerators for tasks such as pattern matching, machine learning inference, and large-scale aggregations. GPUs, for example, excel at executing the same operation across thousands of data items simultaneously, making them ideal for data-parallel stream processing. FPGAs provide low-latency, customizable hardware pipelines for specialized tasks, such as real-time signal processing in industrial or healthcare applications.

4. Integration of Machine Learning and AI

The integration of streaming machine learning and AI is transforming real-time analytics. Techniques such as incremental learning, online clustering, anomaly detection, and reinforcement learning allow systems to adapt to changing data patterns and evolving environments.

Parallel execution is critical for streaming AI workloads. Model partitioning, distributed feature extraction, and parallel inference pipelines enable machine learning algorithms to process high-velocity streams in real time. Research is ongoing to optimize consistency, minimize communication overhead, and improve convergence speed for streaming machine learning models.

5. Unified Batch-Stream Processing Frameworks

There is a growing trend toward unified frameworks that combine batch and stream processing. These frameworks provide consistent abstractions for both

historical and real-time data, enabling organizations to leverage the same pipelines, operators, and parallelism strategies across different workloads.

For instance, systems such as Apache Flink and Apache Spark Structured Streaming allow incremental processing of streaming data while also supporting batch-style computation on historical datasets. Unified frameworks simplify system design, reduce maintenance overhead, and enable optimization of parallel execution across both batch and streaming workloads.

4 SAFETY OF LIFE, BASIC LABOR PROTECTION

4.1. Labor protection requirements when working with electrical equipment

General provisions

The labor protection instructions for an electrician when performing repair and maintenance work on electrical equipment were developed in accordance with the Law of Ukraine “On Labor Protection” (Resolution of the Verkhovna Rada of Ukraine dated 10/14/1992 No. 2694-XII) as amended on 01/20/2018, based on the “Regulations on the Development of Labor Protection Instructions”, approved by the Order of the Labor Protection Supervision Committee of the Ministry of Labor and Social Policy of Ukraine dated January 29, 1998 No. 9 as amended on September 1, 2017, taking into account the “Rules for the Technical Operation of Consumer Electrical Installations”, approved by the Order of the Ministry of Fuel and Energy dated July 25, 2006. No. 258 (as amended by the order of the Ministry of Energy and Coal Industry of Ukraine dated 13.02.2012 No. 91, “Rules for the safe operation of electrical installations of consumers”, approved by the order of the State Supervision Service of Ukraine dated 09.01.1998 No. 4.

All provisions of this labor protection instruction apply to electricians of an educational institution who perform repair and maintenance work on electrical equipment.

Persons not younger than 18 years old who have undergone training in the specialty and who are also allowed to perform repair and maintenance work on electrical equipment independently are:

a medical examination and do not have contraindications due to health to perform this work;

introductory and primary workplace briefings on labor protection;

training in safe methods and techniques of work;

testing of knowledge of the rules for installing electrical installations, safety rules for operating electrical installations, labor protection requirements;

when repairing and maintaining electrical equipment voltage up to 1000V have an electrical safety group not lower than III, and over 1000V - not lower than IV.

Electricians must know and comply with the requirements of the labor protection instructions when performing work on the repair and maintenance of electrical equipment, instructions for working with hand tools, power tools and ladders.

Electricians when performing work on the repair and maintenance of electrical equipment must comply with the requirements of the Rules for the safe operation of electrical installations of consumers and the Rules for the technical operation of electrical installations of consumers, and have an appropriate electrical safety group in accordance with the requirements of these Rules.

When performing work on the repair and maintenance of electrical equipment, the impact of the following harmful and dangerous production factors may be observed:

fall from a height;

electric shock;

increased electric field strength;

increased dustiness of the air in the work area;

increased vibration level;

insufficient illumination of the work area;

physical overload;

neuropsychic overload.

Electricians when performing repairs and maintenance of electrical equipment must use the following PPE:

cotton overalls - for 12 months;

gloves for - 3 months;

leather boots for - 24 months;

dielectric galoshes - on duty;

dielectric gloves - on duty;

dielectric mats - on duty.

An electrician when repairing and maintaining electrical equipment is obliged to:

keep his workplace clean and tidy;

comply with the Rules of Internal Labor Regulations;

be able to use personal and collective protective equipment, fire extinguishing equipment;

be able to provide first aid to accident victims;

know and comply with all requirements of regulatory acts on labor protection, fire protection rules and industrial sanitation.

immediately inform your immediate supervisor about any accident that occurred at work, about signs of an occupational disease, as well as about a situation that poses a threat to the life and health of people;

know the testing dates of protective equipment and devices, the rules for their operation, care and use. It is not allowed to use protective equipment and devices with an expired inspection period;

perform only the assigned work;

comply with the requirements of the equipment operating instructions;

know where the first aid facilities, primary fire extinguishing equipment, main and emergency exits, evacuation routes in the event of an accident or fire are located;

know the telephone numbers of a medical institution (103) and fire department (101).

An electrician may refuse to perform the work assigned to him if a production situation arises that poses a threat to his life and health of others, or to the environment, and report this to his immediate supervisor.

Smoking, drinking alcoholic beverages and other substances that have a narcotic effect on the human body are prohibited in the workplace.

In order to prevent injuries and the occurrence of dangerous situations, the following requirements must be observed: it is impossible to involve third parties in the work;

do not start work if there are no conditions for its safe performance;

perform work only on serviceable equipment, with serviceable devices and tools;

if a malfunction is detected, immediately report it directly to

to the manager or eliminate them on their own, if this applies to their job duties;

not to touch uninsulated or damaged wires;

not to perform work that is not part of their professional duties.

Be able to provide first aid for bleeding, fractures, burns, electric shock, sudden illness or poisoning.

Follow the rules of personal hygiene:

outerwear, hats and other personal belongings should be left in the wardrobe;

work in clean overalls;

eat in the designated place.

Be able to correctly use PPE and collective protection equipment, primary fire extinguishing equipment, fire-fighting equipment, know where they are.

Persons who violate this labor protection instruction for an electrician when performing repair and maintenance work on electrical equipment shall bear disciplinary, administrative, material and criminal liability in accordance with the current legislation of Ukraine.

Safety requirements before starting work

Wear overalls, inspect and prepare the workplace, remove unnecessary objects.

Remove unauthorized persons from the work area and clear the workplace of foreign materials and other objects, fence off the work area and install safety signs.

Make sure that the workplace is sufficiently illuminated, that there is no electrical voltage on the repaired equipment.

Inspect the serviceability of switches, electrical outlets, power cords, electrical wires, connecting cables, make sure that PPE (personal protective equipment) and warning devices (dielectric gloves, safety glasses, galoshes, mats, etc.) are available and in good condition.

When working with a tool, it is necessary to make sure that it is in good condition, that there is no mechanical damage to the insulating coating and that the tool has been tested in a timely manner.

Inspect the workplace for compliance with fire safety requirements and for adequate workplace lighting.

If you find any deficiencies or violations in electrical and fire safety, immediately report them to your immediate supervisor.

4.2. Safety requirements during work

When performing your duties, an electrician must have a certificate of knowledge testing on labor protection. In the absence of a certificate or a certificate with an expiration date, the employee is not allowed to work.

Work in electrical installations is divided into 3 categories in terms of safety measures:

with voltage relief;

without voltage relief on or near live parts;

without voltage relief away from live parts.

Employees performing special types of work that require additional safety requirements must be trained in the safe conduct of such work and have a corresponding entry in the knowledge test certificate.

An employee who serves electrical installations assigned to him with a voltage of up to 1000 V alone must have a III group on electrical safety.

When performing work in electrical installations, it is necessary to carry out organizational measures that ensure the safety of work:

draw up work orders-permits, orders in accordance with the list of works performed in the order of current operation;

prepare workplaces;

admittance to work;

exercise control over the performance of work;

transfer to another workplace;

establish breaks in work and its completion.

To prepare the workplace for work that requires voltage relief, it is necessary to apply, in a certain order, the following technical measures:

perform the necessary shutdowns and take all measures that exclude erroneous or unauthorized switching on of switching equipment;

hang prohibition posters on the drives of manual and remote control keys of switching equipment;

check for the absence of voltage on conductive parts that must be grounded to protect people from electric shock;

install grounding (turn on grounding knives, use portable grounding);

install fences, if necessary, near workplaces or live parts that remain under voltage, and also hang safety posters on these fences.

depending on local conditions, fence live parts before or after their grounding.

At least two workers should work without removing voltage on or near live parts, one of whom, the work supervisor, must have group IV; the others must have group III with mandatory registration of the work with a work permit or order.

When removing and installing fuses under voltage in electrical installations with voltage up to 1000 V, all loads connected to the specified fuses should be disconnected in advance; use insulating pliers or dielectric gloves, and if there are open fuse inserts, then safety glasses.

Work using ladders must be carried out by two people, one of the workers must be at the bottom. Standing on boxes or other objects is prohibited. P

When installing extension ladders on beams, elements of metal structures, etc., the upper and lower parts of the ladder should be securely fixed to the structures.

During maintenance and repair of electrical installations, it is prohibited to use metal ladders.

4.3. Safety requirements after completion of repair and maintenance of electrical equipment

Disconnect (disconnect) the necessary electrical equipment, power tools from the network.

Clean up the workplace, remove parts, material, garbage and waste to special places.

Remove all tools and devices to the designated place.

Remove and remove overalls, PPE, wash hands thoroughly.

Inspect the workplace for compliance with all fire protection requirements.

Notify your immediate supervisor of any deficiencies and malfunctions that occurred during the work. Record this in the operational log.

Safety requirements in emergency situations

In case of fire:

turn off electrical equipment, supply and exhaust ventilation, if any;

notify the fire department by calling 101 and report this to your supervisor, and in his absence, to another official;

proceed to eliminate the source of the fire, using the fire extinguishing agents provided for this purpose. Extinguish electrical equipment that is under voltage can only be extinguished with carbon dioxide fire extinguishers of the OU type or sand. It is prohibited to extinguish them with water or foam fire extinguishers.

The electrician must remember that in the event of a sudden power outage, it can be supplied again without warning.

Mechanisms and devices should be quickly turned off:

in the event of a sudden power outage;

if their further operation threatens the safety of employees;

in the event of a feeling of electric current when touching metal parts of the starting equipment;

in case of sparking;

at the slightest sign of ignition, smoke, or a burning smell;

if an unfamiliar noise appears.

In the event of a short circuit in the power supply network, it is necessary to de-energize the equipment and notify your immediate supervisor.

If an electric shock occurs, the victim should be released from the action of the electric current, for which purpose the electrical network should be turned off or the victim should be disconnected from the conductive parts using dielectric protective equipment and other insulating items and objects (dry clothing, dry pole, rubberized material, etc.), or the wire should be cut (chopped) with any tool with an insulating handle, carefully, without causing additional injuries to the victim. Before the arrival of a medical worker, it is necessary to provide the victim with first aid.

In the event of accidents (injury to a person), immediately notify the immediate supervisor.

CONCLUSIONS

In today’s highly connected, data-filled world, real-time stream analysis has become a key part of modern computer systems. The incredible rise in data creation, fueled by the Internet of Things (IoT), banking systems, social media, sensor networks, and factory automation, has changed how organizations gather, process, and use information. Unlike old-fashioned static data sets, today’s data is continuous, fast, and diverse, coming in streams that need quick analysis. Being able to get useful insights from these streams instantly has become vital in many areas, including finance, healthcare, transportation, smart cities, and industrial oversight.

Traditional methods of handling data, which involve gathering, storing, and analyzing information in batches over time, no longer meet the needs of today's rapidly flowing data streams. Although these batch systems still work well for reviewing past data and managing large volumes of information, they can't deliver the quick responses required for real-time situations. For example, in stock trading, a delay in detecting fraudulent activity can cause significant financial loss; in autonomous vehicles, slow analysis of sensor data can compromise safety; and in health monitoring, delays in identifying issues with a patient’s vital signs can be dangerous. These examples demonstrate that delays in processing constant streams of data are more than just operational problems—they can lead to serious financial, safety, and personal risks.

To tackle these problems, parallel processing has become a key technology for real-time stream analysis, enabling systems to handle many events at once. This greatly increases speed and reduces delays, which is crucial for managing the growing volume and complexity of data in today’s applications.

This paper has provided a thorough look at how computers process data in real-time, stressing its important role in today's technology. It started by explaining the basic ideas behind analyzing data streams and using multiple processors at

once, along with reasons why these methods are popular. Data stream analysis involves processing ongoing, unpredictable data quickly to get immediate results. Multiple processing is a way of doing many tasks at the same time to increase speed and reduce delays. These ideas form the core of today's data handling, where managing continuous, fast-moving information is essential for staying competitive and efficient.

Building on this basic idea, the paper looked into different system setups created to support real-time stream analysis. Usually, these systems have several main parts: data sources, ways to collect the data, processing engines that work in parallel, tools to keep track of the system’s state, and layers to share or connect results. Each of these parts is essential to ensure that data flows smoothly, processing is quick, the system’s memory stays accurate, and results get to users without much delay.

Data sources in these setups are becoming more diverse, including everything from smart gadgets and sensors to websites and social media platforms. Since the data can arrive at different speeds, in various formats, and with varying levels of reliability, we need robust systems to gather it—especially during sudden increases in activity—while keeping everything organized and dependable. This is when multiple collection points start working simultaneously, ensuring all incoming data is captured quickly and efficiently.

The system's main component is the processing engine, which handles complex analysis tasks on incoming data streams. By using different forms of parallelism, such as splitting data, dividing tasks, lining up processes, and working on windows, these engines can process millions of events every second while keeping delays manageable. For instance, data parallelism means processing different parts of a stream in parallel, and pipeline parallelism allows data to flow smoothly through each step. Often, these methods are mixed to create flexible and scalable systems that reduce the chance of slowdowns.

A crucial part of these systems is handling their state, which helps with ongoing tasks like merging data, linking different groups, and spotting complex events. Managing the state well requires careful organization, saving progress, and fixing issues to keep everything accurate and reliable across all parts. It gets harder in systems that run tasks at the same time, where keeping everyone's state in sync takes more effort. Still, good state management is vital for providing accurate, real-time insights in applications that depend on cumulative or time-based calculations.

The output and integration layers complete the architecture by delivering processed results to downstream systems, dashboards, or automated decision-making components. Parallelism is also applied here to allow multiple results to be transmitted or stored concurrently, ensuring that the output does not become a bottleneck in the overall system. In addition, fault tolerance, load balancing, and scalability mechanisms are integrated throughout the architecture to maintain continuous operation even under failure conditions or fluctuating workloads.

The paper also provided a detailed analysis of parallel processing techniques for real-time stream analytics. These techniques include:

1. Data parallelism, where streams are partitioned and processed independently, providing fine-grained scalability.
2. Task parallelism, allowing multiple stages or operations to execute concurrently to reduce overall pipeline latency.
3. Pipeline parallelism, which maintains continuous data flow through sequential processing stages executed in parallel.
4. Window-level parallelism, enabling simultaneous computation over multiple temporal or event-based windows.
5. Operator parallelism, which replicates operators across data partitions for efficient scaling.

The combination of these techniques forms the backbone of high-performance stream analytics systems, allowing them to meet the dual requirements of low latency and high throughput.

Furthermore, the study explored key algorithms and frameworks for parallel stream analytics, including filtering, aggregation, pattern detection, and streaming machine learning. Filtering and transformation algorithms benefit from stateless parallelism, while aggregation and pattern detection algorithms leverage stateful parallel execution. Machine learning algorithms for streaming data, including classification, anomaly detection, and predictive analytics, rely on distributed model execution and parallel feature extraction to scale effectively. Frameworks such as Apache Flink, Apache Spark Structured Streaming, and Apache Kafka Streams provide pre-built abstractions and runtime support, facilitating the implementation of these algorithms while transparently managing parallelism, state, and fault tolerance.

Despite this progress, systems that analyze data in real time still face several problems and limits. Growing the system's capacity isn't always straightforward; uneven data distribution and workload imbalances can cause delays. Waiting times are still affected by the need for coordination, communication, and network traffic. Managing the state of data across multiple locations adds difficulty, especially when trying to find the right balance between saving checkpoints and maintaining system speed. While necessary, safety nets that prevent system failures can add extra work and storage needs. Additionally, managing resources and keeping operational costs down are key to ensuring the system works well and remains affordable. Recognizing and solving these issues is essential for creating systems that are dependable, efficient, and easy to maintain.

The discussion of future trends underscores the field's dynamic, evolving nature. Integration with edge and fog computing is enabling real-time processing closer to the data source, reducing latency and bandwidth usage. Adaptive and elastic parallelism allows systems to dynamically adjust to fluctuating workloads, improving resource utilization and responsiveness. Hardware advancements, including multi-core processors, GPUs, FPGAs, and TPUs, provide new avenues for accelerating parallel stream processing. Data-driven analysis and learning

systems enhance predictive capabilities and decision-making, while unified batch-stream processing frameworks simplify system design and maintenance.

Looking ahead, there are many opportunities for research in this area. Future work could focus on simple, efficient ways to manage system state; better methods to evenly distribute workload and prevent data imbalance; quick fault recovery; running machine learning tasks in parallel on streaming data; energy-saving computing for devices at the edge; and secure, privacy-protected ways to analyze data together. Solving these problems will need teamwork across different fields, including computer systems, processing techniques, data analysis, and artificial intelligence.

In conclusion, using multiple tasks at once is essential for analyzing data as it happens. It turns continuous, fast-moving data flows from a problem into an opportunity, allowing quick and useful insights that support making decisions in many areas. Companies and researchers need to keep exploring, improving, and creating new ways to design systems that handle data streams in parallel, so they can keep up with the growing needs of data-based applications. By making good use of handling many tasks at once, today's analysis systems can provide scalable, quick, and dependable real-time information, making sure the full value of streaming data is recognized.

As digital systems keep advancing, analyzing data as it happens, driven by multiple computers working together, will continue to be a key foundation for responsive, intelligent, and adaptable systems. Its significance will keep growing, especially as data becomes more central to making operational decisions, forecasting outcomes, and automating responses across various fields such as finance, healthcare, industrial automation, smart cities, and others. The way advanced processing techniques, cutting-edge algorithms, and solid system structures work together will ultimately influence the efficiency, dependability, and future success of these real-time data analysis platforms.

REFERNCES

1. The Internet of Things: A survey / M. G. J. van den Brand et al. IEEE Communications Surveys & Tutorials. 2013. Vol. 15, no. 1. P. 164–181. URL: https://www.sciencedirect.com/science/article/pii/S1389128610001568 (дата звернення: 25.01.2026).

2. Turkington B. Real-time Stream Analytics. 1st ed. Birmingham, UK : Packt Publishing, 2016. 320 p. URL: https://www.packtpub.com/product/real-time-stream-analytics/9781785282643.

3. Sakr S., Gaber A. Large Scale and Big Data: Processing and Management. CRC Press, 2014. 614 p. URL: https://www.routledge.com/Large-Scale-and-Big-Data-Processing-and-Management/Sakr-Gaber/p/book/9781466581096.

4. StreamCloud: An Elastic and Scalable Data Streaming System / V. Gulisano et al. IEEE Transactions on Parallel and Distributed Systems. 2012. Vol. 23, no. 12. P. 2351–2365. URL: https://oa.upm.es/16848/1/INVE_MEM_2012_137816.pdf.

5. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing / T. Akidau et al. Proceedings of the VLDB Endowment. 2015. Vol. 8, no. 12. P. 1792–1803. URL: https://www.vldb.org/pvldb/vol8/p1792-akidau.pdf.

6. Hirzel M. et al. A Catalog of Stream Processing Patterns. ACM Computing Surveys. 2014. Vol. 46, no. 4. P. 1–45. URL: https://dl.acm.org/doi/10.1145/2543581.

7. Chen C. L. P., Zhang C. Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences. 2014. Vol. 275. P. 314–347. URL: https://www.sciencedirect.com/science/article/pii/S002002551400374X.

8. Apache Flink: Stream and Batch Processing in a Single Engine / P. Carbone et al. IEEE Data Engineering Bulletin. 2015. Vol. 38, no. 4. P. 28–38. URL: https://ieeexplore.ieee.org/document/7343867.

9. Zaharia M. et al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. Proc. ACM SOSP. 2013. P. 423–438. URL: https://dl.acm.org/doi/10.1145/2517349.2522737.

10. The Design of the Borealis Stream Processing Engine / D. J. Abadi et al. Proc. CIDR. 2005. URL: http://cidrdb.org/cidr2005/papers/3_Abadi.pdf.

11. Kreps J., Narkhede N., Rao J. Kafka: A Distributed Messaging System for Log Processing. Proc. NetDB. 2011. URL: https://www.usenix.org/system/files/conference/netdb11/netdb11-final8.pdf.

12. Trill: A High-Throughput Incremental Query Engine for Diverse Analytics / S. Chandramouli et al. Proceedings of the VLDB Endowment. 2014. Vol. 8, no. 4. P. 401–412. URL: https://www.vldb.org/pvldb/vol8/p401-chandramouli.pdf.

13. The Power of Both Worlds: A Hybrid Approach to Scalable Real-Time Stream Processing / M. A. U. Nasir et al. Proc. IEEE ICDE. 2015. URL: https://ieeexplore.ieee.org/document/7113126.

14. Lohachab K. S., Karambir B. A Review of Real-Time Stream Analytics Frameworks. Journal of Big Data. 2019. Vol. 6, no. 1. URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0216-3.

15. State Management in Apache Flink / P. Carbone et al. Proc. ACM SIGMOD. 2017. URL: https://dl.acm.org/doi/10.1145/3035918.3064035.

16. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark / M. Armbrust et al. Proc. ACM SIGMOD. 2018. URL: https://dl.acm.org/doi/10.1145/3183713.3190664.

17. MillWheel: Fault-Tolerant Stream Processing at Scale / T. Akidau et al. Proceedings of the VLDB Endowment. 2013. Vol. 6, no. 11. URL: https://www.vldb.org/pvldb/vol6/p1128-akidau.pdf.

18. S-Store: Streaming Meets Transaction Processing / J. Meehan et al. Proceedings of the VLDB Endowment. 2015. Vol. 8, no. 13. P. 2134–2145. URL: https://www.vldb.org/pvldb/vol8/p2134-meehan.pdf.

19. Gedik B. et al. SPADE: The System S Declarative Stream Processing Engine. Proc. ACM SIGMOD. 2008. URL: https://dl.acm.org/doi/10.1145/1376616.1376671.

20. Edge Computing: Vision and Challenges / W. Shi et al. IEEE Internet of Things Journal. 2016. Vol. 3, no. 5. P. 637–646. URL: https://ieeexplore.ieee.org/document/7462615.

21. George G. et al. Parallel processing using GPU for real-time data streaming. Proc. IEEE ICSPC. 2017. URL: https://ieeexplore.ieee.org/document/8327318.

22. Y. Leshchyshyn, L. Scherbak, O. Nazarevych, V. Gotovych, P. Tymkiv and G. Shymchuk, «Multicomponent Model of the Heart Rate Variability Change-point,» 2019 IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), Polyana, Ukraine, 2019, pp. 110-113, doi: 10.1109/MEMSTECH.2019.8817379

23. Lytvynenko, S. Lupenko, O. Nazarevych, G. Shymchuk and V. Hotovych, «Mathematical model of gas consumption process in the form of cyclic random process,» 2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT), LVIV, Ukraine, 2021, pp. 232-235, doi: 10.1109/CSIT52700.2021.9648621

24. Bodnarchuk, I., Kunanets, N., Martsenko, S., Matsiuk, O., Matsiuk, A., Tkachuk, R., Shymchuk, H.: Information system for visual analyzer disease diagnostics. CEUR Workshop Proceedings 2488, pp. 43-56 (2019).

25. Шимчук Г. В. Дослідження методів захисту відомих хмарних платформ : кваліфікаційна робота освітнього рівня „Магістр“ „125 – Кібербезпека“ / Г. В. Шимчук. – Тернопіль : ТНТУ, 2022. – 74 с.

26. Методичні вказівки розроблені у відповідності з навчальним планом для студентів освітнього рівня бакалавр спеціальності 126 «Інформаційні системи та технології» / Уклад.: О. Б. Назаревич, Г. В. Шимчук, Н. М. Шведа. – Тернопіль : ТНТУ 2020. – 22 с.

27. V. Kozlovskyi, Y. Balanyuk, H. Martyniuk, O. Nazarevych, L. Scherbak and G. Shymchuk, «Information Technology for Estimating City Gas Consumption During the Year,» 2022 International Conference on Smart Information Systems and Technologies (SIST), Nur-Sultan, Kazakhstan, 2022, pp. 1-4, doi: 10.1109/SIST54437.2022.9945786.