Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN: 2468-4376 https://www.jisem-journal.com/ **Research Article**

Big Data Engineering Patterns for Real-Time Analytics: A Comprehensive Framework

Vamsidhara Reddy Doragacharla
Independent researcher, USA

ARTICLE INFO

Received: 23 Jan 2026

Revised: 28 Jan 2026

ABSTRACT

The modern information environment has radically changed with the accelerating expansion of high-speed data streams of Internet of Things devices, digital interactions, and financial systems. The classic batch processing processes become more and more deficient in the current enterprise demands that require the ability to make instant decisions in milliseconds after the data is generated. Although Lambda architecture is meant to provide a tradeoff between fault tolerance and low-latency processing by having separation of batch and speed layers, it induces significant complexity in operation by having two sets of codebase maintenance and code duplication. The Kappa architecture is a conceptual simplification that assumes all data are continuous streams and removes an architectural duplication of Lambda architectures, in addition to making substantial complexity and resource reduction infrastructural. Stateful computations with many operators can be executed on modern stream computing infrastructures that provide advanced state management schemes to sustain stateful computations such as temporal joins, pattern matching, and long-running aggregations that tolerate operator failure. Layered storage architectures make the distinction between hot, warm, and cold layers of storage data through the use of access patterns and latency requirements, and cloud services use advanced tiering strategies where data can be automatically moved across the storage classes. Column-based storage systems are more effective in analytical queries with a better compression ratio and lower input-output needs, and multi-level caching solutions greatly minimize unnecessary calculations. The event-driven architectures encourage loose coupling among the system components by using event production and consumption patterns that change the flow of information radically, in contrast to the normal request-response systems. The integration of integrated stream processing, efficient state management, and optimized storage architecture provisions allows organizations to satisfy their scalability requirements and enhance maintainability, resource utilization, and democratize real-time analytical performance across a wide range of organizational environments.

Keywords: Real-Time Stream Processing, Kappa Architecture, Event-Driven Systems, Stateful Computation, Columnar Storage

1. Introduction

A fundamental change in the present-day data landscape has been created by the explosive data streams of high velocity, which are the products of Internet of Things (IoT) devices, electronic user interactions, and financial transaction systems. The digital universe is continually experiencing amazing growth, and data creation is growing exponentially, with organizations all over the world digitalizing their activities and consumers becoming more and more connected with each other through connected devices. This expansion includes data that is produced by a wide range of sources, such as mobile devices, embedded sensors, surveillance systems, and autonomous vehicles, and fundamentally changes the way organizations have to treat data processing and analytics [1]. This change has led to a paradigm change in the old method of batch processing to an actual-time stream processing architecture, where

organizations can make actionable insights within milliseconds of data creation. The need to make instant decisions in competitive market settings has made traditional batch-based processing models, which tend to create staleness of data and reduce the actionability of data, more unsatisfactory to meet the needs of the current enterprise environment. The architectural overhead of maintaining two distinct layers of batch and speed, as is the traditional Lambda architecture implementation, has proven to have a significant operational overhead and code duplication as organizations increase their data operations. More sustainable, maintainable, and resource-efficient alternatives can be found in the modern engineering patterns, especially those patterns supporting unified stream processing techniques. This paper presents a vendor-neutral reference model that defines reusable patterns that should be used to design real-time analytics systems in a variety of industries, as well as analyzes the larger societal aspects of democratizing access to real-time data infrastructure.

Aspect	Traditional Systems	IoT-Enabled Systems
Data Sources	Centralized databases and applications	Distributed sensors, embedded devices, mobile systems
Data Velocity	Periodic batch intervals	Continuous high-velocity streams
Processing Location	Centralized data centers	Edge computing and centralized infrastructure
Network Characteristics	Stable connectivity	Variable connectivity with heterogeneous capabilities
Latency Requirements	Minutes to hours acceptable	Sub-second to seconds required
Processing Guarantees	Eventual consistency sufficient	Ordered processing with fault tolerance is essential

Table 1: Data Generation Characteristics and IoT Analytics Requirements [1][2]

2. Foundational Concepts and Event-Driven Architectures

Real-time data engineering requires a very accurate interpretation of the timeliness requirements, which differ significantly across application domains. Connected devices have created streams of sensor readings, changes of state, and telemetry data that require instant processing and analysis, and the speed and volume of such data have changed fundamentally with the development of Internet of Things ecosystems. IoT systems produce data at scales beyond traditional database-based architectures, and more specific streaming platforms are needed that can ingest millions of events per second whilst ensuring ordered processing guarantees and fault tolerance functionality. The design use of IoT analytics should be able to accommodate the heterogeneity of device capabilities, variations in network connectivity, and the distributed character of the edge computing environment, where initial data filtering and aggregation are frequently done before transmission to centralized processing infrastructure [2]. Fraud detection and system health monitoring use cases require decision latencies of several seconds or sub-seconds, but applications like near-real-time operational reporting can tolerate delays of several minutes. These time demands have a tremendous impact on the architecture of ingestion pipelines, processing logic, and storage structures. Theoretically, real-time processing lies on a continuum between true streaming systems, micro-batch processing systems, and low-latency batch analytics, with each having different trade-offs between resource exploitation patterns and computational complexity.

A second conceptual model in real-time data engineering is the event-driven model. In event-driven systems, the event-driven description of changes in environmental states is represented by structured events, which are meaningful occurrences in time. These events replace the traditional data representations based on tables as the main unit of analysis and fundamentally change the direction in which data moves through the organizational systems. Event-driven architectures fundamentally reduce the conventional request-response patterns of system integration and flow of information. Instead of components calling upon other components by using synchronous calls, event-driven systems use the generation and use of events, which are important changes of state or occurrences within the system. This type of architecture encourages lax association among the parts of the system, since event generators do not have any insight regarding which consumers will handle their events and the manner in which they will respond. The event notification pattern enables components to announce state changes without determining future actions, whereas event-carried state transfer adds enough contextual information to the event to enable consumers to update their local state without making further service calls. Event sourcing proceeds with this idea to the role of event as the system of record and records the entire history of the state changes instead of just the current state [4].

The ingestion patterns dictate the procedures by which the raw signals of applications, devices, and operational systems are converted to structured analytic events. The centralized collection model of events gathers the events produced by many producers into a common transport layer, which offers durability and the ordered delivery semantics. In other cases, domain-oriented ingestion policies use event streams of different business domains that develop with relative autonomy. The choice of these architectural patterns is determined by the organizational structure, data governance needs, and the expected frequency of schema change in event definitions. Stream modeling is not limited to the structure of particular events, but includes the organization of streams, partitioning policies, and key selection criteria.

Pattern Type	Characteristics	System Integration Approach
Request-Response	Synchronous component invocation	Direct service coupling with blocking calls
Event Notification	Signal state changes without dictating actions	Loose coupling through event publication
Event-Carried State Transfer	Embed contextual information in events	Local state updates without additional calls
Event Sourcing	Events as a system of record	Complete state change history retention
Watermark-Based Processing	Temporal completeness reasoning	Out-of-order event accommodation
Stream Partitioning	Parallel processing enablement	Localized state management per partition

Table 2: Event-Driven Architecture Patterns and Processing Models [3][4]

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN: 2468-4376 https://www.jisem-journal.com/ **Research Article**

3. Architectural Paradigms: Lambda Versus Kappa Architectures

The Lambda architecture was developed as a powerful design pattern that was aimed at aligning both fault tolerance and low-latency processing goals by implementing discrete layers of batch and speed. The batch layer is the one that processes entire datasets to achieve computational accuracy and consistency, and the speed layer is the one that creates approximate real-time views to address the immediacy needs. Nonetheless, this method of architecture adds a high level of operational complexity due to the requirement of having two sets of codebases deployed with the same business logic - a method that is fundamentally inefficient and error-prone during the lifecycle of systems and requirements. The inherent problem with Lambda architecture lies in the fact that it requires executing and having the same business logic in two different processing structures: one that is streamlined to handle batch processing of historical information and the other to handle real-time information using streams. Such redundancy results in high engineering maintenance overhead, since any change in analytical logic is required to be recreated in each of the two layers, and minor variances in the semantics of the framework may cause differences in the batch and streaming output. This issue is more than just copying of code since it also involves testing strategies, deployment pipelines, and monitoring of operations, as teams need to ensure that the two processing paths generate similar outputs even though they have different architectures [5]. These challenges are compounded by the fact that serving layers need advanced merge logic to resolve potentially conflicting output in batch and speed layers, and provide queries with latency-optimal service.

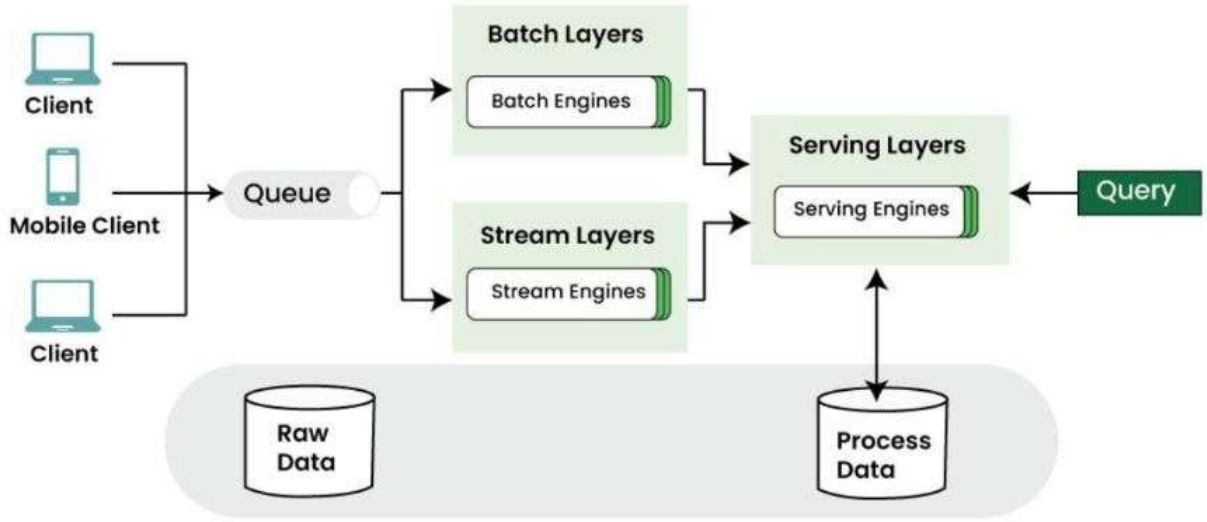

Figure 1: Lambda Architecture Diagram

The Kappa architecture is an abstract simplification where all data are continuous streams, and there is no architectural redundancy between streams as in Lambda-style designs. Through the use of a single processing engine to process historical reprocessing and real-time stream analysis, organizations have the potential to attain massive complexities in infrastructural reduction and resource utilization. Apache Kafka is an example of the underlying technologies that make Kappa-style architectures possible by providing a distributed commit log architecture that conceptualizes all data as event streams that are treated as immutable. Kafka has an architecture that isolates storage issues and processing logic; different processing applications can consume the same event streams with independent consumption

offsets. With this separation, it is possible to perform actual stream processing in which batch reprocessing is just another streaming computation over historical information that reads the entire log at the beginning and not the current tail. This durability is guaranteed by the replicated logs offered by Kafka, such that the events can be reprocessed by the age of the events, whereas the compaction strategies can ensure that the most recent state of any key is kept efficiently, as it is not needed to know the full history of the key [6]. The stream processing model removes the complexity of holding batch and speed layers apart and allows more flexible patterns of deployment where processing logic can be changed and redeployed without requiring an architectural change.

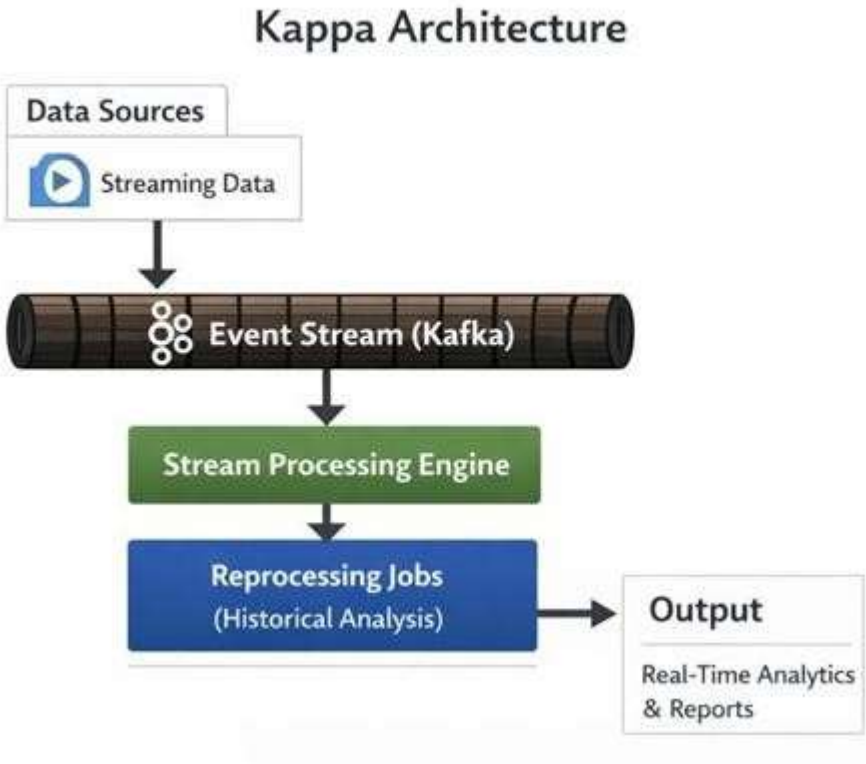

Figure 2: Kappa Architecture Diagram

The nature of the resource utilization in these two architectural paradigms varies significantly. The reason why lambda architectures are often unable to attain high resource utilization rates is that it is difficult to efficiently schedule workloads on heterogeneous batch and streaming components. Conversely, Kappa architectures show better usage, and the efficiency increases of integrated processing systems. The efficiency gains are directly translated to lower operating costs and environmental impact due to energy savings and hardware savings. The event-based processing patterns are also more sustainable due to the implementation of execute-on-demand models. Eventdriven systems do not use a continuous polling system, which keeps the CPU constantly busy whether or not data is accessible, but only when an event of incoming data occurs, event-driven systems run processing tasks. This architecture reduces the amount of idle resource usage whilst ensuring the responsiveness of the workloads needed by real-time analytics. The maintenance implications of architectural decisions are not limited to the short-term operational issues but can be applied to the long-term evolution of the system and the technical debt. Lambda architectures present a high maintenance cost in the form of the need to coordinate any change in logic between batch and streaming codebases, guarantee consistency between layers, and the complexity of serving layer combining operations. Kappa architectures simplify

maintenance by combining codebases into a single system and allowing simplified deployment pipelines, which allow organizations to use engineering resources on feature development instead of architectural coordination.

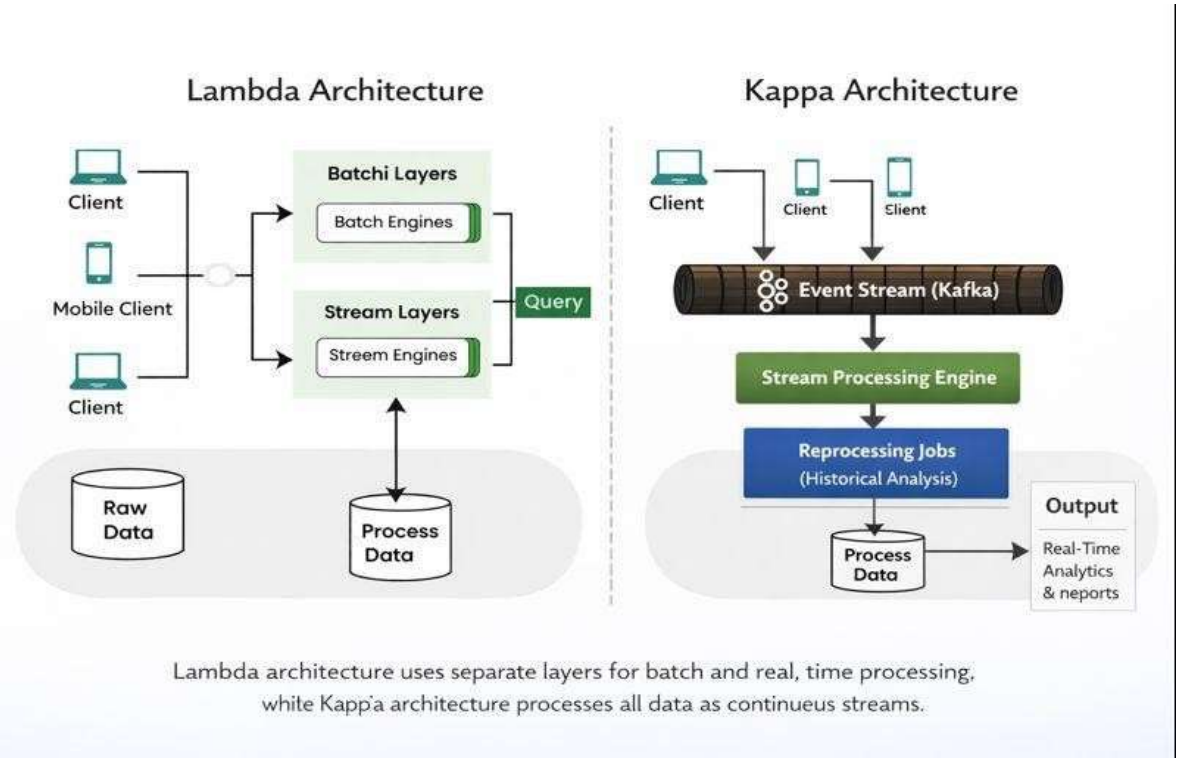

Figure 3: Side-by-Side Comparison Diagram

Architectural Aspect	Lambda Architecture	Kappa Architecture
Code Maintenance	Dual codebases for identical logic	Single unified codebase
Processing Layers	Separate batch and speed layers	Unified stream processing layer
Data Model	Batch tables and streaming views	Immutable event streams throughout
Reprocessing Approach	Batch layer recomputation	Stream replay from the log beginning
Operational Complexity	High due to layer synchronization	Reduced through architectural unification
Testing Requirements	Validation across heterogeneous frameworks	Single framework validation
Deployment Strategy	Multiple pipelines for batch and streaming	Simplified deployment with unified logic
Storage Strategy	Separate batch and real-time stores	Replicated log with compaction options

Table 3: Lambda versus Kappa Architecture Comparison [5][6]

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN: 2468-4376 https://www.jisem-journal.com/ **Research Article**

4. State Management and Processing Patterns in Stream Analytics

The computation and state management of event streams are at the core of processing patterns of real-time analytics systems. Stateless transformations such as filters, projections, and element-wise mappings pose little architectural complexity and are linearly dependent on throughput requirements. Stateful operations like aggregations, stream join, and session boundary detection, however, need to maintain changing state, which is preserved across a sequence of events and time windows. The more recent stream processing systems have developed more elaborate state management mechanisms that need to be consistent across failures and provide high-throughput stream processing. Apache Flink provides state management by means of an abstraction layer between application logic and the state storage backend, so that operators can maintain keyed state, operator state, and broadcast state with varying consistency and performance properties. The checkpoint mechanism of the framework produces snapshots of distributed operator state at fixed time intervals, and can recover to knowngood states after failures, with no loss of data. State backends may be set to store state in memory (so that they run as fast as possible), on local disk (so that they can sustain more state), or in a distributed file system (so that they can be durable), and the framework will take care of state serialization, distribution, and recovery [7]. Such capabilities support complex stateful computations such as temporal joins across multiple streams, pattern matching over a stream of events, and long-running aggregations that do not lose accuracy when operators fail or when the cluster is reconfigured.

The patterns of windowed aggregation divide event streams into temporal or count-based time ranges to calculate measures such as counts, averages, percentiles, and sums. Windows specifications can use fixed boundaries that divide the time into non-overlapping intervals, sliding windows that give continuous updates of metrics with configurable overlap, or session windows that group events based on patterns of activity separated by inactive thresholds. The choice of the windowing semantics has a direct effect on the semantic meaning of the computed metrics and the processing system resource demands. Massive stream processing systems should effectively execute windowing operations involving distributed clusters and correctness guarantees. Twitter Heron, a system to handle hundreds of billions of events per day at thousands of computing nodes, uses windowing based on a topology abstraction where processing logic described as directed acyclic graphs of operators is used. The system architecture does not couple the computation with the management of cluster resources, which means that topologies can be deployed in a wide variety of infrastructure environments, but with the same semantics.

Sessionization is an expert pattern that is quite useful when analyzing user behavior and interaction patterns. In this pattern, the events of individual users or entities are grouped according to time proximity, with a session delimiting the time period between the next set of events. To perform accurate sessionization, it is necessary to per-user state and apply logic of time out so that session integrity can be preserved even in the event of late-arriving events. The trend is critical towards applications such as user journey, behavioral anomaly, and personalization engines that adjust to usage patterns. Semantic processing ensures that exactly-once and at-least-once semantics are conceptual contracts specifying what the system is expected to do in the event of a failure or a retry. To have exactly-once processing, it is important to coordinate the checkpointing mechanisms, replay logic, and idempotent operations to generate the same results, no matter how frequently they are executed. Such assurances are very important to financial applications, inventory control systems, and other areas where duplicate processing may give erroneous or non-uniform states. Such guarantees are normally implemented by distributed coordination protocols, updates of transaction states and significant attention to interaction with other systems. Stateful stream processing combines both significant performance benefits and real-time analytics by storing computational state as a part of the processing infrastructure itself, which eliminates the need to look at any external data store regarding the historical context of each event,

granted to an incoming event, and frees up backend database capacity by a factor of many, while improving the processing latency properties.

Processing Pattern	State Characteristics	Implementation Considerations
Stateless Transformations	No persistent state required	Linear scalability with minimal complexity
Keyed State	Per-key state maintenance	Partitioned state with key-based distribution
Operator State	Shared state across keys	Broadcast or union state distribution
Tumbling Windows	Non-overlapping time segments	Discrete temporal partitioning
Sliding Windows	Overlapping time ranges	Continuous metric updates with higher overhead
Session Windows	Activity-based boundaries	Timeout-driven event grouping
Checkpoint Mechanism	Consistent distributed snapshots	Failure recovery without data loss
Backpressure Handling	Flow control during traffic spikes	Prevention of downstream overwhelm

Table 4: State Management and Windowing Patterns in Stream Processing [7][8]

5. Storage Architectures and Performance Optimization Strategies

Layered storage architectures are inherently used with real-time analytics systems, and they differentiate between hot, warm, and cold layers of data based on access frequency and latency needs. Hot storage stores recent information in memory or solid-state-based storage systems, which can support sub-second query times needed by operational dashboards and real-time application integrations. Warm storage stores data with large temporal windows of data on balanced storage technologies that deliver interactive query performance with moderate latency behavior that can be used in exploratory analysis. Cold storage stores historical information with cost-efficient data storage solutions in which the latency of access is appropriate, as long as compliance, auditing, and trend analysis are tolerated in the long run. Cloud storage systems have also met the advanced tiering mechanisms along with automatic migration of data between classes of storage as per patterns of accessibility and aging policies, to allow organizations to maintain a cost optimization procedure without affecting the availability of data. Amazon S3 has several classes of storage that impose radically different pricing models, with the most frequently accessed data being stored in regular storage that optimizes responses based on low-latency access, infrequently accessed data being moved to storage classes with lower per-gigabyte charges but higher retrieval charges, and archival data being moved to glacier storage with response times of undoubtedly hours though at very low storage costs. Organizational lifecycle policies can be established that can automatically move objects between storage classes as they age, with intelligent tiering policies that can observe access profiles and move data between the tiers automatically [9]. The data orchestration between these levels plays a crucial role in defining the quality of user experience and total cost of ownership, which should be carefully analyzed, taking into consideration the pattern of queries, data retention needs, and cost limitations.

The design of storage layers in real-time analytics involves close coordination of physical data representations and queries that are expected. Figure 1 above shows that serving layers often use denormalized data models to eliminate join operations and provide quick access paths to particular query patterns. Meanwhile, the storage beneath can be based on more normalized forms to make data

maintenance easier and schema evolution easier. Architectural designs such as write-optimized append-only logs to read-optimized columnar stores or hybrid storage systems that can support both a transaction and analytical workload are common in production systems. The basic performance attributes of any storage system vary radically depending on its internal structure, with row-oriented databases favoring transactional workloads that need to access entire records and column-oriented systems favoring analytical workloads that sum up values across a large number of rows to a given attribute. Column stores are also able to achieve high compression ratios; similar values are stored side by side, and storage requirements are reduced by an order of magnitude over row stores, using run-length encoding, dictionary encoding, and bit-packing. The lower I/O demands are directly reflected in increased query speed with analytical workloads because decompressing data in memory and reading compressed columnar data off disk when only a subset of columns is required by the query is much more efficient than simply scanning through all rows when this is not the case. In modern columnar databases, the lack of rebuilding column values into complete tuples until later in query processing, and batch-processing of values with CPU SIMD instructions are additional performance optimizations [10]. Such architectural options have significant impacts on the storage performance and query performance features of the real-time analytics systems.

Multi-leveled caching policies are critical performance optimization techniques for real-time analytical loads. Storing commonly used results of analytics in memory caches that are placed closer to consumption layers can enable systems to save many redundant computations and database queries, directly into a better user experience in terms of reduced latency and higher system capacity. The Cache hit rates, eviction policies, and time-to-live settings need to be fine-tuned depending on the query patterns and freshness data requirements. Real-time analytics value realization is at the serving layer, which is where the computational results are delivered to human analysts or other downstream systems. Operational dashboards can make use of continuous query systems, which can refresh displays as new data streams change visualizations, giving analysts real-time system state and business indicators. Embedded analytics applications have application programming interfaces that provide metrics or decision outputs to applications on request time with latency requirements of less than a second. The patterns that may be used are selective result materialization patterns, incremental view maintenance patterns, and intelligent pre-computation patterns that can be used in systems to achieve very high latency goals with control over the costs of computation. Trust and interpretability are important non-functional criteria of real-time analytical systems, as users need to have a clear definition of metrics, with consistency in processing data that arrives late, and with clarity in documentation of computational procedures so they can interpret the results properly and make effective decisions.

Conclusion

The shift to consolidated and event-driven data engineering spaces is a strategic requirement among organizations that compete in data-heavy spaces, where incorporating Kappa-style architectural designs and powerful state processing structures can realize both the scalability requirements and, at the same time, enhance the maintainability of the system and affordability of the resources. The achieved efficiency in resource use by means of cohesive streaming strategies is reflected in the physical savings in the cost of infrastructure and environmental effects, dealing with the technical optimization and sustainability issues. In addition to the immediate performance implications, optimized real-time data engineering patterns can have much more far-reaching impacts on the wider context of society by making the real-time analytical environment affordable and easy to use by resource-constrained organizations such as the government, emergency response departments, and rural health practitioners. The development of real-time analytics platforms is incremental, on a hybrid architecture that integrates both batch processing and streaming processing, shares transformation logic among different temporal processing modes, and migrates workloads to realtime processing gradually, providing practical migration strategies for organizations. With tools and structures still evolving, the theoretical

differentiation between streaming and batch processing is becoming obsolete, with combined execution engines that offer uniform programming models across temporal scales. The outlook is that future directions seem to be toward more automation of system optimization, such as adaptive resource optimization, automatic dataflow optimization, and intelligent query planning, and privacy-preserving architectures and edge computing paradigms gain more popularity as data sovereignty demands grow and latency requirements increase. The basic idea underlying architectural choices based on enduring design patterns instead of temporary technology choices makes it possible to construct real-time analytics systems that can withstand evolutionary forces but at the same time make sense conceptually, be operationally feasible, and economically viable. The intersection of technical efficiency gain with increased accessibility is both an engineering success and an opening to less biased distribution of data-driven decision-making power within organizational and societal settings, inherently altering how organizations use temporal streams of data to achieve a competitive edge and operational quality.

References

[1] David Reinsel, John Gantz, John Rydning, "The Digitization of the World: From Edge to Core," IDC White Paper, 2018. [Online]. Available: https://www.seagate.com/files/www-content/ourstory/trends/files/idc-seagate-dataage-whitepaper.pdf

[2] Tianqi Yu, Xianbin Wang, "Real-Time Data Analytics in Internet of Things Systems," ResearchGate, 2020. [Online]. Available: https://www.researchgate.net/publication/343711981_RealTime_Data_Analytics_in_Internet_of_Things_Systems

[3] Tyler Akidau et al., "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," ACM Digital Library, 2015. [Online]. Available: https://dl.acm.org/doi/10.14778/2824032.2824076

[4] Martin Fowler, "What do you mean by 'Event-Driven?" martinfowler.com, 2017. [Online]. Available: https://martinfowler.com/articles/201701-event-driven.html

[5] Jay Kreps, "Questioning the Lambda Architecture," O'Reilly Radar, 2014. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/

[6] Tom van Eijk, "Kafka: The Definitive Guide - Meet Kafka (Chapter 1)," Medium, 2024. [Online]. Available: https://medium.com/@t.m.h.v.eijk/kafka-the-definitive-guide-meet-kafka-chapter-1449ecd947e4e

[7] Paris Carbone, et al., "State Management in Apache Flink: Consistent Stateful Distributed Stream Processing," ACM Digital Library, 2017. [Online]. Available: https://dl.acm.org/doi/10.14778/3137765.3137777

[8] Sanjeev Kulkarni, et al., "Twitter Heron: Stream Processing at Scale," ACM Digital Library, 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2723372.2742788

[9] Alexander Yu, "Amazon S3 Pricing and Optimization Guide," CloudForecast.io, 2024. [Online]. Available: https://www.cloudforecast.io/blog/amazon-s3-pricing-and-optimization-guide/

[10] Daniel J. Abadi, "Column-Stores vs. Row-Stores: How Different Are They Really?" ACM Digital Library, [Online]. Available: https://dl.acm.org/doi/10.1145/1376616.1376712