Module 3
Part 1

Data Management Techniques

Contents

- Introduction
- Stream
- Stream Processing
- Data Stream Management System (DSMS)
- Complex Event Processing (CEP)
- Differences between DSMS and CEP
- The characteristics of stream data in IoT
- General architecture of a stream-processing system in IoT
- Continuous logic processing system
- Challenges in stream-processing systems

Stream

- A stream is a sequence of data elements ordered by time.
- The structure of a stream could consist of discrete signals, event logs, or any combination of time-series data
- In terms of representation, a data stream has an explicit timestamp associated with each element, → measurement of data order.
- Based on this, we formally define the denotation of stream in the context of IoT, as a Data Element–Time pair (s, Δ)

- **Denotation of stream: →**
  - **Data Element–Time pair (s, Δ)**
- **S →sequence of data elements that are made available to the processing system over time**
- **A data element may consist of several attributes, but it is normally atomic**
- **Typical element types include immutable data tuples of the same or similar category, as well as heterogeneous events that come from a variety of sources.**
- **Depending on the specific application scenario, data elements can be either regularly generated by sensor networks or randomly produced by real-world events such as updates to a particular database table, and system logs produced by Internet services.**

- **Δ → sequence of a timestamp that denotes the sequence of data elements**
- **The use of a timestamp is necessary to reconstruct the logic sequence for the following analytics**
- **In addition, timestamps can be also used to evaluate the real-time property of a stream-processing system, by checking on whether the results have been presented on time**
- **Normally, Timestamps can be implemented in two forms:**

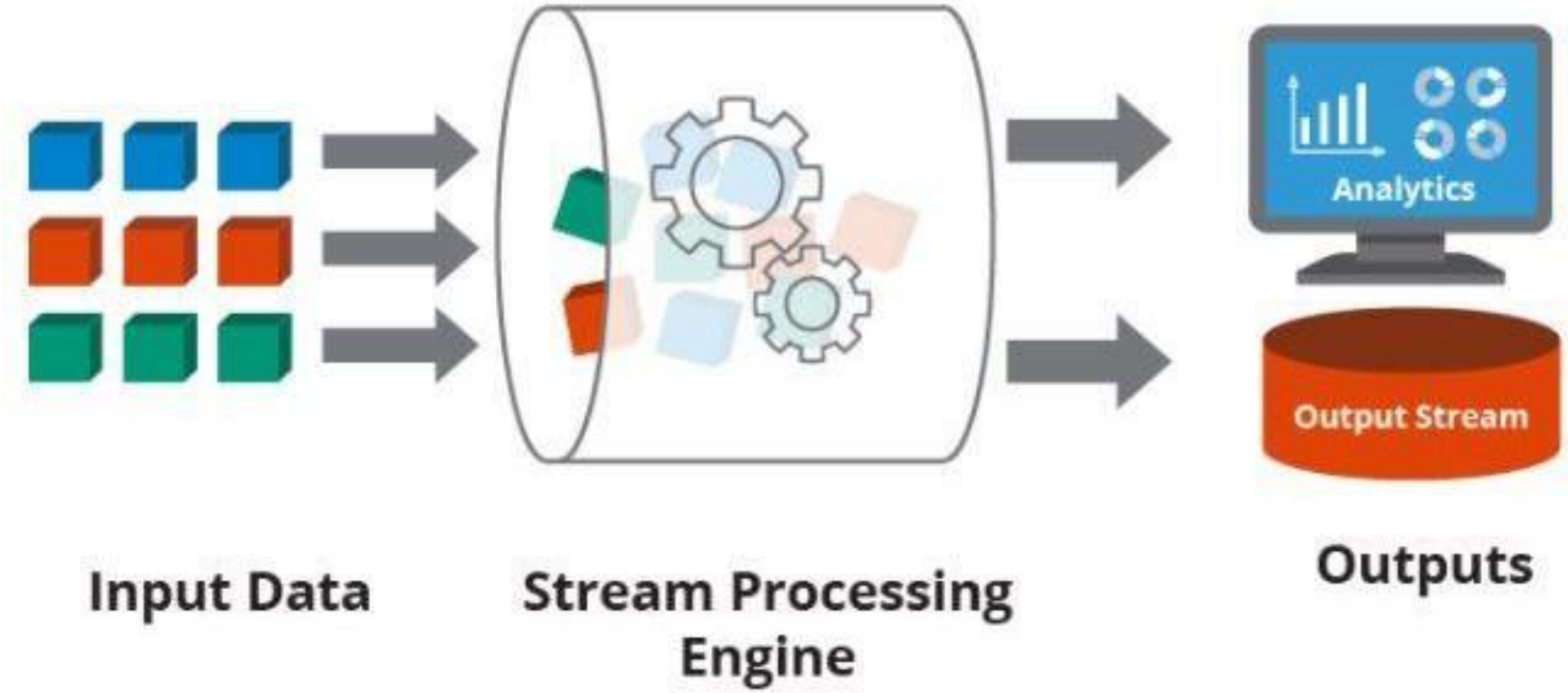
i. **as a string of Absolute Timevalues**

ii. **as a sequence of Positive Real-time Intervals**

Stream Processing

- Stream processing is a one-pass data-processing paradigm that always keeps the data in motion to achieve low processing-latency.
- Stream processing supports message aggregation and delivery;
- Also it is capable of performing real-time asynchronous computation.
- The most important feature of the streaming paradigm is that it does not have access to all data;
- By contrast, it normally adopts the one-at-a-time processing model, which applies standing queries or established rules to data streams in order to get immediate results upon their arrival.

Stream Processing



..Stream Processing

Dedicated Logic-processing System:

- **All of the computation is handled by the continuously dedicated logic-processing system; which is a Scalable, Highly Available, and Fault-tolerant.**
- As a consequence of the timeliness requirement, computations for analytics and pattern recognition should be relatively simple and generally independent.

Comparison of the Stream Model & Batch Model

Aspects	Stream Model	Batch Model
Management target	Transient streams	Persistent data batch and relations
Amount of data	Possibly infinite	Finite
Processing model	In-memory processing	Store-then-process and in-memory processing
Query model	Continuous and standing-by query	One-time query
Access model	Sequential access	Random access
Result repeatability	Nearly impossible	Easy
Pattern of result update	Incremental update	Global update
Focus of processing	Low latency and high throughput	High accuracy and comprehensiveness

Transient → Temporary/ In a short time

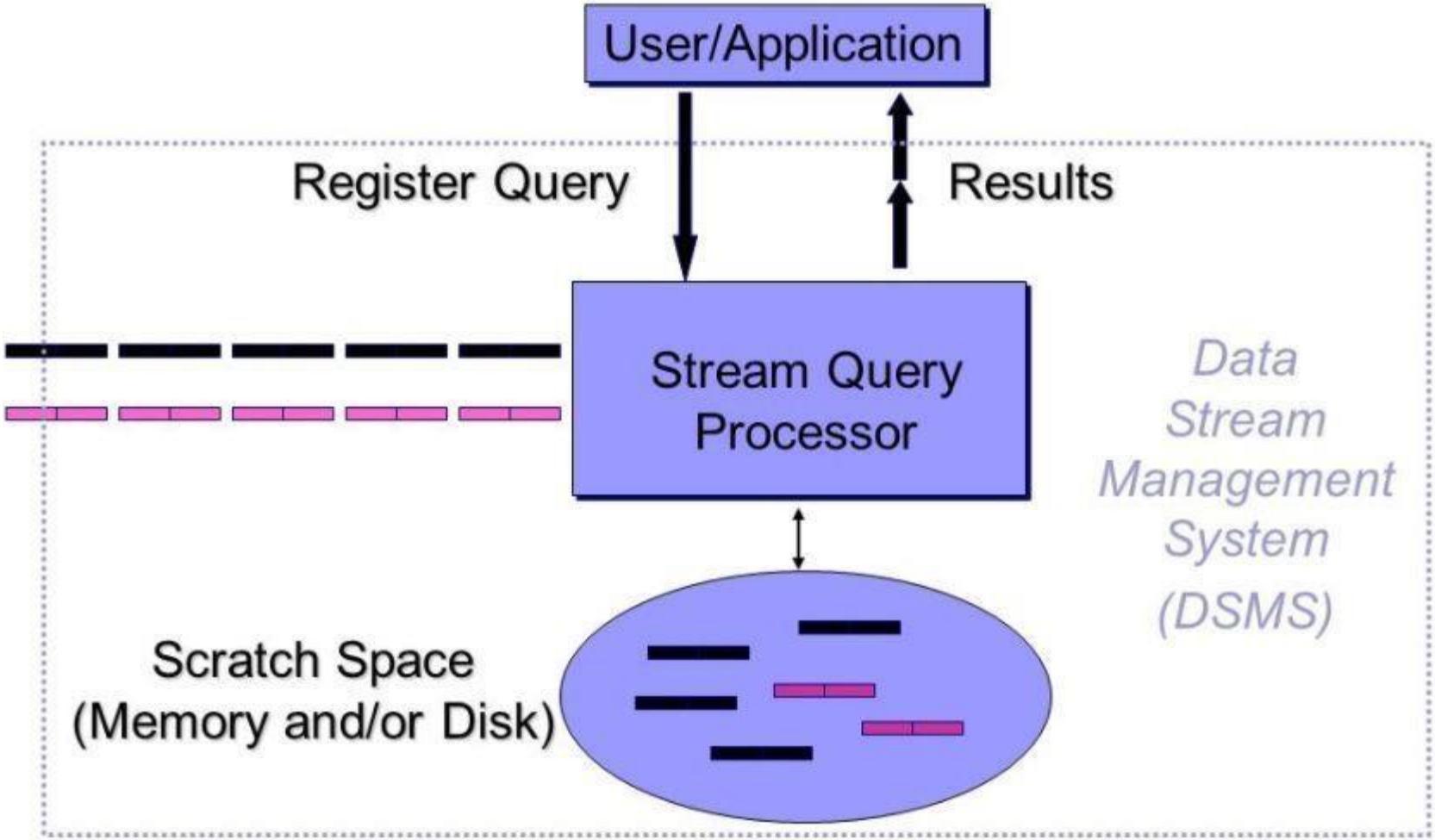
..Stream Processing

- ✓ When it comes to the application of stream processing, we have identified two utterly different types of use cases
  1. Data Stream Management System (DSMS)
  2. Complex Event Processing (CEP)

i. Data Stream Management System (DSMS)

- Similar to the Traditional DBMS.
- Goal is also to manipulate a huge amount of available data to constitute data synopsis, schema, or some other mathematical or statistical model that is easy to understand and interpret.
- Specifically, data streams within the DSMS are joined, filtered, and transformed with the use of continuous and long-standing queries
- However, since the throughput requirement of stream processing has soared during the recent decade and the corresponding DSMS has become increasingly distributed

Throughput→ measure of how many units of information a system can process in a given amount of time.


User/Application
Register Query
Results
Stream Query Processor
Data Stream Management System (DSMS)
Scratch Space (Memory and/or Disk)

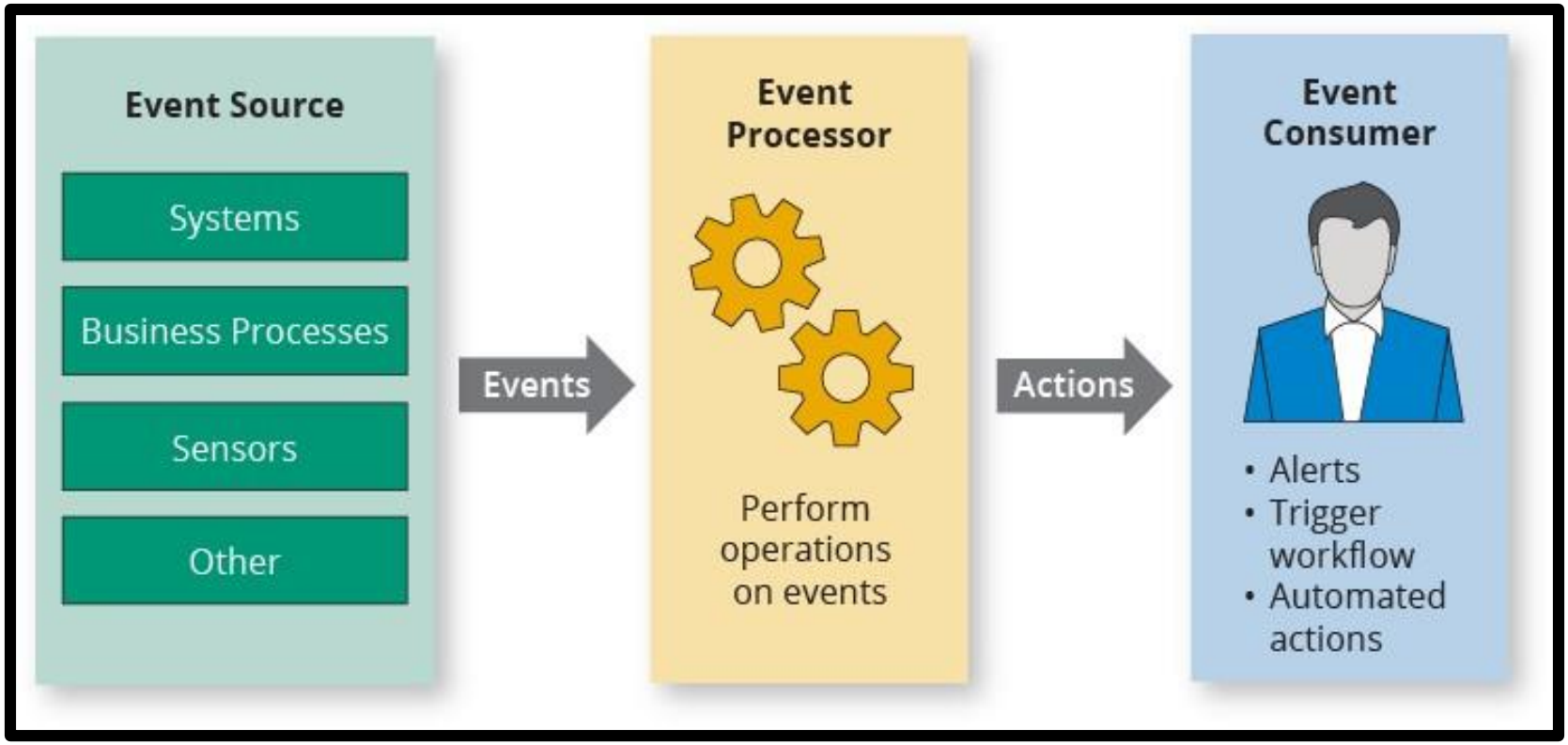
- Sticking to such a declarative model makes it painful to horizontally scale, and even harder to maintain the required availability and fault- tolerance ability
- Therefore, adopts the imperative way to implement long-time queries, by using the provided programming API.
- Where a segment of code is performed upon the arrival of each incoming data element to compose the whole-analysis logic.
- A typical use-case of DSMS includes:
  - Face recognition from a continuous video stream&
  - Calculation of user preference according to his or her click history

DBMS & DSMS

Database management system (DBMS)	Data stream management system (DSMS)
Persistent data (relations)	volatile data streams
Random access	Sequential access
One-time queries	Continuous queries
(theoretically) unlimited secondary storage	limited main memory
Only the current state is relevant	Consideration of the order of the input
relatively low update rate	potentially extremely high update rate
Little or no time requirements	Real-time requirements
Assumes exact data	Assumes outdated/inaccurate data
Plannable query processing	Variable data arrival and data characteristics

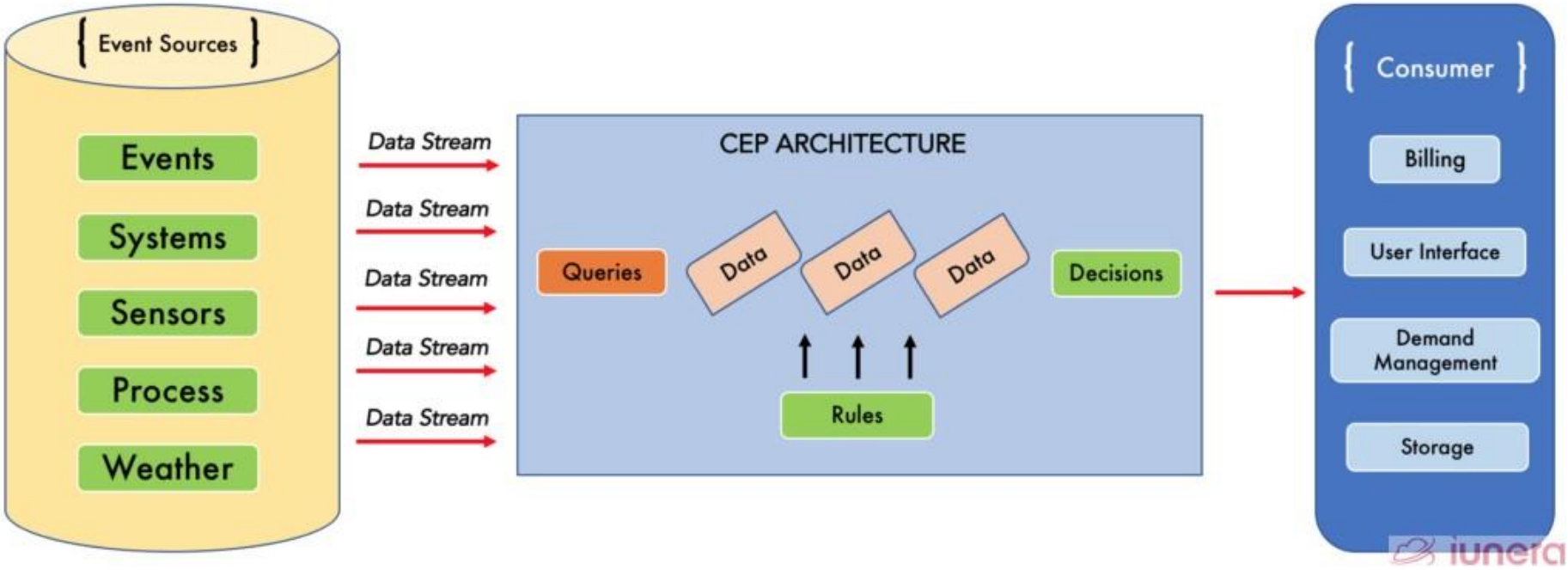
I. Complex Event Processing (CEP)

- It is essentially tracking and processing streams of raw events in order to derive significant events and identify meaningful insights from them.



I. Complex Event Processing (CEP)

- There are several techniques being used to achieve that goal.
- The most notable one is to implement and configure the processing logic as a set of inferring rules in the knowledgebase so that they could be used in the decision-making process of identifying complex patterns.



I. Complex Event Processing (CEP)

- To define and preserve the mutual relationship of events, various types of event-processing languages have been proposed
- Besides, CEP systems normally require that the maintenance of state and the preservation of event relationship, which makes the microbatch model a preferable option compared to the one-at-a-time model.
- In contrast to the primary goal of DSMS, which performs stream analytics at a geographically concentrated place;
- The major concern of CEP is to infer the needed insight from the vast volume of raw events to stream as fast as possible.
- Therefore, the computation complexity of CEP logic is usually lower than that of DSMS.

Table 8.2 Differences Between Two Use-Cases of Stream Processing: DSMS and CEP

Aspects	DSMS	CEP
Processing target	Continuous streams of data	Discrete events
Typical data sources	Video or audio stream, user clicks, social media context.	Sensory information System and service logs
Data variety	Structured, semi/unstructured	Normally structured
Logic implementation	Continuously queries	Event-matching rules or state automaton
Amount of applied logic	Small	Large
Typical application scenario	Quantitative analytics	Qualitative inference
Scalability	Horizontal scale-out	Vertical scale-up
Preferred venue of processing	Collect and aggregate information to a single location to achieve centralized processing	Amortize the processing task throughout the data chain and bring the computation near the data source to relieve the network overhead
Notification of decision	Usually provide analytics result for another system to make a decision	Make decision based on detected insight and inform the outside world as fast as possible

✓ ***Apache Storm** can easily **fulfill the requirement of CEP by using the microbatch paradigm** to become a typical event-processing platform that is capable of identifying meaningful patterns from incoming raw events.*

The Characteristics Of Stream Data In IoT

1. Timeliness and Instantaneity

- **Ensuring the timeliness of processing requires the ability to collect, transfer, process, and present the stream data in real-time**
- The data generation in IoT environments mainly depends on the status of data sources
- Therefore, it is necessary to build an adaptive platform that can elastically scale with respect to fluctuating processing demands, and still remain portable and configurable in response to the continuously shifting processing needs

2. Randomness and Imperfection

- Randomness and data imperfection are two direct consequences of the dynamic nature of stream data.
- The data generation process may induce randomness because the data sources are normally independently installed in different environments, which makes it nearly impossible to guarantee the sequence of data arrival across different streams
- The data transmission process can also result in disorder and other defections, as some tuples may be lost, damaged, or delayed due to the constantly changing network conditions

3. Endlessness and Continuousness

- As long as the data sources are alive and the stream-processing system is properly functioning, newly generated data will be continuously appended to the data channel until the whole application is explicitly turned off.

4. Volatility and Unrepeatability

- Most of the stream data will be discarded once they have finished traversing through the streamprocessing system, which makes the existence of data quite volatile.
- Even if the data sources are able to replay the data stream upon the retransmit request, the new stream is unlikely to be exactly the same as the previous one

Table 8.3 Characteristics of Stream Data and the Corresponding Processing Requirements

Characteristics	Corresponding Requirement
Timeliness and instantaneity	1. Data cannot be detained in any phase of the processing chain, so there should be a comprehensive data-collection subsystem working as a driving force that powers the data in motion once they are generated. 2. For compute-intensive applications, a data aggregation subsystem is needed to gather the collected data for centralized processing. 3. Each phase of the processing chain is preferable to be horizontal scalable in order to keep pace with the fluctuated workload.
Randomness and imperfection	1. For cleansing and coordination purposes, data should be first buffered in a message subsystem before being processed. 2. A declarative or imperative CLPS is responsible for implementing the application logic and handling possible data-stream imperfections.
Endlessness and continuousness	1. The storage subsystem can only be used as an assistance component that preserves the data synopsis or the query results. 2. Ensuring the availability is one of the core design principles due to the continuousness of workload.
Volatility and unrepeatability	1. The data value and insights discovered from the streams should be immediately submitted to other services or presented to users through a presentation subsystem. 2. The fault-tolerance ability is another system design principle, as it is costly or even impossible to replay the incoming stream during the recovery of system failures.

General Architecture Of A Stream-processing System IN IoT

- A stream-processing architecture should include an integral data- processing chain that covers the whole lifespan of data (from its generation up to its consumption)
- Fig. 8.2 presents a general architecture of a stream-processing system.
- This architecture breaks down the whole data-processing chain into several stages according to the functionality and target
- Identified six separate streaming components which are responsible for data generation, collection, buffering, logic processing, storage, and presentation, respectively

General Architecture Of A Stream-processing System IN IoT

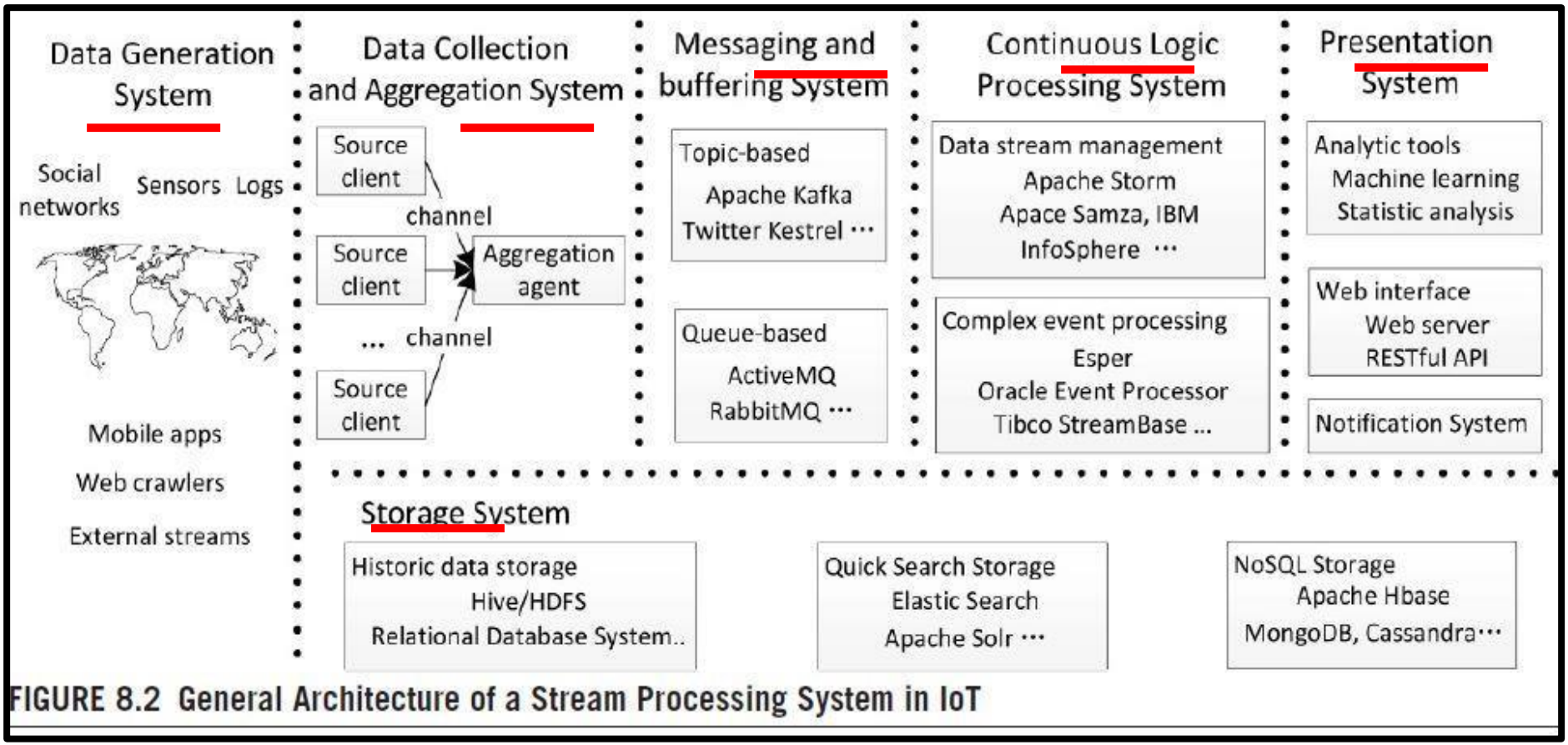


FIGURE 8.2 General Architecture of a Stream Processing System in IoT

1. Data-generation System

- The data-generation system denotes the spectrum of data sources that continuously produce raw information for the data-processing chain.

- We can still categorize the generated data into three types:

i. *Static data:*, refers to the long-term information that has already been stored in remote locations

ii. *Centralized stream data:*, is a special type of stream that only comes from a single centralized data source

iii. *Distributed stream data:* is the most common data type used in IoT applications, where data of this type dynamically come from various distributed places in heterogeneous formats, such as sensory information from sensor networks

2) Data Collection and Aggregation System

- To collect and aggregate different types of data, various forms of source clients are independently installed, while several aggregation channels are provided to gather these stream data into a centralized buffer, using hierarchical aggregation agents.

3) Messaging and Buffering System

- Plays the role of a message broker in the whole data-processing chain
- There are two types of message buffers;

1. topic-based → which support a higher-level programmability;
2. queue- based → mainly optimized for performance concerns

4) Storage System

- Supportive components for a stream-processing architecture.
- Keeping all the data, that need to be stored are either established knowledge, which can guide the future processing, or meaningful data synopsis, which might inspire the future interest of users.

5) Continuous Logic Processing System (CLPS)

- It is responsible for processing aggregated data according to the designated continuous logic, Which could either come from the Data Stream Management or Complex Event Processing background

6) Presentation System

- Supportive components for a stream-processing architecture
- Serves as an interface for immediately hands over the data value to the higher level analytic tools, or directly delivers the results or notifications to the end users
- It is also responsible for receiving search-command or query updates from the external environment so that it can make the stream- processing system more adaptive and responsive

Continuous Logic Processing System

- The origin of the CLPS dates back to the beginning of this century.
- The first generation of CLPS, pioneered by NiagaraCQ and STREAM is only suited for certain processing scenarios in which only a small amount of data are generated.
- In addition to that, the types of operations supported by these prototypes are also limited.
- They are usually used as functional extensions of the existing Data Base Management Systems (DBMS).
- NiagaraCQ → Defines a simple command-language to create and drop continuous queries over XML files.
- STREAM → Directly supports SQL-like query language.

Continuous Logic Processing System

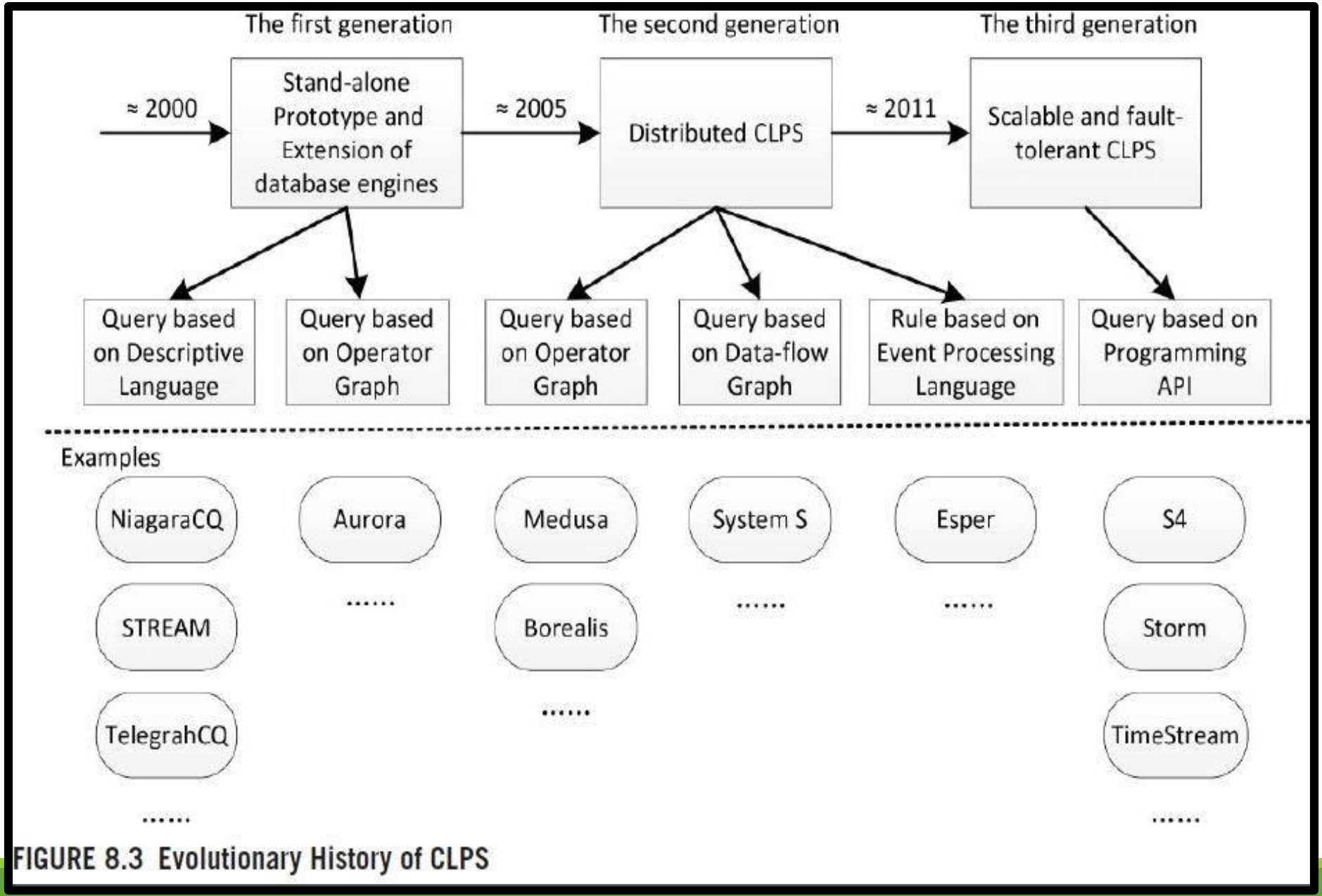


FIGURE 8.3 Evolutionary History of CLPS

Continuous Logic Processing System

- **TelegraphCQ** is developed on PostgreSQL to cope with the high value and diverse data streams, and enables the possibility of adaptive querying.
- **Aurora** is the last breakthrough founded in the first generation of CLPS, here the continuous queries are implemented by explicit operator graphs rather than declarative query languages such as SQL.
- The project **Medusa** is an extension to Aurora, which leads to a scalable and QoS-oriented architecture.
- **Borealis** engine was developed on top of Medusa in order to integrate some advanced capabilities, including dynamic query modification, result revision, and flexible monitoring.

- System S developed by IBM, proposed a query model based on data- flow graph to hide the implementation details as much as possible.
- The core objective is to achieve highly scalable, resource-efficient processing with a balanced resource-allocation mechanism.
- Esper is suitable for distributed event processing that has different types of events defined.
- S4 developed by Yahoo! is generally perceived as the first CLPS that meets the criteria of fully scalable and fault-tolerant.
- Storm supports almost arbitrary programming languages such as Clojure, Java, Ruby, and Python to implement the spouts and bolts, which are the logical operations in Storm.
- TimeStream written in C# by Microsoft, it is able to handle an aggregation data-source with a data generation speed of 700,000 URLs per s.

we compare these alternatives of CLPS in terms of:

i. **System Architecture:** it outlines internally how a CLPS is organized and coordinated.

ii. **Data Transmission**: the way that streaming data feeds the processing system. Pull-based means that CLPS is responsible for actively fetching data, whereas push-based means passive message- reception.

iii. **Development Language**: which languages are being used to develop the CLPS.

iv. **Programming:** which components need to be programmed to apply the continuous logic.

v. **Partitioning and Parallelism:** how data is partitioned to achieve processing parallelization.

vi. **Accurate Recovery:** whether the CLPS is able to accurately reproduce the same processing result when failures occur to the system.

vii. **State Consistency:** whether the system is able to ensure the consistency state for all of the participating components during the processing procedure.

Table 8.4 Comparison Between State-of-the-Art CLPS Implementations

Aspects	Apache Storm	S4	Spark Streaming	Apache Samza	Apache Flink	Esper
System architecture	Master–slave	Symmetric	Master–slave	Master–slave	Master–slave	Master–slave
Data transmission	Pull-based	Push-based	1. Push-based with flume 2. Pull-based with a custom sink	Pull-based	Push-based	Pull-based
Develop language	Clojure	Java	Scala, Java	Java	Java, Scala	Java, C#
programming	Spouts and Bolts	Processing elements	Distributed datasets	Samza job	Data stream and transformations	EPL
Partitioning and parallelism	Sending to different tasks	Based on key-value pairs	Sending to different tasks	Sending to different tasks	Sending to different tasks	Partition based on context
Accurate recovery	Yes, with trident	No	Yes	Yes	Yes	No
State consistency	No	No	Yes, with state DStream	Yes, with embedded key-value store	Yes, with asynchronous distributed snapshots	No

Challenges in stream -processing systems

- The current stream-processing systems have been greatly improved to cater to the emerging needs of IoT applications

---

- A **stream-processing system** now should **satisfy the following criteria:**

  1) **Horizontal scalability to accommodate different sizes of processing needs**

  2) **Easy to program and manage**

  3) **Capable of dealing with possible hardware faults**

- **The following aspects summarize the challenges that still need to be further addressed**

  - **Scalability**
  - **Robustness**
  - **SLA-Compliance**
  - **Load Balancing**

Scalability

- Scalability does not just refer to the ability to expand the system to catch up to the ever-increasing data streams, so that the promise of the Quality of Service (QoS) or Service Level Agreement (SLA) could be honored
- Elasticity, the ability to dynamically scale to the right size on demand, is the future and advanced form of scalability
- The user requirement may change over time, the system should dynamically provision new resources by taking into account the characteristics of the available hardware infrastructure, and free up some of them when they are no longer needed

Robustness

- Fault-tolerance is a common place topic when it comes to the design and implantation of stream processing systems
- Its availability is one of the most crucial prerequisites to guarantee the correctness and significance of real-time processing
- Designing a hybrid and configurable fault-tolerance mechanism that is capable of recovering the system from unforeseeable failures is an open research-question left to be answered

SLA-Compliance

- It depends on which platform the system is running on, and how stakeholders are involved
- But an inherent requirement is to achieve cost-efficiency, which minimizing the monetary cost for the users, as well as reducing the operational cost for the provider (possibly data centers)

Load Balancing

- The major target of which is to normally improve the performance of the system, especially by maximizing the throughput
- A wrong balancing decision may lead to unnecessary load-shedding, dropping arrived messages when the system is deemed to be overloaded, which impairs the veracity of the processing result