Streaming processing systems

Context

- With the popularisation of the IoT, the no. intelligent devices used for monitoring, managing, and servicing has rapidly increased.
- The interconnected data sources generate fresh data continuously, forming a large number, or a massive flow, of data streams that will eventually overwhelm the traditional data management systems
- Meanwhile, the evergrowing data generation has been accompanied by the escalating demands for low-latency data processing.

Stream processing

- The desire of fast data analysis gives birth to the emergence of stream processing, *a new in-memory processing paradigm that allows for the collection, analysis, and visualisation of streaming data with only seconds or milliseconds latencies*.
- Stream processing is a paradigm to handle data streams upon arrival, powering latency-critical application such as fraud detection, algorithmic trading, and health surveillance

Particularities of stream processing

- Unlike the traditional store-first, process-later batch paradigm, stream processing continuously consumes incoming data to provide immediate insights
  - The incoming data are handled upon arrival, with the results being incrementally updated while the data flow through the system.
- Presented with only limited resources to handle continuous inputs, stream processing has no random access to the whole stream
  - Instead, it installs processing logic over time- or buffer-based windows, conducting lightweight and independent computations over recently arriving data.
  - In this way, the strict latency requirement can be met by proper workload balancing and processing parallelisation on a host of distributed resources

Distributed stream processing or splitting

- Stream processing needs specific SLAs on end-to-end latency, sustained stream throughput, and processing semantic guarantee to cope with the dynamic nature of input streams and the shared nature of the infrastructure
- The core concept behind distributed stream processing engines is the processing of incoming data items in real time by modelling a data flow in which there are several stages which can be processed in parallel.
- Other techniques include splitting the data stream into multiple sub-streams and redirecting them into a set of networked nodes

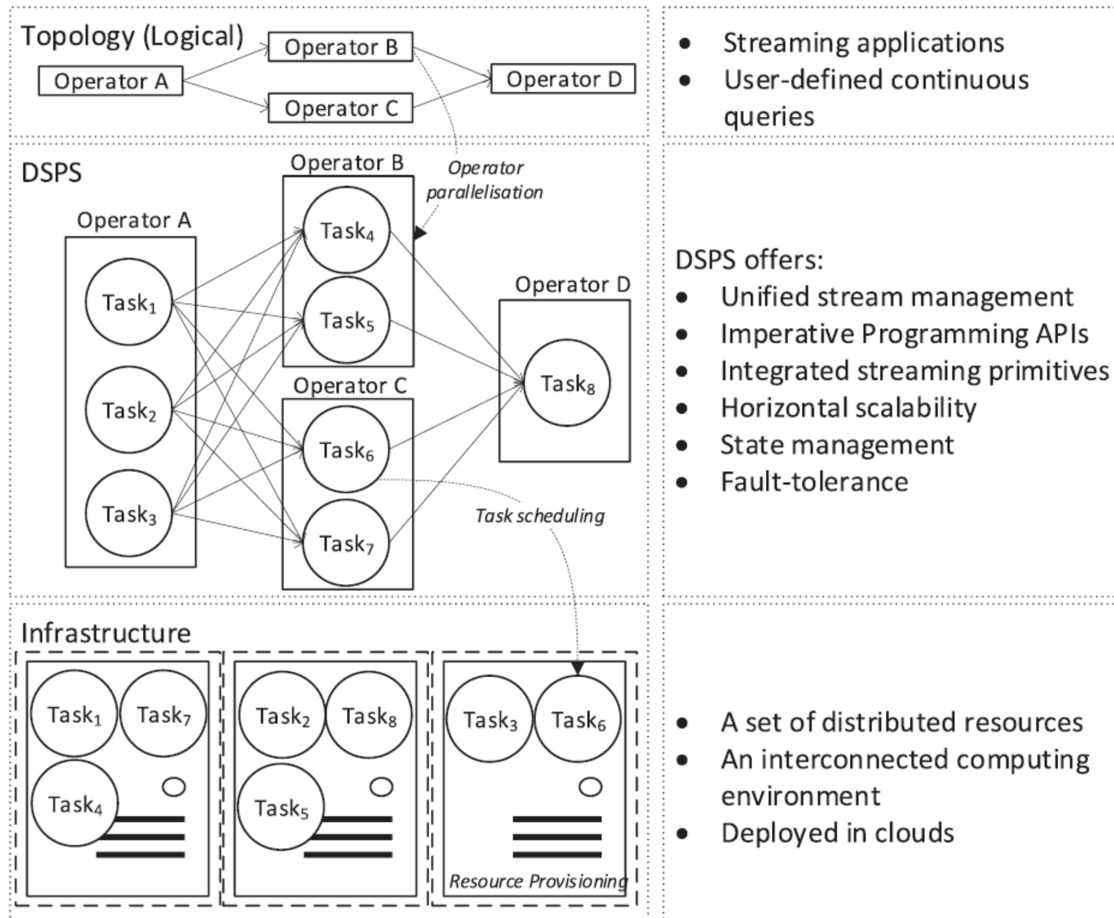
Why stream processing & resource management

- there are a variety of Distributed Stream Processing Systems (DSPSs) that facilitate the development of streaming applications
- resource management and task scheduling is not automatically handled by the DSPS middleware and requires a laborious process to tune toward specific deployment targets

Streaming system: application+DSPS+infrastructure

- From a structural perspective, a DSPS works as the middleware of a distributed system, offering
  - unified stream management,
  - imperative application programming interfaces (APIs), and
  - a set of streaming primitives to simplify the application implementation.
- State-of-the-art DSPSs: Apache Storm and Apache Flink
  - further provide transparent fault-tolerance, scalability, and state management for the upper layer applications, while abstracting away the complexity of coordinating distributed resources.
- A typical streaming system is thus a three-tier structure comprising:
  - user-applications, DSPS, and the underlying infrastructure.

Hierarchical structure of a streaming system



X.Liu, R. Buyya. Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions. ACM Comput. Surv. 53, 3, (May 2021),. DOI: 10.1145/3355399

Deploy a streaming system

- labour-intensive task to deploy a streaming system in a distributed environment satisfying certain Quality of Service (QoS) requirements with minimal resource cost
- Three decisions:
  - (1) resource provisioning—determining the composition of the processing infrastructure,
  - (2) operator parallelisation—configuring the degree of parallelism for streaming logic, and
  - (3) task scheduling— deciding the placement of streaming tasks on distributed resources
- Cloud computing offers a scalable&elastic resource pool

(1) Resource provisioning

- describes the activities to estimate, select, and allocate appropriate resources from the service provider to constitute the interconnected stream processing environment.
- *Resource estimation:*
  - Estimate the type and amount of resources needed by the system to meet its performance and cost targets articulated in the SLA.
  - Can be derived from the analysis of historical data as well as the prediction of future workload
  - Its accuracy is often affected by the instantaneous, unexpected fluctuation of inputs and system performance variations due to the dynamic nature of data streams.
- *Resource adaptation:*
  - the real resource demands can fluctuate along with the varying workload, or remain vague and unclear even after the system is brought online.
  - finding the right point in time to scale in/out and choosing the right adaptation scheme remains a huge challenge.
  - The profitability of adaptation is affected by a no. factors such as the selected billing model.
  - The non-negligible network latency must be taken into consideration when performing system adaptation in a distributed manner

(2) Operator parallelisation

- divides a parallel operator into several functionally equivalent replicas, each handling a subset of the whole operator inputs to accelerate data processing
- *Parallelism calculation*:
  - require accurate profiling of stream workload and probing the processing capability of each task.
  - the number of cores/threads in a CPU confines the maximum degree of runtime parallelism
- *Parallelism adjustment*:
  - Over-parallelisation and under-parallelisation can occur at runtime as a result of workload change or resource adaptation.
  - Challenge: monitor and profile streaming tasks at a fine-grained level to reveal the true performance bottleneck of the application.
- *Balancing data source (inject data in graph)/sinks (peripheral operator only consume):*
  - needs to be fine-tuned as their performances are correlated due to the producer and consumer communication model in the streaming system.
  - An overly powerful data source may cause severe backlogs in data sinks, whereas an inefficient data source would starve the subsequent operators and encumber the overall throughput

(3) Task scheduling

- dynamically maps streaming tasks resources, such that data streams are partitioned and processed at different locations simultaneously and independently.
- the load balancing of stream routing relies on the DSPS to properly partition data streams among the streaming tasks belonging to the same operator
- Task scheduling for stream processing systems is similar to workflow scheduling for batch processing systems
- Objectives:
  - Minimising inter-node communication
  - Mitigating resource contention
  - Performance-oriented scheduling

Apache Storm

- Distributed stream processing engine used by Twitter following extensive development
- Its initial release was 17 September 2011, and by September 2014 it had become open-source
- used by companies such as Groupon, Yahoo!, Spotify, Verisign, Alibaba, Baidu, Yelp, and many more
- the defined topology acts as a distributed data transformation pipeline.
- the programs in Storm are designed as a topology in the shape of DAG, consisting of ‘spouts’ and ‘bolts’

Spouts

- ‘Spouts’ read the data from external sources and emit them into the topology as a stream of ‘tuples’.
- This structure is accompanied by a schema which defines the names of the tuples’ fields.
- Tuples can contain primitive values such as integers, longs, shorts, bytes, strings, doubles, floats, booleans, and byte arrays.
- Additionally, custom serializers can be defined to interpret this data.

Bolts

- The processing stages of a stream are defined in ‘bolts’ which can perform data manipulation, filtering, aggregations, joins, and so on.
- Bolts can also constitute more complex transforming structures that require multiple steps (thus, multiple bolts).
- The bolts can communicate with external applications such as databases and Kafka queues

Typical examples of Storm's usage

- Processing a stream of new data and updating databases in real time, for example in trading systems wherein data accuracy is crucial;
- Continuously querying and forwarding the results to clients in real time, for example streaming trending topics on Twitter into browsers,
- A parallelization of a computing-intensive query on the fly, i.e., a distributed Remote Procedure Call (RPC) wherein a large number of sets are probed

Problems of Storm & solutions

- Problems
  - Storm topologies, once created, run indefinitely until killed.
    - the inefficient scattering of application's tasks among Cluster nodes has a lasting impact on performance.
  - Storm's default scheduler implements a Round Robin strategy.
  - For resource allocation purposes, Storm assumes that every worker is homogenous.
    - This design results in frequent resource over-allocation and inefficient use of inter-system communications
- Solutions
  - D-Storm from 2017 (academic)
    - Its scheduling strategy is based on a metaheuristic algorithm Greedy, which also monitors the volume of the incoming workload and is resource-aware.
  - Heron has replase Storm in 2018 at Twitter (commercial)
    - new distributed stream processing engine, Heron, which continues the DAG model approach, focuses on various architectural improvements such as reduced overhead, testability, and easier access to debug data.

Streaming and edge computing

- processing of continuous data streams as an ideal edge application, especially when those data streams are on end user premises and have a low access rate (e.g., video surveillance)
- Promise of edge computing: less bandwidth utilization in the core network
  - Typically, all raw values would be streamed to the cloud; however, given the increase in data, this might overload the core network.
  - This is relevant since wide-area network bandwidth remains a scarce resource
  - The same holds true for many of today's wireless access networks
  - Especially large, continuous data streams can be a burden on backhaul networks.
  - Distributed processing and aggregation of data streams along the path to the consumer can help to mitigate this.

Filtering at edge

- to discard irrelevant data
- since not all data is equally important, bandwidth savings can be achieved by discarding irrelevant data before it is transmitted for further processing.
- example:
  - thresholding of temperature readings in an application where an alarm should be raised when a certain value is exceeded
  - temperature readings are irrelevant as long as they are within the normal range and thus need not be transmitted.

Pre-processing at edge

- Data is transformed from one representation to another.
- Besides saving bandwidth, reducing data locally can also help to save energy and reduce local storage needs.
- Discarding data could be interpreted as a special case of such a transformation
- Other possible transformations: the aggregation of data streams over time, data compression, data alteration, or bridging between formats.
- For instance, real-time video analysis (a likely killer app for edge computing),
  - only forwarding results of the analysis, e.g., the number of objects in the frame, instead of entire video streams or pre-process data for a face recognition application.
- In case of time-critical data stream processing apps, distributing operations entirely at the edge can reduce end-to-end latencies substantially