Introduction to Data Stream Processing

Corso di Sistemi e Architetture per Big Data
A.A. 2016/17

Valeria Cardellini
The reference Big Data stack

- High-level Interfaces
- Data Processing
- Data Storage
- Resource Management
- Support / Integration
Why data stream processing?

• Applications such as:
  – Sentiment analysis on multiple tweet streams @Twitter
  – User profiling @Yahoo!
  – Tracking of query trend evolution @Google
  – Fraud detection
  – Bus routing management @city of Dublin

• Require:
  – Continuous processing of unbounded data streams generated by multiple, distributed sources
  – In (near) real-time fashion
Why data stream processing?

• In the past years data stream processing (DSP) was considered a solution for very specific problems (e.g., financial tickers)

• But now we have (and will have) more general settings
  – E.g., Internet of Things
Why data stream processing?

• Decrease the overall latency to obtain results
  – No data persistence on stable storage
  Recall “Latency numbers every programmer should know”!

Valeria Cardellini - SABD 2016/17
Traditional DSP challenges

• Stream data rates can be high and data arrive in large volumes
  – High resource requirements for processing (clusters, data centers, distributed Clouds)

• Processing stream data has real-time aspects
  – Stream processing applications have QoS requirements, e.g., end-to-end latency
  – Must be able to react to events as they occur
New challenge for large-scale DSP

• Goals: increase scalability and reduce latency
• How? Rely on distributed and near-edge computation
Data stream

• “A data stream is a **real-time, continuous, ordered** (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety.

Queries over streams run continuously over a period of time and incrementally return new results as new data arrive.”

DSP application model

- A DSP application is made of a network of operators (processing elements or PE) connected by streams, at least one data source and at least one data sink.

- Represented by a directed graph:
  - Graph vertices: operators
  - Graph edges: streams

- Graph can be cyclic:
  - Some systems only support directed acyclic graph (DAG)

- Graph topology rarely changes
DSP programming model

- Data flow programming
- **Flow composition**: techniques for creating the topology associated with the flow graph for an application
- **Flow manipulation**: the use of processing elements (i.e., operators) to perform transformations on data
Data flow manipulation

• How the streaming data is manipulated by the different operators in the flow graph?

• Operator properties:
  – Operator type
  – Operator state
  – Windowing
DSP operator

• A self-contained processing element that:
  – transforms one or more input streams into another stream
  – can execute a generic user-defined code
    • Algebraic operation (filter, aggregate, join, ..)
    • User-defined (more complex) operation (POS-tagging, …)
  – can execute in parallel with other operators
Types of operators

- **Edge adaptation**: converting data from external sources into tuples that can be consumed by downstream operators
- **Aggregation**: collecting and summarizing a subset of tuples from one or more streams
- **Splitting**: partitioning a stream into multiple streams
- **Merging**: combining multiple input streams
Types of operators

- **Logical and mathematical operations**: applying different logical processing, relational processing, and mathematical functions to tuple attributes
- **Sequence manipulation**: reordering, delaying, or altering the temporal properties of a stream
- **Custom data manipulations**: applying data mining, machine learning, ...
DSP operator: state

• The operator can be stateless or stateful
• **Stateless**: know nothing about the state (e.g., filter, map) and thus process tuples independently of each other, independently of prior history, or even from the order of arrival of tuples
  – Easily parallelized
  – No synchronization in a multi-threaded context.
  – Restart upon failures without the need of any recovery procedure.
DSP operator: state

- **Stateful**: keep some sort of state and thus involve maintaining information across different tuples to detect complex patterns.
  - E.g., some aggregation or summary of processed elements, or state-machine for detecting patterns for fraudulent financial transaction
  - State might be shared between operators
  - A subset of recent tuples kept in a window buffer
Sliding windows

- **Window**: a buffer associated with an input port to retain previously received tuples
- **Eviction data policy**: how many data items should we keep in the buffer and process each time?
  - *Count-based* window, e.g., last \( n \) items held in the window
  - *Time-based* window, e.g. from \([t-T]\) to \([t]\)
Sliding windows

• How often should we evaluate the window?
  – **Eager approach**: output new result items as soon as available (but can be difficult to implement efficiently)
  – **Lazy approach**: slide window by $s$ seconds (or $m$ items)
How to define a DSP application

- **Formal language**: more rigor and expressiveness
  - Declarative language: specify the result (SQL-like); e.g., IBM Streams Processing Language
  - Imperative language: specify the composition of basic operators, e.g., SQuAI (Stream Query Algebra) used in Aurora/Borealis

- **Topology description**: more flexibility
  - Explicitly define the operators (built-in or user-defined) and the links through a directed graph (often called topology)
“Hello World”: a variant of WordCount

• Goal: emit the top-k words in terms of occurrence when there is a rank update

• Where are the bottlenecks?

• How to scale the DSP application in order to sustain the traffic load?
“Hello World”: a variant of WordCount

- The usual answer: replication!
- Use data parallelism
Example of DSP application: DEBS’14 GC

http://debs.org/?p=75

• Real-time analytics over high volume sensor data: analysis of energy consumption measurements for smart homes
  – Smart plugs deployed in households and equipped with sensors that measure values related to power consumption

• Input data stream:
  2967740693, 1379879533, 82.042, 0, 1, 0, 12

• Query 1: make load forecasts based on current load measurements and historical data
  – Output data stream:
    ts, house_id, predicted_load

• Query 2: find the outliers concerning energy consumption
  – Output data stream:
    ts_start, ts_stop, household_id, percentage
Example of DSP application: DEBS’15 GC

http://debs.org/?p=56

• Real-time analytics over high volume spatio-temporal data streams: analysis of taxi trips based on data streams originating from New York City taxis

• Input data streams: include starting point, drop-off point, corresponding timestamps, and information related to the payment

```
07290D3599E7A0D62097A346EFCC1FB5,E7750A37CAB07D0DFF0AF7E3573AC141,2013-01-01 00:00:00,2013-01-01 00:02:00,120,0.44,-73.956528,40.716976,-73.962440,40.715008,CSH,3.50,0.50,0.50,0.00,0.00,4.50
```
Example of DSP application: DEBS’15 GC

http://debs.org/?p=59

- **Query 1**: identify the top 10 most frequent routes during the last 30 minutes
- **Query 2**: identify areas that are currently most profitable for taxi drivers
- Both queries rely on a sliding window operator
  - Continuously evaluate the query results
- Use geo-spatial grids to define the events of interest
Example of DSP application: DEBS’16 GC

http://debs.org/?p=59

- Real-time analytics for a dynamic (evolving) social-network graph
- *Query 1*: identify the posts that currently trigger the most activity in the social network
- *Query 2*: identify large communities that are currently involved in a topic
Data streaming system

- Distributed system that executes stream graphs
  - continuously calculates results for long-standing queries
  - over potentially infinite data streams
  - using operators
    - that can be stateless or stateful
- System nodes may be heterogeneous
- Must be highly optimized and with minimal overhead so to deliver real-time response for high-volume DSP applications
Operator placement

• Determine, within a set of available distributed computing nodes, those nodes that should host and execute each operator of a DSP application

• See next lesson
Big data centers

• Which frameworks for data stream processing?
• Usually run in locally distributed clusters within large data centers
• Assumptions:
  – Scale out and not scale up
    • Commodity servers
    • Data-parallelism is king
  – Software designed for failure

Source: Google
DSP: processing model

- Two stream processing models:
  - **One-at-a-time**: each tuple is individually sent
  - **Micro-batched**: some tuples are grouped before being sent

<table>
<thead>
<tr>
<th></th>
<th>One-at-a-time (e.g., Apache Storm)</th>
<th>Micro-batched (e.g., Apache Spark)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower latency</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Higher throughput</td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>At-least-once semantics</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Exactly-once semantics</td>
<td>In some cases</td>
<td></td>
</tr>
<tr>
<td>Simpler programming model</td>
<td></td>
<td>✔️</td>
</tr>
</tbody>
</table>


The two approaches are complementary with distinct trade-offs and are suitable to different types of applications.
Apache Storm

- **Apache Storm**
  - Open-source, real-time, scalable streaming system
  - Provides an abstraction layer to execute DSP applications
  - Initially developed by Twitter

- **Topology**
  - DAG of *spouts* (sources of streams) and *bolts* (operators and data sinks)
Stream grouping in Storm

• Data parallelism in Storm: how are streams partitioned among multiple tasks (threads of execution)?

• Shuffle grouping
  – Randomly partitions the tuples

• Field grouping
  – Hashes on a subset of the tuple attributes
Stream grouping in Storm

- **All grouping (i.e., broadcast)**
  - Replicates the entire stream to all the consumer tasks

- **Global grouping**
  - Sends the entire stream to a single bolt

- **Direct grouping**
  - Sends tuples to the consumer bolts in the same executor
Storm architecture

- Master-worker architecture
Storm components: Nimbus and Zookeeper

• Nimbus
  – The master node
  – Clients submit topologies to it
  – Responsible for distributing and coordinating the topology execution

• Zookeeper
  – Nimbus uses a combination of the local disk(s) and Zookeeper to store state about the topology
Storm components: worker

- **Task**: operator instance
  - The actual work for a bolt or a spout is done in the task
- **Executor**: smallest schedulable entity
  - Execute one or more tasks related to same operator
- **Worker process**: Java process running one or more executors
- **Worker node**: computing resource, a container for one or more worker processes
Storm components: supervisor

• Each worker node runs a supervisor
• It receives assignments from Nimbus and spawns workers based on the assignment
• It contacts Nimbus with a periodic heartbeat protocol, advertising the topologies that they are currently running, and any vacancies that are available to run more topologies
Other frameworks

- Apache Flink
- Twitter Heron
- Apache Spark Streaming
  - Reduce the size of each stream and process streams of data (*micro-batch processing*)
- Cloud-based frameworks
  - Google Cloud Dataflow
  - Amazon Kinesis