Extreme Computing

Data streams and low latency processing
DATA STREAM BASICS
What is a data stream?

• Large data volume, likely structured, arriving at a very high rate
  – Potentially high enough that the machine cannot keep up with it
• Not (only) what you see on youtube
  – Data streams can have structure and semantics, they’re not only audio or video

• Definition (Golab and Ozsu, 2003)
  – A data stream is a real-time, continuous, ordered (implicitly by arrival time of explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor it is feasible to locally store a stream in its entirety.
Why do we need a data stream?

• Online, real-time processing

• Potential objectives
  – Event detection and reaction
  – Fast and potentially approximate online aggregation and analytics at different granularities

• Various applications
  – Network management, telecommunications
    Sensor networks, real-time facilities monitoring
  – Load balancing in distributed systems
  – Stock monitoring, finance, fraud detection
  – Online data mining (click stream analysis)
Example uses

- Network management and configuration
  - Typical setup: IP sessions going through a router
  - Large amounts of data (300GB/day, 75k records/second sampled every 100 measurements)
  - Typical queries
    - What are the most frequent source-destination pairings per router?
    - How many different source-destination pairings were seen by router 1 but not by router 2 during the last hour (day, week, month)?

- Stock monitoring
  - Typical setup: stream of price and sales volume
  - Monitoring events to support trading decisions
  - Typical queries
    - Notify when some stock goes up by at least 5%
    - Notify when the price of XYZ is above some threshold and the price of its competitors is below than its 10 day moving average
Structure of a data stream

- Infinite sequence of items (elements)
- One item: structured information, i.e., tuple or object
- Same structure for all items in a stream
- Timestamping
  - Explicit: date/time field in data
  - Implicit: timestamp given when items arrive
- Representation of time
  - Physical: date/time
  - Logical: integer sequence number
Database management vs. data stream management

- Data stream management system (DSMS) at multiple observation points
  - Voluminous streams-in, reduced streams-out
- Database management system (DBMS)
  - Outputs of data stream management system can be treated as data feeds to database
DBMS vs. DSMS

**DBMS**
- Model: persistent relations
- Relation: tuple set/bag
- Data update: modifications
- Query: transient
- Query answer: exact
- Query evaluation: arbitrary
- Query plan: fixed

**DSMS**
- Model: transient relations
- Relation: tuple sequence
- Data update: appends
- Query: persistent
- Query answer: approximate
- Query evaluation: one pass
- Query plan: adaptive
Windows

- Mechanism for extracting a finite relation from an infinite stream
- Various window proposals for restricting processing scope
  - Windows based on ordering attributes (e.g., time)
  - Windows based on item (record) counts
  - Windows based on explicit markers (e.g., punctuations) signifying beginning and end
  - Variants (e.g., some semantic partitioning constraint)
Ordering attribute based windows

- Assumes the existence of an attribute that defines the order of stream elements/records (e.g., time)

- Let $T$ be the window length (size) expressed in units of the ordering attribute (e.g., $T$ may be a time window)

\[ t_i - t_i' = T \]

- Sliding window

\[ t_{i+1} - t_i = T \]

- Tumbling window
Count-based windows

- Window of size $N$ elements (sliding, tumbling) over the stream
- Problematic with non-unique timestamps associated with stream elements
- Ties broken arbitrarily may lead to non-deterministic output
- Potentially unpredictable with respect to fluctuating input rates
  - But dual of time based windows for constant arrival rates
  - Arrival rate $\lambda$ elements/time-unit, time-based window of length $T$, count-based window of size $N$; $N = \lambda T$
Punctuation-based windows

• Application-inserted “end-of-processing”
  – Each next data item identifies “beginning-of-processing”
• Enables data item-dependent variable length windows
  – Examples: a stream of auctions, an interval of monitored activity
• Utility in data processing: limit the scope of operations relative to the stream
• Potentially problematic if windows grow too large
  – Or even too small: too many punctuations
Putting it all together: architecting a DSMS
STREAM MINING
Data stream mining

- Numerous applications
  - Identify events and take responsive action in real time
  - Identify correlations in a stream and reconfigure system
- Mining query streams: Google wants to know what queries are more frequent today than yesterday
- Mining click streams: Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- Big brother
  - Who calls whom?
  - Who accesses which web pages?
  - Who buys what where?
  - All those questions answered in real time
- We will focus on frequent pattern mining
Frequent pattern mining

• Frequent pattern mining refers to finding patterns that occur more frequently than a pre-specified threshold value
  – Patterns refer to items, itemsets, or sequences
  – Threshold refers to the percentage of the pattern occurrences to the total number of transactions
    • Termed as support
• Finding frequent patterns is the first step for association rules
  – $A \rightarrow B$: $A$ implies $B$
• Many metrics have been proposed for measuring how strong an association rule is
  – Most commonly used metric: confidence
  – Confidence refers to the probability that set $B$ exists given that $A$ already exists in a transaction
    • $\text{confidence}(A \rightarrow B) = \frac{\text{support}(A \wedge B)}{\text{support}(A)}$
Frequent pattern mining in data streams

• Frequent pattern mining over data streams differs from conventional one
  – Cannot afford multiple passes
    • Minimised requirements in terms of memory
    • Trade off between storage, complexity, and accuracy
    • You only get one look
  • Frequent items (also known as heavy hitters) and itemsets are usually the final output
• Effectively a counting problem
  – We will focus on two algorithms: lossy counting and sticky sampling
The problem in more detail

- **Problem statement**
  - Identify all items whose current frequency exceeds some support threshold $s$ (e.g., 0.1%)
Lossy counting in action

• Divide the incoming stream into windows
First window comes in

- At window boundary, adjust counters
Next window comes in

- At window boundary, adjust counters
Lossy counting algorithm

- Deterministic technique; user supplies two parameters
  - Support $s$; error $\varepsilon$
- Simple data structure, maintaining triplets of data items $e$, their associated frequencies $f$, and the maximum possible error $\Delta$ in $f$ : $(e, f, \Delta)$
- The stream is conceptually divided into buckets of width $w = 1/\varepsilon$
  - Each bucket labelled by a value $N/w$ where $N$ starts from 1 and increases by 1
- For each incoming item, the data structure is checked
  - If an entry exists, increment frequency
  - Otherwise, add new entry with $\Delta = b_{current} - 1$ where $b_{current}$ is the current bucket label
- When switching to a new bucket, all entries with $f + \Delta < b_{current}$ are released
Lossy counting observations

• How much do we undercount?
  – If current size of stream is $N$
  – …and window size is $1/\varepsilon$
  – …then frequency error $\leq$ number of windows, i.e., $\varepsilon N$

• Empirical rule of thumb: set $\varepsilon = 10\%$ of support $s$
  – Example: given a support frequency $s = 1\%$,
  – …then set error frequency $\varepsilon = 0.1\%$

• Output is elements with counter values exceeding $sN - \varepsilon N$

• Guarantees
  – Frequencies are underestimated by at most $\varepsilon N$
  – No false negatives
  – False positives have true frequency at least $sN - \varepsilon N$

• In the worst case, it has been proven that we need $1/\varepsilon \times \log(\varepsilon N)$ counters
Sticky sampling

- Create counters by sampling
- Maintain exact counts thereafter
Sticky sampling algorithm

- Probabilistic technique; user supplies three parameters
  - Support \( s \); error \( \varepsilon \); probability of failure \( \delta \)
- Simple data structure, maintaining pairs of data items \( e \) and their associated frequencies \( f : (e, f) \)
- The sampling rate decreases gradually with the increase in the number of processed data elements
- For each incoming item, the data structure is checked
  - If an entry exists, increment frequency
  - Otherwise sample the item with the current sampling rate
    - If selected, add new entry; else ignore the item
- With every change in the sampling rate, toss a coin for each entry
  - Decreasing the frequency of the entry for each unsuccessful coin toss
  - If frequency goes down to zero, release the entry
Sticky sampling observations

• For a finite stream of length $N$
• Sampling rate $= \frac{2}{N\epsilon} \times \log \left( \frac{1}{s\delta} \right)$
  – $\delta$ is the probability of failure—user configurable
• Same guarantees with lossy counting, but probabilistic
• Same rule of thumb as lossy counting, but with a probabilistic and user configurable failure probability $\delta$
• Generalisation to infinite streams of unknown $N$
  – (probabilistically) expected number of counters is $= \frac{2}{\epsilon} \times \log \left( \frac{1}{s\delta} \right)$
  – Independent of $N$

• Comparison
  – Lossy counting is deterministic; sticky sampling is probabilistic
  – In practice, lossy counting is more accurate
  – Sticky sampling extends to infinite streams with same error guarantees as lossy counting
STORM AND LOW-LATENCY PROCESSING
Low latency processing

• Similar to data stream processing, but with a twist
  – Data is streaming into the system (from a database, or a network stream, or an HDFS file, or …)
  – We want to process the stream in a distributed fashion
  – And we want results as quickly as possible

• Numerous applications
  – Algorithmic trading: identify financial opportunities (e.g., respond as quickly as possible to stock price rising/falling
  – Event detection: identify changes in behaviour rapidly

• Not (necessarily) the same as what we have seen so far
  – The focus is not on summarising the input
  – Rather, it is on “parsing” the input and/or manipulating it on the fly
The problem

• Consider the following use-case
  • A stream of incoming information needs to be summarised by some identifying token
    – For instance, group tweets by hash-tag; or, group clicks by URL;
    – And maintain accurate counts
  • But do that at a massive scale and in real time
  • Not so much about handling the incoming load, but using it
    – That's where latency comes into play
• Putting things in perspective
  – Twitter's load is not that high: at 15k tweets/s and at 150 bytes/tweet we're talking about 2.25MB/s
  – Google served 34k searches/s in 2010: let's say 100k searches/s now and an average of 200 bytes/search that's 20MB/s
  – But this 20MB/s needs to filter PBs of data in less than 0.1s; that's an EB/s throughput
A rough approach

- Latency
  - Each point 1 – 5 in the figure introduces a high processing latency
  - Need a way to transparently use the cluster to process the stream

- Bottlenecks
  - No notion of locality
    - Either a queue per worker per node, or data is moved around
    - What about reconfiguration?
      - If there are bursts in traffic we need to shutdown, reconfigure and redeploy
Storm

• Started up as backtype; widely used in Twitter
• Open-sourced (you can download it and play with it!)
  – http://storm-project.net/
• On the surface, Hadoop for data streams
  – Executes on top of a (likely dedicated) cluster of commodity hardware
  – Similar setup to a Hadoop cluster
    • Master node, distributed coordination, worker nodes
    • We will examine each in detail
• But whereas a MapReduce job will finish, a Storm job—termed a topology—runs continuously
  – Or rather, until you kill it
Storm topologies

• A Storm topology is a graph of computation
  – Graph contains nodes and edges
  – Nodes model processing logic (i.e., transformation over its input)
  – Directed edges indicate communication between nodes
  – No limitations on the topology; for instance one node may have more than one incoming edges and more than one outgoing edges

• Storm processes topologies in a distributed and reliable fashion
Streams, spouts, and bolts

- **Streams**
  - The basic collection abstraction: an unbounded sequence of tuples
  - Streams are transformed by the processing elements of a topology

- **Spouts**
  - Stream generators
  - May propagate a single stream to multiple consumers

- **Bolts**
  - Subscribe to streams
  - Streams transformers
  - Process incoming streams and produce new ones
Storm architecture

- Nimbus
- Zookeeper
- Supervisor
- Worker
- Bolt
- Spout
- Zookeeper
- Distributed coordination

Storm job topology
Task allocation

Storm cluster master node

www.inf.ed.ac.uk
From topology to processing: stream groupings

• Spouts and bolts are replicated in tasks, each task executed in parallel by a worker
  – User-defined degree of replication
  – All pairwise combinations are possible between tasks
• When a task emits a tuple, which task should it send to?
• Stream groupings dictate how to propagate tuples
  – Shuffle grouping: round-robin
  – Field grouping: based on the data value (e.g., range partitioning)
Zookeeper: distributed reliable storage and coordination

- **Design goals**
  - Distributed coordination service
  - Hierarchical name space
  - All state kept in main memory, replicated across servers
  - Read requests are served by local replicas
  - Client writes are propagated to the leader
  - Changes are logged on disk before applied to in-memory state
  - Leader applies the write and forwards to replicas

- **Guarantees**
  - Sequential consistency: updates from a client will be applied in the order that they were sent
  - Atomicity: updates either succeed or fail; no partial results
  - Single system image: clients see the same view of the service regardless of the server
  - Reliability: once an update has been applied, it will persist from that time forward
  - Timeliness: the clients’ view of the system is guaranteed to be up-to-date within a certain time bound
Putting it all together: word count

// instantiate a new topology
TopologyBuilder builder = new TopologyBuilder();

// set up a new spout with five tasks
builder.setSpout("spout", new RandomSentenceSpout(), 5);

// the sentence splitter bolt with eight tasks
builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("spout"); // shuffle grouping for the output

// word counter with twelve tasks
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word")); // field grouping

// new configuration
Config conf = new Config();

// set the number of workers for the topology; the 5x8x12=480 tasks
// will be allocated round-robin to the three workers, each task
// running as a separate thread
conf.setNumWorkers(3);

// submit the topology to the cluster
StormSubmitter.submitTopology("word-count", conf, builder.createTopology());
Summary

- Introduced the notion of data streams and data stream processing
- Discussed the architecture of a data stream management system
  - Differences to a DBMS
  - Architectural choices
- Described use-cases and algorithms for stream mining
  - Lossy counting and sticky sampling
- Introduced frameworks for low-latency stream processing
  - Storm