Big Data
Data Streaming

Alessandro Margara
alessandro.margara@polimi.it
http://home.deib.polimi.it/margara
Big Data philosophical perspective

• What is more valuable: intelligence or experience?

• Traditional computer science: *logic* (intelligence)
  – Understand the problem
  – Build models and algorithms
  – Answer questions by implementing the model

• New science: *statistics* (experience)
  – Collect data
  – Answer questions with data
    • What did others do?
Data science

• New approach to do science
  1. Collect data
  2. Generate hypotheses
  3. Validate hypotheses
  4. Goto step 1 and 2

• Why is this a good approach?
  – Automated: no thinking

• Why is this a bad approach?
  – No ground truth
  – More data, more errors (heterogeneity)
  – With enough data you can prove (almost) anything
Big Data success story

- Google Translate
  - Collect snippets of translations
  - Match sentences to snippets
  - Continuously debug the system

- Why does it work?
  - There are tons of snippets on the Web
  - There is a ground truth that helps debugging the system
Business perspective

• It is a business model
  – Users pay with data (e.g., Google, Facebook, …)
    • Use service → give data
  – Google and co. sell your data to advertisers
    • The user pays the advertisers indirectly
Technical perspective

- Collect all the data
  - The more the better ➔ statistical relevance
  - Keeping all is cheaper than deciding what to keep

- Decide independently what to do with data
  - Run experiments on data when question arises

- Huge difference to traditional information systems!
  - Decide upfront what data to keep and why
Consequences

• Volume: data at rest
  – It is going to be a lot of data!

• Velocity: data in motion
  – It is going to arrive fast

• Variety: many different formats
  – Different versions, different sources

• Veracity: not always correct
MAP REDUCE
Problem scope

- Need to scale to 100s or 1000s of computers, each with several processor cores

- It is likely that the input data set will not fit a single computer’s hard drive

- A distributed file system (e.g., GFS, HDFS) is typically required
Problem scope

- **Scalability to large data volumes**
  - Scan 1000 TB on 1 node @ 100MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

- **Required functions**
  - Automatic parallelization & distribution
  - Fault-tolerance
  - Status and monitoring tools
  - A clean abstraction for programmers
How MapReduce works?
Example: word count

- **Map**: \((k_1, v_1) \rightarrow \text{list} (k_2, v_2)\)
  
  ```java
  map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value: EmitIntermediate(w, "1");
  ```

- **Reduce**: \((k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)\)
  
  ```java
  reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
  ```
Example: word count

The overall MapReduce word count process

Input

Splitting

Deer Bear River

Car Car River

Deer Car Bear

Mapping

Deer, 1
Bear, 1
River, 1

Car, 1
Car, 1
River, 1

Deer, 1
Car, 1
Bear, 1

Shuffling

Bear, 1
Bear, 1

Car, 1
Car, 1
Car, 1

Deer, 1
Deer, 1

Reducing

Bear, 2

Car, 3

Deer, 2

River, 2

Final result

Bear, 2

Car, 3

Deer, 2

River, 2
MapReduce scheduling

- One master, many workers
  - Input data split into $M$ map tasks (typically 64 MB)
  - Reduce phase partitioned into $R$ reduce tasks ($\text{hash}(k) \mod R$)
  - Tasks are assigned to workers dynamically

- Master assigns each map task to a free worker
  - Considers locality of data to worker when assigning a task
  - Worker reads task input (often from local disk)
  - Worker produces $R$ local files containing intermediate $k$/$v$ pairs

- Master assigns each reduce task to a free worker
  - Worker reads intermediate $k$/$v$ pairs from map workers
  - Worker sorts & applies user’s reduce operation to produce the output
MapReduce data locality

• Goal: To conserve network bandwidth

• In GFS, data files are divided into 64MB blocks and 3 copies of each are stored on different machines

• Master program schedules map() tasks based on the location of these replicas
  – Put map() tasks physically on the same machine as one of the input replicas (or, at least on the same rack / network switch)

• This way, thousands of machines can read input at local disk speed
  – Otherwise, rack switches would limit read rate
MapReduce fault tolerance

• On worker failure
  – Master detects failure via periodic **heartbeats**
  – Both completed and in-progress map tasks on that worker should be re-executed
    • Output stored on local disk
  – Only in-progress reduce tasks on that worker should be re-executed
    • Output stored in global file system
  – All reduce workers will be notified about any map re-executions

• On master failure
  – State is check-pointed to GFS: new master recovers & continues

• Robustness
  – Example: lost 1600 of 1800 machines once, but finished fine
DATA STREAM PROCESSING
Data stream processing

- Computer programming paradigm based on two concepts
  - Streams of data that flow through
  - Processing operators
Data stream processing

- Computer programming paradigm based on two concepts
  - Streams of data that flow through
  - Processing operators

- As we will see, different flavors
  - Graph based: the developers “draw” the graph of operators
  - Functional: the developers write functional transformations that define the operator graph
  - Declarative: the developers declaratively specify the desired results, and the runtime compiles the request into an operator graph
Data stream processing

• Why is it relevant?

• Why is it relevant in the context of (parallel and) distributed systems?

• Why is it associated to Big Data?
Relevant for reactive applications

- Financial Analysis
- Traffic Monitoring
- Fraud Detection
- System Monitoring

Velocity!
Reactive applications

- Continuous flows of information from the external environment

- Need to process them on the fly
  - To extract meaningful knowledge

- Typical requirements
  - Process large volumes of data as soon as the data is produced
    - ... to timely produce new results
      - High throughput
      - Low delay
Reactive applications

- Can we use existing technologies for batch processing (e.g., DBMSs)?
  - They are not designed to minimize latency
  - We need a whole new model!
Relevant for parallelism

• Stream processing is about applying the *same* operators on multiple data

• Easy to parallelize
  – At least, for stateless operators
    • Filter
      – Select only some data elements based on some characteristic
    • Map
      – Apply a given function to each and every data element
    • …
Relevant for Big Data

• Reading data from the disk is expensive
  – It is impossible to store all the data in memory (on a single machine)
  – Random access to data becomes almost unfeasible

• Stream processing model
  – Read the data from the disk
    • Producing a stream of data
  – Process the data through a network of multiple operators
    • Operators can be located on different nodes
    • The stream flows through the operators network only once
  – Store the results back on the disk
    • Or use them on-the-fly
(Some) background

- Active DBs
  - Early 90s

- Data Stream Management Systems (DSMSs)
  - 2000s

- Complex Event Processing (CEP)
  - 2000s

- Reactive Programming (RP)
  - Late 90s
  - Last few years
Active DB

- Traditional DB
  - Human-active database-passive
  - Processing is exclusively driven by queries

- Active DB
  - Event Condition Action (ECA) rules
  - Part of the reactive behavior moves from the application to the DB
  - Mostly DB extensions
    - View maintenance
    - Integrity checking
DSMS

- Data streams are (unbounded) sequences of data elements

- Often, the most recent data is more relevant as it describes the current state of a dynamic system
DBMS

- Persistent data
- One-time queries
- Read intensive
- Random access
- Access plan determined based on the actual data

DSMS

- Transient streams
- Continuous queries
- Update intensive (append)
- Sequential access (one pass)
- Unpredictable data characteristics and arrival patterns
DSMS (CQL)

Stream-to-Relation
(Windows)

Relation-to-Relation
(Relational Operators)

Relation-to-Stream
(New/All results)
DSMS (SQuAl)

- Stream-to-stream operators
  - E.g., filter, project, map, aggregate, join, …
- Embedded windows to make operators non-blocking
- Operators combined in a dataflow graph
Event-based systems

• Software architecture in which the components
  – Publish notifications of event occurrences
  – Subscribe to the events they are interested in

• Ideal for dynamic environments
  – Loosely coupled components
  – Implicit communication
    • Anonymous
    • Asynchronous
    • Multicast
Event-based systems

- In event-based systems the processing task consists in matching events against subscriptions.

- Different degrees of expressivity
  - Topic-based, content-based, ...

Producers
Publishers

Event-based system

Consumers
Subscribers
Complex Event Processing (CEP)

- CEP adds the ability to deploy rules that define *composite* events starting from *primitive* ones
  - E.g. if Temp(val > 10) and then Smoke within 5 min, trigger Fire
Stream/reactive programming

• Programming abstractions to simplify the design of reactive applications

• Focus on streams as unbounded collections of elements
  – (Functional) operators produce output streams from input streams
  – Similar to dataflow DSMSs

• Focus on programming language integration
Stream programming

filter \{\hspace{1em} \}\hspace{1em} map \{\hspace{1em} \to \hspace{1em} \}\hspace{1em}

37
Stream programming: Java 8

List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

myList
    .stream()
    .parallel()
    .filter(s -> s.startsWith("b"))
    .collect(...);
Reactive Languages and FRP

- **Signals**: What about expressing functional dependencies as constraints?

```scala
val a = 3
val b = 7
val c = a + b  // Statement
...
println(c) > 10
a = 4
println(c) > 10

val a = Var(3)
val b = Var(7)
val c = Signal{ a + b }  // Constraint
...
println(c) > 10
a = 4
println(c) > 11
```
Big data stream processors

• Several systems have been proposed to perform streaming computations on clusters
  – Similar to MapReduce / Hadoop …
  – … but focusing on streaming data

• Perhaps the most well known are
  – Apache Storm / Heron
    • Dataflow approach
    • Used within Twitter
  – Apache Spark Streaming, Apache Flink
    • Functional approach
Big data stream processors
Big data stream processors
Big data stream processors

- New concerns
  - Query deployment in large computational infrastructures
    - Operator placement
    - Operator migration
    - Auto-scaling: \textit{elasticity}
  - Fault tolerance
    - Later in this course you will see some algorithms to achieve fault tolerance in distributed systems
ARCHITECTURE
Architecture

- Two key design choices
  - Batched vs continuous computation
  - Pipelined vs scheduled

- Different trade-off between latency and throughput

- Different approaches to load balancing
Batched vs streaming

- In batched computation the stream is divided into batches of data

- Each operator processes a batch at a time
  - Waits until enough data elements are available to fill a batch
    - Might introduce some delay
  - Processing elements in batch is typically more efficient
    - E.g., optimized memory access, takes advantage of Single Instruction Multiple Data (SIMD) hardware support
      - GPU, vectorial instructions in CPU
    - Typically increases the throughput
Batched vs streaming

- In streaming computation each element gets processed as soon as it is available (and the operator has enough resources to handle it)
  - Lower delay
  - Possibly lower throughput

- Trade-off: micro-batch
  - Adopted by Apache Spark
  - Small batches guarantee high throughput with acceptable delays for many application scenarios
    - Typically in the second range
Pipelined vs scheduled

- In the case of continuous computations, the operator graph is almost always \textit{materialized}
  - At system start up, the operators are instantiated and allocated to physical nodes
  - Data moves from operator to operator
  - Operators form a processing \textit{pipeline}
Pipelined vs scheduled

- In the case of batched computations, operators can be scheduled (instantiated on a node) on demand, when a batch is available for processing.

- For instance, Apache Spark splits each stream into batches and instantiates new operator instances to process them.
Scheduled processing in Spark

Discretized stream processing

Records processed in batches with short tasks
Each batch is a RDD (partitioned dataset)
Load balancing

• In the case of pipelined execution, it is difficult to allocate the “right” amount of resources to the system
  – Some operators might be underused and still occupy hardware resources
  – Some operator might get overloaded

• Automatic scalability (or elasticity) is desirable in practice to avoid the waste of resources
Load balancing

- Scalability is achieved through operator replication
  - Multiple instances of the same operator
  - The number of instances can change at runtime

- Replication is possible
  - If the operator is stateless
    - Filter, map, …
  - Or, if instances operate on different data partitions
    - E.g., word count in MapReduce: reducers operate on different partitions of all possible words
Load balancing

• Load balancing is easier in the case of scheduled execution

• The stream processor can adopt techniques that are similar to scheduling algorithms in operating systems
Load balancing

Traditional systems

- unevenly partitioned streams
- bottleneck node

Static scheduling of continuous operators to nodes can cause bottlenecks

Spark Streaming

- more load on partition → longer tasks
- tasks scheduled based on available resources

Dynamic scheduling of tasks ensures even distribution of load
WINDOWS
Blocking operators

- Some operations cannot be performed over unbounded streams of data
  - So called blocking operators

- Example: negation
  - Let us assume a stream of integer
  - When we can safely say that id does not include any 0
  - The stream is unbounded
    - If we do not see any 0, we are blocked forever!
Blocking operators

- Other examples
  - Aggregates
    - Minimum, maximum
  - Constraints on aggregates
    - Average temperature > 0?

- Also some non-blocking operators can be impossible to implement in practice
  - Join of two streams
  - Need to store each element of one stream forever
    - It can possibly be joined with some future elements from the other stream!
Solution: windows

• Windows isolate the portion of the stream to take into account during the evaluation of a function

• Most common types
  – Count-based (physical): fixed number of elements
  – Time-based (logical): fixed amount of time

• The aforementioned types depend on two parameters
  – Size
  – Slide
Windows

Example:
- Count-based size 4 slide 2
- Function: avg
- Results: 2, 3, 2, 1
Windows

Sliding window: overlapping elements (slide < size)

Tumbling window: no overlapping elements (slide = size)
Windows

- Sliding windows often combined with a slide of 1
  - “Continuously” sliding, every time a new element arrives
  - Example: sliding windows in the Esper CEP system
TIME
Time in stream processing

• In most applications, windows are time-based
  – Data Stream Management Systems (DSMSs) assume that the most recent elements are the most relevant

• Several applications include temporal patterns
  – Complex Event Processing (CEP) in environmental monitoring
    • High temperature and presence of smoke within 5 minutes
  – CEP in financial monitoring
    • A stock index increases by 5% in one hour then decreases by at least 2% in the subsequent two hours, then …
The previous examples demonstrate the importance of time in stream processing.

But what is time?
  - Two aspects to take into account
    • Nature of time
    • Clock
Clock

- Which clock do we use to measure time?

- **Processing time**
  - The wall clock time of the machine that is performing the computation
    - What if the stream processor is running on multiple machines … ?
    - We will come back to this issue in a few slides

- **Event time**
  - Time is a meta-data associated to each and every data element
    - A “timestamp”
  - Who timestamps the elements?
    - The sources
      - What if they are running on multiple machines … ?
    - The stream processing system
Processing time

- Easy to manage

- “Out of the control” of the application
  - A source produces one element every 1ms
  - The stream processor transforms each input element into zero, one or more output elements
  - If the stream processor is not overloaded, the input and the output are reasonably synchronized …
  - Otherwise the stream processor can accumulate data
    - No synchronization between the input and the output
    - No way for a receiver to tell how “up-to-date” is the information it received
Processing time

- “Out of the control” of the application
  - No explicit synchronization in the case of multiple sources
  - The order of processing depends on the network delays between the sources and the processor
Processing time

- “Out of the control” of the application
  - No explicit synchronization in the case of distributed processing
  - The order of processing depends on the network delays between the processors
  - The order of processing depends on the speed and load of each processor
Event time

- If the processor timestamps the data, then the receiver gains some knowledge of “how old” are the results it receives.
Event time

- If the sources timestamp the data, then they have full control of the order in which data elements are processed
  - However, the processor might receive elements out-of-order (timestamps order)
  - It needs to buffer data elements until it is “sure” that no other previous element will arrive
    - Might introduce undesired delays
Event time

- Order is preserved even in the presence of distributed processing
Nature of time

• In the case of event time
  – Timestamps can assume different forms

• We will not go into the details, but we will give you an intuition
  – Example:
    • Data elements encode “states”
    • State might last for a time interval
    • It is natural to encode time as intervals …
Interval time model

- Which is the immediate successor of A?
  - Choose according to end time only: B
    - But it started before A!
  - Exclude B: C, D
    - Both of them?
    - Which of them?
  - No other event strictly between A and its successor: C, D, E
    - Seems a natural definition
    - Unfortunately we lose associativity!
      - $X \rightarrow (Y \rightarrow Z) \neq (X \rightarrow Y) \rightarrow Z$
    - May prevent rule rewriting for processing optimizations
QUESTIONS?