Distributed Systems
Processing Big Data

Tenshi Hara
tenshi.hara@tu-dresden.de
https://www.rn.inf.tu-dresden.de/hara
Outline

- What is Big Data?
- Batch Processing
  - Map/Reduce
  - Apache Hadoop
- Stream processing
  - Apache Apex
Example System: E-Commerce

Predictive Analytics
- Price optimisation
- Personalised recommendations
  - ‘Inspired by your wish list’
  - ‘Recommendations for you’
  - ‘Related to items you have viewed’
  - ‘Customers who bought this item also bought’
- System Health Checks
- Log data analysis
- Anomaly detection
Application Domains

- Financial applications
  - trend analysis: continuous analysis of stocks to identify trends
  - fraud detection: observation of continuous streams of credit card transactions

- Security
  - intrusion detection: analyse network traffic in real-time, generating alerts when something unexpected happens

- Manufacturing
  - inventory management: continuous analysis of RFID readings to track valid paths of shipments and to capture irregularities
  - manufacturing control systems often require anomaly detection

- Environment Sensing
  - disaster prediction: process data coming from sensors deployed in the field (earthquakes)
What is Big Data?

- **Gartner IT Glossary:**
  
  *Big data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation.*


- **3 Dimensions**
  
  - **High Volume**
    - large data-sets
  
  - **High Velocity**
    - grow rate of data sets
    - demand for processing speed
  
  - **High Variety**
    - heterogeneous data sources and representation
How big is Big Data?

- **Large data-sets (volume)**
  - Facebook: 1.79 billion monthly active Facebook users (Q3/2016)
  - Google self-driving car: 2 PB data per car per year
  - Amazon: product catalogue of ~1.5 billion products

- **Change rate and demand for processing speed (velocity)**
  - Twitter: 6000 tweets per second, 500 million per day
  - Facebook: 4.5 billion likes generated daily (5/2013)
  - Facebook: Five new profiles created every second
  - Google self-driving car: nearly 1 GB/s

- **Data sources and representations (variety)**
  - Log files, sensor data, social network data, user click streams
Challenges

- Storage
- Data Access
- Performance
- Timeliness
- Scalability
- Fault-tolerance

⇒ Computation-centric technologies are not appropriate!
- Programming model for processing big data sets
  - inspired by MapReduce in functional programming like LISP, but not equivalent
  - easy parallelisation of large computations
  - re-execution as primary mechanism for fault tolerance
- Two key operations, one extra
  - **map** \((k1,v1) \rightarrow \text{list}(k2,v2)\)
    applying a map operation to each logical ‘record’ in our input in order to compute a set of intermediate key/value-pairs
  - **reduce** \((k2,\text{list}(v2)) \rightarrow \text{list}(v2)\)
    applying a reduce operation to all the values that share the same key, in order to combine the derived data appropriately
  - **shuffle** \(\text{list}(k2,v2) \rightarrow k2,\text{list}(v2)\)
    output of multiple map executions to reduce input
Map/Reduce – Word Count Example

map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

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:
map(docID, "Apple Orange Mango") emits
{(Apple,1), (Orange,1), (Mango,1)}

Example:
reduce(Apple, {1,1,1,1}) emits
{Apple,4}
Map/Reduce – Word Count Distributed

Automatic partitioning of input data into set of M splits

Each input split can be processed by map operator on different machine

Sort and Shuffle

Reduce

Final Output

Master manages workers distributed in cluster
Apache Hadoop

- Top Level Project in the Apache Software Foundation
- Open Source implementation of MapReduce
- Large Ecosystem around core
  - Hive – data warehouse infrastructure; SQL-like
  - Pig – high-level scripting language and platform
  - Impala - SQL query engine for data stored in a computer cluster
  - Solr – distributed searching and indexing
  - Spark/Apex – Stream processing
  - Flume and Sqoop - data import

Unstructured/Semi-structured Data

Structured Data
Hadoop Distributed File System (HDFS)\(^1\)

- designed to run on large clusters of commodity hardware
- traditional hierarchical file organisation with directories and files
- large data sets (typical file size is Gigabytes or Terabytes)
  - block-based (64MB or 128MB as default size)
  - optimised for throughput rather than low latency access
- simple coherency model (write-once-read-many)
- fault-tolerance (blocks are replicated)
- scales to hundreds of nodes and millions of files

\(^1\) http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
Hadoop HDFS – Architecture

Management of namespace and file access
- MetaData and File System Operations
  - create, open, close, rename, etc.
  - Mapping of files to blocks at DataNodes

Block Operations
- create, read, write, delete
- replication of blocks

Hadoop HDFS – Management

- HDFS communication is based on RPC on top of TCP/IP → Client and DataNode protocols
- NameNode never initiates RPC requests, but only responds to requests issued by DataNodes or clients
- User data never flows through NameNode

Files are split into blocks of equal size

Block size and replication factor are configurable per file

NameNode makes all decisions regarding replication of blocks

each block resides on a different DataNode

Replica placement

for the common case, when the replication factor is three, HDFS’s placement policy is to put one replica on one node in the local rack, another on a different node in the local rack, and the last on a different node in a different rack

Replica selection

closest to the reader (node, rack, data center)

Interfaces for applications to move computation ‘close’ to DataNodes

HDFS Fault-tolerance

- Failure types: nodes, network errors or partitions, corrupted data
- Replication as primary mechanism: staging and replication pipelining
  - Client writes data to temporary local file
  - if local file size reaches HDFS block size, creation request is sent to NameNode
  - NameNode creates file meta data and allocates data blocks on DataNodes
  - NameNode responds with list of DataNodes for n replicas
  - Client flushes data block to first DataNode in small portions (4KB)
  - DataNode writes data locally and forwards portio to next DataNode (replication pipelining)

HDFS Fault-tolerance

- Failure types: nodes, network errors or partitions, corrupted data
- Re-replication
  - DataNode failures detected by NameNode due to missing heatbeats
  - corrupted data detected based on checksum
  - replication factor of blocks may fall below specified value
  - NameNode initiates re-replication

Yet Another Resource Negotiator (YARN)

- generic resource-management and distributed application framework for Hadoop
- enables support for multiple programming models (in addition to MapReduce)

**Resource Container**

- abstract notion for resource requests and allocations
- incorporates resource elements such as memory, CPU, disk, network, etc.

- per-machine **NodeManagers** manage local resource containers
  - launching the applications’ containers
  - monitoring their resource usage
  - reporting resource usage to ResourceManager
Yet Another Resource Negotiator (YARN)

- Central **ResourceManager** manages resources on cluster of nodes
  - Knowledge about global resource usage from periodic status updates of NodeManagers
  - Scheduler allocates resource containers for applications

- Per-Application **ApplicationMaster**
  - Negotiating appropriate resource containers from the Scheduler
  - Tracking their status and monitoring for progress
  - Runs itself as a container
YARN Architecture

1. Submit application with request for launching ApplicationMaster
2. ResourceManager allocates appropriate container and launches ApplicationMaster
3. ApplicationMaster registers with ResourceManager, allows Client to resolve ApplicationMaster for direct communication

1 http://de.hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/
4. ApplicationMaster negotiates appropriate resource containers via the resource-request protocol

5. ApplicationMaster launches container by providing launch specification to NodeManager, connects launched container to ApplicationMaster for direct communication

1 http://de.hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/
6. Application code executing within the container provides necessary information (progress, status, etc.) to its ApplicationMaster via an application-specific protocol.

7. During execution, client communicates directly with its ApplicationMaster to get status, progress updates, etc. via an application-specific protocol.

---

1 http://de.hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/
8. ApplicationMaster pushes updates into ResourceManager and deregisters after execution or termination by client.
Apache Hadoop Vendors

- **Cloudera CDH**  ‘enterprise ready’ Open-Source Distribution (also: related products)
- **Hortonworks** open source solution and a major contributor to the Apache Hadoop project
- **MapR Distribution** Distribution that allows Hadoop to be accessed via Network File System (NFS)
- **IBM Open Platform** Free to use open source platform integrating Apache projects for Big Data
- **Amazon Elastic MapReduce** Pay-as-you-use offer for management and processing of big data sets

Unbounded data:
- ‘a type of ever-growing, **essentially infinite** data set’
- time-dependent
- examples: activity tracking, stock market
- finite ‘batch’ data sets are called ‘bounded’

Unbounded data processing
- ongoing mode of data processing applied to unbounded data

Streaming
- execution engine designed for unbounded data sets
### Domains of time

1. **Event time**
   - Time at which events actually occur
   - Time stamp assigned to event

2. **Processing time**
   - Time at which events are observed/processed in the system
   - Later than time stamp

3. **In practice:**
   - Skew between event time and processing time
   - Non-zero and (highly) variable due to latency introduced by input sources, data transmission, shared resources, etc.
   - Data cannot be analyzed within the context of when they are observed → notion of windowing

---

Event-time vs. Processing-time

- Window → Grouping into finite pieces along temporal boundaries
- Processing time windowing → Grouping of data based on observation
- Event time windowing
  - Grouping of data based on occurrence
  - Issue: How can you determine when all the system has observed all data for a given event time $X$?
    - some notion of completeness used
    - upper limit for occurrence of delayed data required
- Event time has to be used to ensure correctness (if timing between events matters)
What is Apex?

- **Open Source Real-time streaming application platform**
- Designed to process massive amounts of real-time events
- Consists of two components
  - **Apex Core** – Engine which facilitates real-time processing
  - **Apex Malhar** – Library of out-of-the-box operators
- Complete implementation in Java
- End-User-friendly – User can focus on business logic
  - Platform handles all details of streaming execution
  - Removes need to maintain separate clusters for applications
Apex & Hadoop

- Apex streaming applications
  - run in Hadoop cluster as **YARN-native applications (Streaming Application Manager (STRAM))**
  - support natively all of Hadoop’s distributed operating system capabilities; e.g., resource management, fault-tolerance, scalability
- enables use of existing Hadoop infrastructure for streaming applications
- designed to scale with Hadoop cluster and big data applications
Apex – Architecture

- Machine nodes (physical or virtual)
- Hadoop (YARN)
- HDFS
- Rest API
- Custom Operators
- Apex Malhar Operators
- Apache Apex Core (Streaming Engine)
- Streaming Application
- Streaming Application

Distributed Systems – Processing Big Data
Application are represented as Directed Acyclic Graph (DAG)

- **Operators** $\rightarrow$ computation units/business logic (nodes in DAG)
- **Streams** $\rightarrow$ sequence of data tuples that connect operators (edges in DAG)
- **Tuples** $\rightarrow$ smallest atomic data elements that flow over a stream
- **Ports** $\rightarrow$ part of Operator; ends of a Stream
Streaming Application Model

- completely asynchronous real-time computation model
- Goal: unblocked as possible, minimal overhead
- all *computations done in memory* on arrival of data
  - Option to save output to disk (HDFS)
- Streams consists of tuples
  - Tuples enter a stream through input port and leave through output port of operator
- Tuples are *always in-order*
Streaming Windows

- fundamental concept of streaming platform
- breaking up stream into equal finite time slices → **streaming windows**
- each window:
  - has specified window size (default 500ms)
  - unique window ID
  - contains ordered set of tuples in that time slice
  - preceded/terminated by begin_window/end_window event
- Why window concept? → **Avoid per tuple bookkeeping costs.**
  - Platform computations at tuple level
  - Bookkeeping at window level
Streaming Windows

- Input operator inserts control tuples which mark window boundaries
- Different operators can be processing different windows

Application Windows

- Build in support
- Group of consecutive streaming windows (multiple of streaming window)
- Can be set differently per Operator
- Used for data aggregation (e.g. sum, average, max/min,...)
Structure of Apex Application:

- **Application class**: instantiates Operators & Streams and connects them to DAG in `populateDAG(...)` method

```java
public class Application implements StreamingApplication {
    populateDAG(DAG dag, Configuration conf) {
        dag.addOperator(args) // Add Operators to dag
        dag.addStream(args) // Add Streams between operators
        // Additional optional configurations + Hints to YARN
    }
}
```

- **Operator Class** for each Operator
  - instantiate needed Input-Ports & Output-Ports
  - override process() method of input-port, do processing and emit() tuple on output-port
  - other important methods: setup(), beginWindow(), endWindow(), emitTuples()
DAG of **WordCount** Streaming Application

- reads file from HDFS, emits lines & words, counts them and passes result to console output once end of file is reached

EOF...End of File
**InputLineReader** (Input Adapter Operator)
- Open file, read lines
- Output-ports: 1. emit each line to WordReader, 2. emit EOF Control tuple for FileWordReader-Operator

**WordReader**
- Input: lines, extract words from lines, Output: emit each word

**UniqueCounter**
- Out-of-the-box Malhar Operator
- Counts number of times tuple exists in window
- Output: Map from tuples to counts emitted at end of each application window
**FileWordReader**
- Input-ports: 1. Maps of Words with count, 2. EOF Control tuple
- Instantiate Map of <words, counts>-pairs
- Add incoming word-frequencies to internal Map
- If EOF tuple arrives -> Output: final aggregated Map

**ConsoleOutputOperator**
- Out-of-the-box Malhar Operator
- simply writes incoming tuples to the console
Streaming scenarios often include high fluctuation of incoming load

**Scales linearly with Hadoop**
- simply add more commodity nodes to Hadoop → physical setup
- Hadoop native platform takes care of resource management (without any downtime)

Scaling through **partitioning of Operators**
- logical setup
- partitioning of operator at design time → **Static Partitioning**
- Operators can be scaled up/down at runtime based on load or SLA → **Dynamic Partitioning** (auto scaling)

Distribution of tuples?
- **Data unaware**: using load balancing algorithm (Round Robin, ...)
- **Data aware/sticky key**: uses hash key of tuple to send same kind of tuples to same physical Operator
Logical DAG with three operators

- Logical DAG will be transformed into a physical DAG
- Unifier needed to aggregate partitioned Operators
- Unifiers can become bottlenecks -> cascading unifiers possible

Physical DAG with Unifier and two partitions of Operator 1
Parallel Partitioning

- when all downstream operators use same partitioning scheme → effective optimisation called *parallel partition*
- all downstream operators are also partitioned → create computation flow per partition
- completely skip the insertion of intermediate Unifier operators
- only one Unifier at the end, where tuple volume might be much lower

Physical DAG with Operator 1&2 using parallel partitioning
Yahoo! Finance Application – Goal

```
$ curl 'https://download.finance.yahoo.com/d/quotes.csv?s=IBM,GOOG,AAPL,YHOO&f=sl1vt1'

"IBM",203.966,1513041,"1:43pm"
"GOOG",762.68,1879741,"1:43pm"
"AAPL",444.3385,11738366,"1:43pm"
"YHOO",19.3681,14707163,"1:43pm"
```

<table>
<thead>
<tr>
<th>stock symbol</th>
<th>price</th>
<th>incremental volume</th>
<th>last trade time</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>203.966</td>
<td>1513041</td>
<td>1:43pm</td>
</tr>
<tr>
<td>GOOG</td>
<td>762.68</td>
<td>1879741</td>
<td>1:43pm</td>
</tr>
<tr>
<td>AAPL</td>
<td>444.3385</td>
<td>11738366</td>
<td>1:43pm</td>
</tr>
<tr>
<td>YHOO</td>
<td>19.3681</td>
<td>14707163</td>
<td>1:43pm</td>
</tr>
</tbody>
</table>

Quote
- last trade price
- last trade time
- total volume for the day

Per-minute Chart Data
- highest trade price
- lowest trade price
- volume during that minute

Simple Moving Average (SMA)
- trade price over 5 minutes

1[https://apex.apache.org/docs/apex/application_development/#test-application-yahoo-finance-quotes]
Stock Tick Input

Operator Config
5 min = 300s; streaming window size = 1s
⇒ Application Window Count = 300
(300 streaming windows per Application Window)
StockTickerInput reads live data from the Yahoo! Finance API once per interval.

- emits the price, the incremental volume, and the last trade time of each stock symbol (e.g., AAPL, IBM)
- utilises the Yahoo! Finance CSV web service interface

The operator has three output ports that emit:

- key/value-pair <stock symbol, price>
- key/value-pair <stock symbol, incremental volume>
- key/value-pair <stock symbol, last trade time>

from: https://apex.apache.org/docs/apex/application_development/#test-application-yahoo-finance-quotes
public class StockTickerInput implements InputOperator {

    public DefaultOutputPort<KeyValPair<String, Double>> priceOutputPort = new DefaultOutputPort<KeyValPair<String, Double>>();

    public DefaultOutputPort<KeyValPair<String, Long>> volumeOutputPort = new DefaultOutputPort<KeyValPair<String, Long>>();

    public DefaultOutputPort<KeyValPair<String, String>> timeOutputPort = new DefaultOutputPort<KeyValPair<String, String>>();

    private HashMap<String, Long> lastVolume = new HashMap<>();

    ...
}
public void emitTuples() { /* simplified pseudo-code! */
    // example row: "IBM",203.966,1513041,"1:43pm"
    String url = "http://finance.yahoo.com/d/quotes.csv?s=AAPL,IBM"
    Response res = new HttpRequest(url).getBody();
    for (Row row: res) {
        String symbol = row.get(0);
        double currentPrice = row.get(1);
        long currentVolume = row.get(2);
        String timestamp = row.get(3);
        long vol = currentVolume;
        if (lastVolume.containsKey(symbol)) {
            vol -= lastVolume.get(symbol);
        }
        priceOutputPort.emit(new KVPair(symbol, currentPrice));
        volumeOutputPort.emit(new KVPair (symbol, vol));
        timeOutputPort.emit(new KVPair(symbol, timestamp));
        lastVolume.put(symbol, currentVolume);
    }
}
Challenges and Issues

- correctly handle operator state that is larger than memory
  - e.g., Apache Apex Dedup Operator¹
  - uses ManagedStorage backed by HDFS
- keep fault tolerance in mind:
  - e.g., reading from file ‘line by line’
    → line-offset needs to be restored after operator failure
  - e.g., Apache Apex FsInputOperator²
- consider event-time vs. processing-time semantics³
- derive useful partitioning strategies⁴
  - static partitioning vs. dynamic partitioning
  - ‘out of the box partitioning’ vs. ‘custom partitioning’

¹ http://de.slideshare.net/DataTorrent/deep-dive-of-deduplication-using-apache-apex-and-rts/11
² https://apex.apache.org/docs/malhar/operators/fsInputOperator/
⁴ https://apex.apache.org/docs/apex/application_development/#dynamic-partitioning
Information Flow Processing (IFP)¹

**Data Stream Processing (DSP)**

- *data stream processing model describes the IFP problem as processing streams of data coming from different sources to produce new data streams as an output*
- evolution of traditional data processing,
- **roots in database management systems**
- dealing with transient data that is continuously updated
- *Execute standing queries that run continuously*

Complex Event Processing (CEP)

- views flowing information items as notifications of events happening in the external world, which have to be filtered and combined to understand what is happening in terms of higher-level events.
- detecting occurrences of particular patterns
- roots in publish/subscribe domain
- Notification to interested parties if pattern detected
- increasing the expressive power of the subscription language to consider complex event patterns

---

• Big data requires specific infrastructures for high-performance, scalable and fault-tolerant processing

• Batch Processing based on MapReduce
  • Apache Hadoop widely adopted infrastructure to exploit large clusters of commodity hardware for processing
  • Growing ecosystem with solutions at various abstraction levels

• Stream Processing addresses demand for timely processing and infinite data sets; Challenges are¹
  • correctness (to parity with batch processing)
  • Reasoning about time (to get beyond batch processing)

• Lambda-Architecture to combine batch and streaming
  • Run streaming system (low-latency, inaccuracy) and batch system (delay, correctness) in parallel, both performing the same calculation

• Future stream processing solutions will provide strict superset of batch systems (Kappa architecture)

References

- http://hadoop.apache.org/docs/current/
- https://www.datatorrent.com/blog/blog-introduction-to-checkpoint/
- https://apex.apache.org/docs/apex/application_development/#test-application-yahoo-finance-quotes