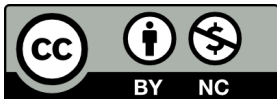


Business 4720 - Class 20

Analytics at Industrial Scale – Big Data Analytics

Joerg Evermann

Faculty of Business Administration
Memorial University of Newfoundland
`jevermann@mun.ca`



Unless otherwise indicated, the copyright in this material is owned by Joerg Evermann. This material is licensed to you under the [Creative Commons by-attribution non-commercial license \(CC BY-NC 4.0\)](#)

What You Will Learn:

- ▶ Distributed data storage
 - ▶ Hadoop HDFS
- ▶ Distributed computation
 - ▶ Hadoop Map-Reduce
 - ▶ Spark
 - ▶ Dataframe operations
 - ▶ Spark SQL
 - ▶ Spark MLlib

Further Reading

Hrishikesh V. Karmabelkar (2018) *Apache Hadoop 3 Quick Start Guide*. Packt Publishing. Birmingham, UK.

Tom White (2012) *Hadoop – The Definitive Guide*. 3rd edition. O'Reilly Media. Sebastopol, California, US.

Bill Chambers and Matei Zaharia (2018) *Spark – The Definitive Guide*. O'Reilly Media. Sebastopol, California, US.

Jules Damji et al. (2020) *Learning Spark – Lightning-Fast Data Analytics*. 2nd edition. O'Reilly Media. Sebastopol, California, US.

Characterized by any one or more of:

- ▶ Large volume
- ▶ Large "velocity" (volume per time)
- ▶ Large variety (of data types and sources)

Big Data Example – CERN

”Conseil Europeenne pour la Recherche Nucleaire”

<https://www.home.cern/science/computing/data-centre>¹

Servers	≈ 12000	
CPU Cores	≈ 330000	
Disks	≈ 220000	
Total Disk Space	≈ 950000	TB
DB Transactions per second	≈ 20000	
File Transfer Throughput	≈ 500000	Gb/s

¹ Accessed Feb 23, 2024

Hadoop

- ▶ Initial release 2006
- ▶ Maintained by the Apache Foundation
- ▶ Inspired by Google File System (GFS) (2003) and Google MapReduce (2004) for large data management
- ▶ Early use cases by Yahoo (2009) and Facebook (2012) drove adoption
- ▶ Distributes data storage and computation across a cluster of computers
- ▶ *Data locality* means moving computation to data, not data to computation

- ▶ **Reliability:** Hardware and software failure tolerance through replication and automatic recovery
- ▶ **Scalability:** Dynamically adding and removing storage nodes and cluster re-balancing (more than 10,000 nodes in Hadoop 3)
- ▶ **Cost effective:** Open source, runs on commodity hardware, can use heterogeneous nodes
- ▶ **Cloud support:** Vendors offering turn-key Hadoop systems

Main Hadoop Components

- ▶ **HDFS:** Hadoop Distributed File System
- ▶ **MapReduce:** Software framework for processing large data volumes
- ▶ **YARN:** Yet Another Resource Negotiator (cluster and compute job manager)

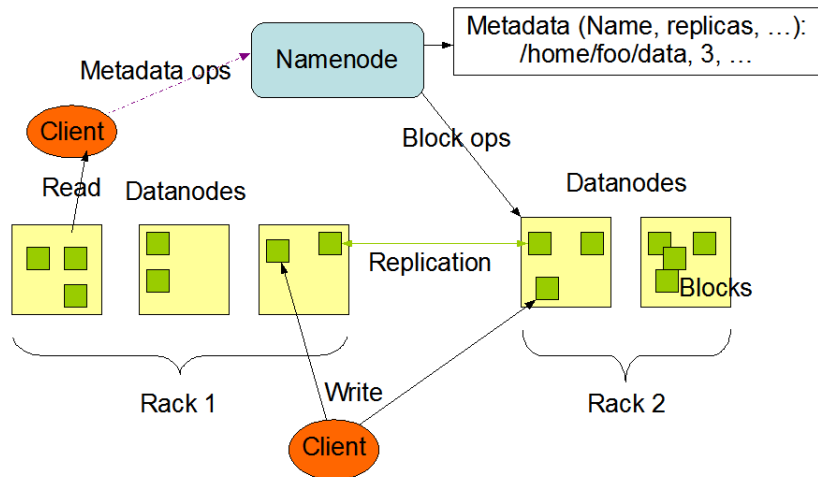


https://commons.wikimedia.org/wiki/File:Apache_Hadoop.png

- ▶ **Streaming data access:** Data is written and read linearly, processed one item at a time
- ▶ **Large datasets:** Multiple gigabytes or terabytes, hundreds of computers per clusters, millions of files per node
- ▶ **Write once:** Files once written are only read or appended to
- ▶ **Moving computation is cheaper than moving data:** Move compute applications to the server that stores the data

HDFS Architecture

HDFS Architecture



Source: Apache Foundation (<https://hadoop.apache.org/docs/>)

NameNode

- ▶ One NameNode per cluster (plus secondary/backup)
- ▶ Manages file namespace (sub-directories, file names, etc.)
- ▶ Regulates access to files
- ▶ Provides file operations such as opening, closing, renaming, etc.

DataNode

- ▶ Files are split into blocks stored on DataNodes
- ▶ DataNodes handle read and write requests of clients
- ▶ DataNodes perform block operations for file operations by NameNode

Working with HDFS

Use the `hdfs dfs` command to interact with the distributed file system. The commands are similar to the regular Linux commands to interact with files.

<code>hdfs dfs -cat</code>	Print a file to standard output
<code>hdfs dfs -cp</code>	Copy a file or directory
<code>hdfs dfs -df</code>	Display free space
<code>hdfs dfs -du</code>	Display disk usage
<code>hdfs dfs -get</code>	Copy files to the local file system
<code>hdfs dfs -head</code>	Print the first kilobyte of a file
<code>hdfs dfs -ls</code>	List files and directories

Continued . . .

<code>hdfs dfs -mkdir</code>	Make a directory
<code>hdfs dfs -mv</code>	Move a file or directory
<code>hdfs dfs -put</code>	Copy files from the local file system
<code>hdfs dfs -rm</code>	Remove files or directories
<code>hdfs dfs -rmdir</code>	Removes a directory
<code>hdfs dfs -tail</code>	Print the last kilobyte of a file
<code>hdfs dfs -concat</code>	Concatenate existing files into a target file

Working with HDFS – Examples

Start the Hadoop cluster NameNode, DataNode, and YARN service:

```
sudo systemctl start hadoop.service
```

Download an event log file:

```
wget https://evermann.ca/busi4720/eventlog.short.log
```

Put the event log on to the Hadoop Distributed File System:

```
hdfs dfs -put eventlog.short.log
```

Display the start and end of it:

```
hdfs dfs -head eventlog.short.log  
hdfs dfs -tail eventlog.short.log
```

Working with HDFS – Examples [cont'd]

Show disk usage and disk free space:

```
hdfs dfs -du  
hdfs dfs -df
```

Copy the event log:

```
hdfs dfs -cp eventlog.short.log eventlog.copy.log
```

List all files:

```
hdfs dfs -ls
```

Web Interface

- ▶ NameNode overview at <http://localhost:9870>
- ▶ HDFS explorer at <http://localhost:9870/explorer.html#/>

Accessing HDFS

- 1 View the list of files on the HDFS file system.
- 2 Create a new directory in the HDFS named "testdir".
- 3 Verify that 'testdir' has been created by listing all files.

Exercise 2: Manipulating Files in HDFS

- 1 Use a text editor to create a text file called "example.txt" on your local file system and write "Hello HDFS" into it.
- 2 Copy "example.txt" from your local file system to the directory "testdir" in the HDFS.
- 3 Read the contents of the file from the HDFS.
- 4 Delete "example.txt" from the HDFS.

Exercise 3: Understanding HDFS Block Size

- 1 Create a large text file (e.g., larger than 128MB, the default block size in many HDFS installations) named "largefile.txt" on your local file system.
- 2 Copy "largefile.txt" to HDFS:
- 3 Use the `hdfs fsck` command with the `-files` `-blocks` options to check how HDFS has stored the file in terms of blocks.
- 4 Observe and note the number of blocks the file is split into and their locations.

What is it?

- ▶ Programming model for parallel processing of data
- ▶ Move computation to data nodes

Strengths

- ▶ Massively parallelizable
- ▶ Conceptually simple: Only 2 types of functions

Drawbacks

- ▶ Disk limited: Intermediate results are written to disk
- ▶ Stateless functions only
- ▶ Non-iterative, acyclic dataflow programs only

MapReduce – Basic Steps

1 Map

- ▶ Reads key–value pairs of input²
- ▶ For each input key and value, outputs a list of key–value pairs

Map : $(key1, value1) \rightarrow list(key2, value2)$

2 Shuffle

- ▶ Distributes data based on keys produced by *map*
- ▶ All values for the same key are sent to the same reducer

3 Reduce

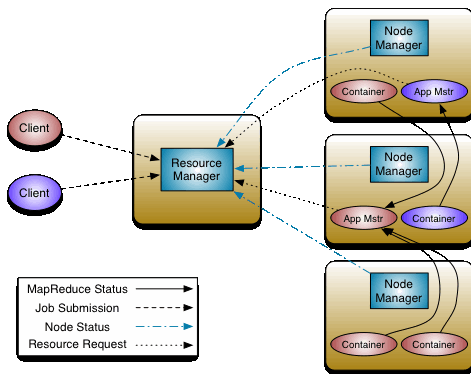
- ▶ Processes all values for a given key
- ▶ For each input key and its values, outputs a list of key–value pairs

Reduce : $(key2, list(value2)) \rightarrow list(key3, value3)$

²By default, for text input, each line is a key–value pair, separated by the first tab character

MapReduce on Hadoop – YARN cluster manager

- ▶ Submit an application (set of jobs) to **Resource Manager**
- ▶ **Application Master** tracks status of jobs and tasks
- ▶ Tasks distributed to nodes
- ▶ **Node Managers** manage local resources



https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn_architecture.gif

MapReduce on Hadoop

- ▶ Specify input directory
 - ▶ Data in multiple data files
 - ▶ Data files are stored in blocks distributed across cluster
- ▶ One *map job* is executed for each input block
 - ▶ *Map* jobs are executed on node where input block is located
 - ▶ Necessary program files are sent to each node if necessary
 - ▶ Map output is moved to nodes for *reduce* job ("shuffle")
- ▶ Execute a *reduce job* on every node for maximum parallelization

MapReduce Example – Word Count

- ▶ *Hadoop MapReduce* is programmed in Java
- ▶ *Hadoop Streaming* allows mappers and reducers as executable programs (e.g. in Python)

The Mapper:

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '{}\t{}'.format(word, 1)
```

The Reducer:

```
#!/usr/bin/env python
import sys

word_counts = dict()

for line in sys.stdin:
    word, count = line.split('\t', 1)
    count = int(count)

    if word not in word_counts:
        word_counts[word] = count
    else:
        word_counts[word] = word_counts[word] + count

for word, count in word_counts.items():
    print('{}\t{}'.format(word, count))
```


MapReduce Example – Word Count [cont'd]

Try it locally:

```
wget https://evermann.ca/busi4720/map.py
wget https://evermann.ca/busi4720/reduce.py
wget https://evermann.ca/busi4720/hamlet.txt
chmod +x *.py
```

Run the mapper and view its output:

```
cat hamlet.txt | ./map.py > map.out
less map.out
```

Run the reducer and view its output:

```
cat map.out | ./reduce.py > reduce.out
sort -k2 -rn reduce.out | less
```

MapReduce Example – Word Count [cont'd]

Put the text file on the HDFS:

```
hdfs dfs -mkdir hamlet
hdfs dfs -put hamlet.txt hamlet
hdfs dfs -ls hamlet
```

Run the MapReduce job on the Hadoop cluster:

```
mapred streaming \  
-input hamlet -output hamlet.out \  
-mapper map.py -reducer reduce.py \  
-file map.py -file reduce.py
```

Examine the results:

```
hdfs dfs -ls hamlet.out
hdfs dfs -get hamlet.out/part-*
cat part-* | sort -k2 -rn | less
```

Hands-On Exercise – Map Reduce

Modify the `map.py` and `reduce.py` python code on the previous slide to count **word length frequencies**.

Tips:

- ▶ The map function should output the length of each word as key
- ▶ Use the `len(.)` function to determine the length of a word
- ▶ The reduce function will use numerical keys, convert the keys it reads appropriately.

MapReduce Use Case – Scalable Process Discovery



1 α -Miner

- ▶ 2 MapReduce phases

2 Flexible Heuristic Miner

- ▶ 5 MapReduce phases
- ▶ Random process, 47 activity types
- ▶ 5 million traces, 80GB event logs
- ▶ Three cluster sizes:
 - 1 Single node cluster, 2 CPUs
 - 2 10-Node cluster, 2 CPUs each
 - 3 10-Node cluster, 10 CPUs each

Source: Evermann, J. (2016) Scalable Process Discovery using Map-Reduce. *IEEE TSC*, 9 (3), 469-481.

<https://doi.org/10.1109/TSC.2014.2367525>

Example – Flexible Heuristic Miner MapReduce Pipeline

- ▶ Keys and values can be complex
- ▶ Define a comparison function to shuffle

map1: $(Int, Text) \rightarrow set(CaseID, (Event, TimeStamp))$

shuffle1: $set(caseID, (Event, TimeStamp)) \rightarrow (CaseID, set(Event, TimeStamp))$

reduce1: $(CaseID, set(Event, TimeStamp)) \rightarrow set((Event, Event), (Int, Bool, Int))$

combine2: $set((Event, Event), (Int, Bool, Int)) \rightarrow set((Event, Event, (Int, Bool, Int)))$

reduce2: $((Event, Event), set(Int, Bool, Int)) \rightarrow set(c, (Event, Event, Int, Float))$

reduce3: $set(c, (Event, Event), set(Int, Float)) \rightarrow set(c, (Event, Event))$

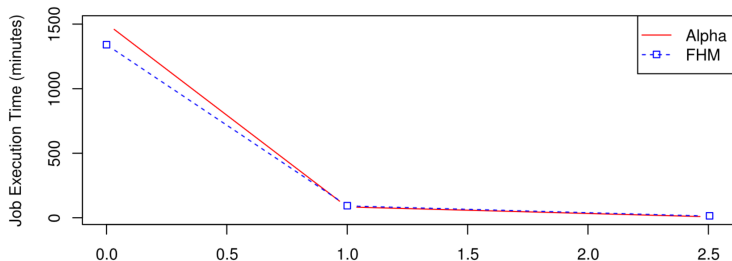
map4: $(Int, Text) \rightarrow set(CaseID, (Event, TimeStamp))$

shuffle4: $set(CaseID, (Event, TimeStamp)) \rightarrow (CaseID, set(Event, TimeStamp))$

reduce4: $(CaseID, set(Event, TimeStamp)) \rightarrow set((Event, set(Event), Bool), Int)$

reduce5: $((Event, set(Event), Bool), set(Int)) \rightarrow ((Event, set(Event), Bool), Int)$

MapReduce Use Case – Results



Source: Evermann, J. (2016) Scalable Process Discovery using Map-Reduce. *IEEE TSC*, 9 (3), 469-481. <https://doi.org/10.1109/TSC.2014.2367525>

α Algorithm

Single node	25:00 hours
Medium cluster	1:24 hours
Large cluster	0:08 hours

FHM

Single node	22:21 hours
Medium cluster	2:01 hours
Large cluster	0:17 hours



Apache Pig

[https://en.wikipedia.org/
wiki/File:
Apache_Pig_Logo.svg](https://en.wikipedia.org/wiki/File:Apache_Pig_Logo.svg)

- ▶ High-level programming
- ▶ Pig Latin language
- ▶ Pig Latin programs run as MapReduce jobs on Hadoop
- ▶ Procedural, not declarative (SQL)

[https://pig.apache.org/docs/
latest/basic.html](https://pig.apache.org/docs/latest/basic.html)

Apache Pig Latin Example – Word Count

```
input_lines = LOAD 'hamlet.txt' AS (line:chararray);
-- Extract words from each line and put them into
-- a pig bag datatype, then flatten the bag to get
-- one word on each row
words = FOREACH input_lines \
    GENERATE FLATTEN(TOKENIZE(line)) AS word;
-- create a group for each word
word_groups = GROUP words BY word;
-- count the entries in each group
word_count = FOREACH word_groups \
    GENERATE COUNT(words) AS count, group AS word;
-- order the records by count
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO 'hamlet.out';
```

Source: https://en.wikipedia.org/wiki/Apache_Pig

Apache Pig Latin – Relational Operators

LOAD	STORE	DUMP
FILTER	DISTINCT	FOREACH ... GENERATE
SAMPLE	JOIN	GROUP
CROSS	ORDER	LIMIT
UNION	SPLIT	



https://commons.wikimedia.org/wiki/File:Apache_Hive_logo.svg

- ▶ Data warehouse
- ▶ HiveQL similar to SQL
- ▶ Translate HiveQL to run as MapReduce jobs on Hadoop

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual>

HiveQL Example – Word Count

```
DROP TABLE IF EXISTS docs;
CREATE TABLE docs (line STRING);
LOAD DATA INPATH 'hamlet.txt'
  OVERWRITE INTO TABLE docs;

CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
  (SELECT explode(split(line, '\s'))
    AS word FROM docs) temp
GROUP BY word
ORDER BY word;
```

Source: https://en.wikipedia.org/wiki/Apache_Hive



[https://commons.wikimedia.org/](https://commons.wikimedia.org/wiki/File:Apache_Spark_logo.svg)

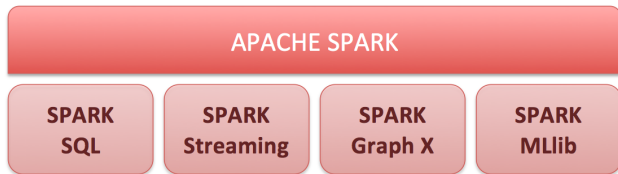
[wiki/File:Apache_Spark_logo.svg](https://commons.wikimedia.org/wiki/File:Apache_Spark_logo.svg)

- ▶ Origin at UC Berkeley in 2009
- ▶ Donated to Apache Software Foundation in 2013
- ▶ Quickly adopted due to performance advantages over MapReduce
- ▶ Builds on Hadoop HDFS and YARN
- ▶ Can use other file storage (S3, Azure, etc.) and cluster managers (Mesos, Kubernetes, etc.)

Apache Spark – Characteristics

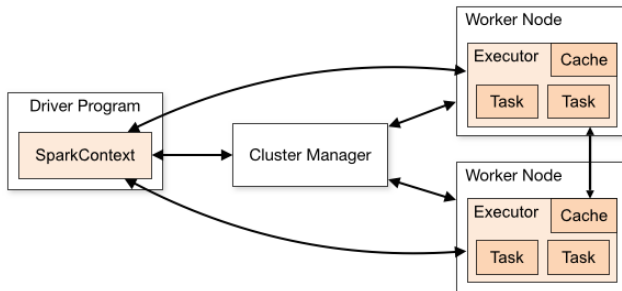
- ▶ **In-Memory Processing** with spill-over to disk
- ▶ **Efficiency** and much faster processing speed compared to MapReduce.
- ▶ **Integration** with Hadoop and its components
- ▶ **Unified Engine** for batch processing, real-time streaming, machine learning, and graph processing
- ▶ **Advanced Analytics** for machine learning, graph processing, SQL and structured data processing
- ▶ **Language Support** for Java, Scala, Python, and R
- ▶ **Scalability** from a single server to thousands of nodes.
- ▶ **Fault Tolerance** through execution engine that provides "lineage" information to recompute lost data
- ▶ **Ease of Use** allows quicker development than Hadoop's MapReduce.

Apache spark – Main Components



`https://commons.wikimedia.org/wiki/File:Sch%C3%A9ma_d%C3%A9tail_outils_spark.png`

Apache Spark – Cluster Overview



<https://spark.apache.org/docs/latest/img/cluster-overview.png>

- ▶ YARN manages resources for Spark applications
- ▶ Spark RDD partitions correspond to HDFS blocks
- ▶ Spark is location aware, schedules job execution on nodes close to data
- ▶ Two layers of fault tolerance: HDFS replication and Spark RDD lineage

Apache Spark – Data Abstractions

1 RDD ("Resilient Distributed Dataset")

- ▶ Dependencies: Data "lineage" information; how the RDD is computed; may be used to reconstruct an RDD
- ▶ Partitions: Split data among executors; parallelize computation, with location info
- ▶ Low-level programming interface focused on MapReduce
- ▶ Procedural, no query optimization

2 DataFrame

- ▶ Inspired by Pandas and R data frames
- ▶ Builds on RDD
- ▶ Named columns with data types
- ▶ High-level programming interface
- ▶ Immutable
- ▶ Declarative, query optimization ("Catalyst" query optimizer)

3 Dataset

- ▶ Strongly typed variant of DataFrame for Java and Scala

Apache Spark – Execution Principles

Transformations

- ▶ Transform a DataFrame into another DataFrame without altering the original (immutability)
- ▶ **Examples:** `map()`, `select()`, `filter()`, `groupBy()`, `orderBy()`, `join()`
- ▶ Recorded in data lineage
- ▶ **Lazy evaluation:** Delay execution until **action** is invoked; allows query optimization

Actions

- ▶ Returns a result or writes result to storage
- ▶ Does not produce another DataFrame
- ▶ **Examples:** `collect()`, `count()`, `show()`, `take()`
- ▶ Triggers execution of transformations

Apache Spark – Basics

Start the local Hadoop cluster (if not already running) and the PySpark console:

```
sudo systemctl start hadoop.service  
pyspark --master yarn
```

Read a file from HDFS and get some statistics:

```
textFile = spark.read.text( \  
    'hdfs://localhost:9000/user/busi4720/hamlet.txt')  
# Number of lines  
textFile.count()  
# First row  
textFile.first()  
# How many lines contain the word Hamlet?  
textFile \  
    .filter(textFile.value.contains("Hamlet")) \  
    .count()
```

Word count example in Pyspark:

```
# Import useful functions from Spark SQL:
from pyspark.sql import functions as sf

wordCounts = textFile \
    .select(sf.explode(sf.split(textFile.value, "\s+")) \
        .alias("word")) \
    .groupBy("word") \
    .count() \
    .orderBy("count")
wordCounts.collect()
```

Apache Spark – Basic Transformations and Actions

Transformations (PySpark)

<code>select()</code>	<code>filter()</code>	<code>where()</code>
<code>withColumnn()</code>	<code>groupBy()</code>	<code>sort()</code>
<code>distinct()</code>	<code>drop()</code>	<code>cov()</code>
<code>orderBy()</code>	<code>withColumnRenamed()</code>	<code>union()</code>
<code>join()</code>		

Actions (PySpark)

<code>show()</code>	<code>collect()</code>	<code>take()</code>
<code>count()</code>	<code>head()</code>	<code>tail()</code>
<code>write.csv()</code>	<code>toPandas()</code>	

Apache Spark – Schemas and DataFrames

Schema describes the columns of data frames and their types.

- ▶ No need for Spark to infer types
- ▶ No need for Spark to read data to infer schema
- ▶ Error detection when reading data

Define a schema using Spark schema DDL:

```
logSchema = \  
  'caseID STRING, \  
  activity STRING, \  
  ts TIMESTAMP'
```

Spark DDL Data Types

STRING	TINYINT	SMALLINT	INT
BIGINT	BOOLEAN	FLOAT	DOUBLE
DATE	DECIMAL	TIMESTAMP	BINARY
STRUCT	ARRAY	MAP	

Apache Spark – Schemas and DataFrames

Read the data from HDFS into a data frame:

```
fname='hdfs://localhost:9000/user/busi4720/\
eventlog.short.log'

data = spark.read \
    .format('csv') \
    .option('delimiter', '\t') \
    .option('header', 'false') \
    .schema(logSchema) \
    .load(fname)
```

Query the schema, count rows, show 5 rows, and a summary:

```
data.printSchema()
data.count()
data.show(5)
data.summary().show()
```

Apache Spark – SQL

Register a data frame as a temporary SQL table ("view"):

```
data.createOrReplaceTempView('log')
```

Alternatively, create a permanent SQL table:

```
data.write.saveAsTable('log_table')
```

Query the SQL table, will return a data frame:

```
result_df = spark.sql('select * from log limit 5')  
result_df.show()
```


Apache Spark – SQL

Create a Directly-Follows-Graph (DFG) from a log. Define the SQL Query:

```
sql_query = \  
'SELECT COUNT(*), l1.activity AS activity1, \  
  l2.activity AS activity2, AVG(l2.ts - l1.ts) AS dtime \  
  FROM log AS l1 JOIN log AS l2 ON l1.caseid=l2.caseid \  
  WHERE l2.ts = (SELECT MIN(ts) FROM log l3 \  
    WHERE l3.caseid=l1.caseid AND l3.ts > l1.ts) \  
  GROUP BY GROUPING SETS((l1.activity, l2.activity))'
```

Run the query, show the results and explain the query plan:

```
dfg = spark.sql(sql_query)  
dfg.count()  
dfg.show()  
  
dfg.explain(mode='formatted')  
dfg.explain(True)
```

Run as self-contained application:

```
# Download file
wget https://evermann.ca/busi4720/spark_dfg.py
# Submit to Spark/Hadoop cluster
spark-submit --master yarn spark_dfg.py \
  hdfs://localhost:9000/user/busi4720/eventlog.short.log
```

Result will be written to HDFS.

Job Tracker

Use Hadoop Job Tracker at `https://localhost:8088` to track status of nodes and progress of jobs.

Apache Spark – Compare to PostgreSQL

```
psql
```

Create table and read data from CSV file:

```
CREATE TABLE log(  
  caseId VARCHAR(20),  
  activity VARCHAR(10),  
  ts TIMESTAMP);  
\COPY log FROM 'eventlog.short.log'  
  WITH DELIMITER E'\t';
```

Execute query:

```
SELECT COUNT(*), 11.activity, 12.activity,  
AVG(12.ts - 11.ts) AS dtme  
FROM log AS 11 JOIN log AS 12 ON 11.caseid=12.caseid  
WHERE 12.ts = (SELECT MIN(ts) FROM log 13  
  WHERE 13.caseid=11.caseid AND 13.ts > 11.ts)  
GROUP BY (11.activity, 12.activity);
```

Spark ML Frameworks

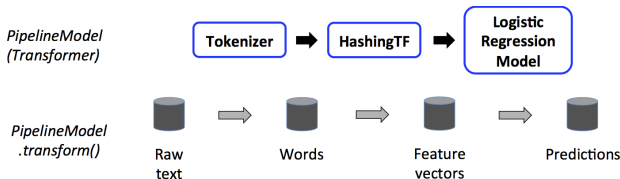
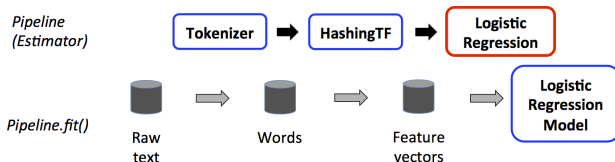
- ▶ **Spark.mllib**: RDD focused, maintenance only
- ▶ **Spark.ml**: Dataframe focused, actively developed

Spark ML Techniques

- ▶ **Supervised**: Classification, Regression
- ▶ **Unsupervised**: Clustering, Principal Component Analysis, etc.

Apache Spark – ML Pipelines

- **Transformers:** Accept a dataframe, execute `transform()` method, return a dataframe
- **Estimators:** Accept a dataframe, execute `fit()` method, return a transformer



<https://spark.apache.org/docs/latest/ml-pipeline.html>

Apache Spark – Selection of available ML Models

Classification

Logistic regression	Decision trees	Random forests
Gradient-boosted trees	Multilayer perceptron	Support vector machines
	Naive bayes	

Regression

Linear regression	Generalized linear regression
Decision tree regression	Random forest regression
GBT regression	Survival regression

Unsupervised

K-means clustering	Principal components
--------------------	----------------------

Apache Spark – ML Example

Full example at

https://evermann.ca/busi4720/spark_ml.py

Run as self-contained application:

```
# Download file
wget https://evermann.ca/busi4720/spark_ml.py
# Submit to Spark/Hadoop cluster
spark-submit --master yarn spark_ml.py \
    hdfs://localhost:9000/user/busi4720/mushrooms.csv
```

Job Tracker

Use Hadoop Job Tracker at <https://localhost:8088> to track status of nodes and progress of jobs.

Apache Spark – ML Example

Get the dataset:

```
wget https://evermann.ca/busi4720/mushrooms.csv  
hdfs dfs -put mushrooms.csv
```

<https://archive.ics.uci.edu/dataset/848/secondary+mushroom+dataset>
CC-BY 4.0 license

Define the schema:

```
the_schema = 'class STRING, `cap-diameter` DOUBLE, \  
  `cap-shape` STRING, `cap-surface` STRING, \  
  `cap-color` STRING, `does-bruise-or-bleed` STRING, \  
  `gill-attachment` STRING, `gill-spacing` STRING, \  
  `gill-color` STRING, `stem-height` DOUBLE, \  
  `stem-width` DOUBLE, `stem-root` STRING, \  
  `stem-surface` STRING, `stem-color` STRING, \  
  `veil-type` STRING, `veil-color` STRING, \  
  `has-ring` STRING, `ring-type` STRING, \  
  `spore-print-color` STRING, habitat STRING, \  
  season STRING'
```


Load data:

```
fname='hdfs://localhost:9000/user/busi4720/\nmushrooms.csv'

data = spark.read \
    .format('csv') \
    .option('delimiter', ',') \
    .option('header', 'true') \
    .schema(the_schema) \
    .load(fname)
data = data.drop('veil-type')
data = data.fillna('NULL')
```

Apache Spark – ML Example

Import all required pieces:

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import StandardScaler, \
    StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml.evaluation \
    import BinaryClassificationEvaluator
from pyspark.ml import PipelineModel
```

Create the necessary **transformers** for the pipeline. Collect all numerical features:

```
numFeatures = VectorAssembler(
    inputCols = ['cap-diameter', 'stem-width',
                 'stem-height'],
    outputCol = 'numFeatures')

scaler = StandardScaler(inputCol='numFeatures',
                        outputCol='numFeaturesS')
```

Encode categorical variables as one-hot (dummy variables):

```
categoricalCols = \
    [name for (name, dtype) in data.dtypes \
     if dtype=='string']
indexOutputCols = \
    [x + 'index' for x in categoricalCols]
oheOutputCols = \
    [x + 'ohe' for x in categoricalCols]

stringIndexer = StringIndexer(
    inputCols = categoricalCols,
    outputCols = indexOutputCols,
    handleInvalid='skip')

oheEncoder = OneHotEncoder(
    inputCols = indexOutputCols,
    outputCols = oheOutputCols)
```

Apache Spark – ML Example

Assemble all features into a feature vector:

```
vecAssembler = VectorAssembler(  
    inputCols = oheOutputCols+['numFeaturesS'],  
    outputCol = 'feature_vec')
```

Encode the target classes as numbers:

```
stringIndexTarget = StringIndexer(  
    inputCols = ['class'],  
    outputCols = ['classIndex'],  
    handleInvalid='skip')
```

Create the classification **estimator**:

```
logReg = LogisticRegression(  
    featuresCol = 'feature_vec',  
    labelCol = 'classIndex')
```

Put all components into the **pipeline**:

```
pipeline = Pipeline(stages=[
    numFeatures,
    scaler,
    stringIndexer,
    oneEncoder,
    vecAssembler,
    stringIndexTarget,
    logReg])
```

Apache Spark – ML Example

Create train/test data split:

```
train_data, test_data = \  
    data.randomSplit([.66, .33], seed=1)
```

Fit the model to the training data:

```
pipelineModel = pipeline.fit(train_data)
```

Summary of the training data:

```
summary = pipelineModel.stages[-1].summary  
summary.accuracy  
summary.areaUnderROC  
summary.fMeasureByThreshold.show()  
summary.precisionByLabel  
summary.recallByLabel  
summary.roc.show()
```

Apache Spark – ML Example

Fitted estimators (including whole pipelines) become transformers. Predict for both training and testing data:

```
trainPred = pipelineModel.transform(train_data)
testPred = pipelineModel.transform(test_data)
```

Evaluate the model using AUC:

```
evaluator = BinaryClassificationEvaluator(
    labelCol='classIndex')
evaluator.evaluate(trainPred)
evaluator.evaluate(testPred)
```

Save the fitted model for later re-use:

```
pipelineModel.write().overwrite().save('myFirstModel')
```

Load a saved model:

```
savedModel = PipelineModel.load('myFirstModel')
```

Stream Analytics

- ▶ Network ("directed acyclic graph") of nodes
- ▶ Ingest records, process records, emit records
- ▶ Record-by-record processing
- ▶ Low latencies
- ▶ Resource intensive

Example Use Cases

- ▶ Customer click-stream analysis for real-time pricing
- ▶ Machine sensor data for failure warnings/alarms
- ▶ Financial/payments transaction fraud monitoring
- ▶ Market data analytics, news monitoring
- ▶ Activity records for process compliance monitoring
- ▶ ...

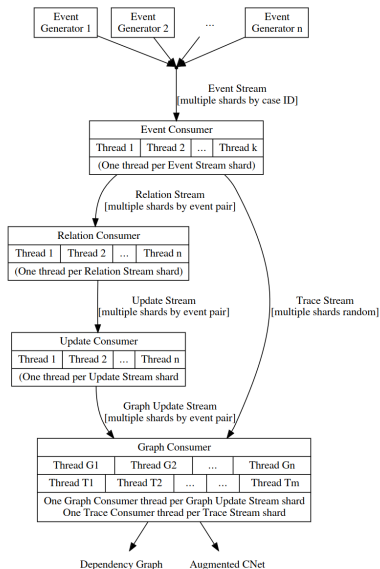
Process Discovery with Streaming Data



- ▶ Flexible Heuristics Miner
- ▶ Activity completion records
- ▶ Record-by-record processing
- ▶ 57,000 – 160,000 traces per min
- ▶ 2,000,000 – 5,500,000 events per min
- ▶ 5 16-core, 32GB nodes

Source: Evermann, J., Rehse, J.-R., and Fettke, P.: Process Discovery from Event Stream Data in the Cloud - A Scalable, Distributed Implementation of the Flexible Heuristics Miner on the Amazon Kinesis Cloud Infrastructure. *CloudBPM Workshop on Business Process Monitoring and Performance Analysis in the Cloud at the 8th IEEE International Conference on Cloud Computing Technologies and Science (CloudCom 2016)* .

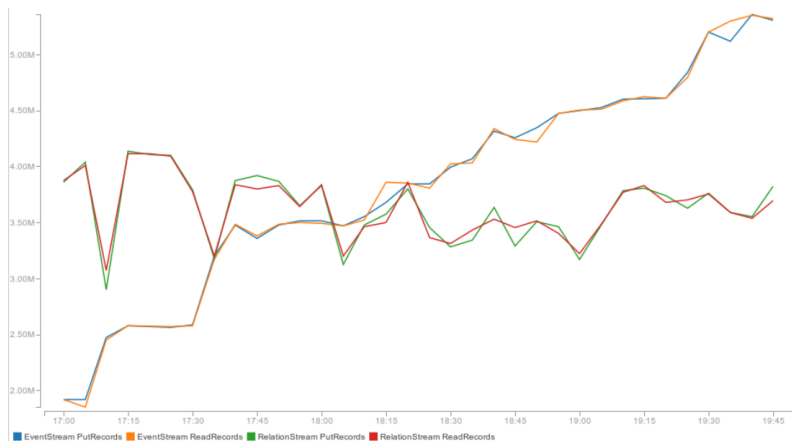
Process Discovery with Streaming Data



- Implemented on AWS Kinesis
- Directed acyclic graph (DAG)
- Multiple event generators
- Multiple record streams
- Stream is queue with "Put" and "Read" operations
- Streams contain multiple "shards" (same keys)
- Multiple threads/executors per shard (key)

Process Discovery with Streaming Data

Performance Results:



Principles

- ▶ Micro-batches
- ▶ Data stream as unbounded dataframe/table
- ▶ Unified programming model for batch and stream processing
- ▶ Support for time windows
- ▶ Wide range of input sources and output destinations
- ▶ Optimized stateful stream transformations and aggregations

Apache Spark – Stream Analytics

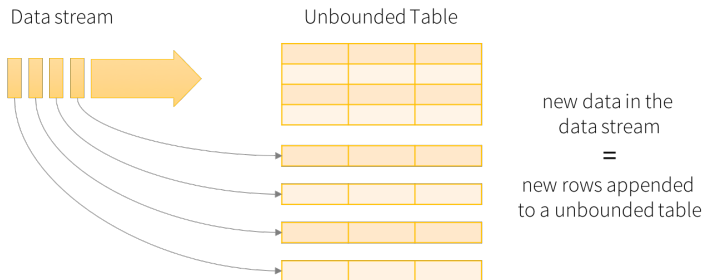


<https://spark.apache.org/docs/latest/img/streaming-arch.png>



<https://spark.apache.org/docs/latest/img/streaming-flow.png>

Apache Spark – Stream Analytics



Data stream as an unbounded table

<https://spark.apache.org/docs/latest/img/structured-streaming-stream-as-a-table.png>

Triggering

- ▶ Micro-batch (process next batch when prior batch completed)
- ▶ Time trigger
- ▶ Once
- ▶ Continuous

Output Modes

- ▶ Append: Assume older output remains valid
- ▶ Update: Change parts of older output (requires appropriate output destination, e.g. PostgreSQL, but not HDFS)
- ▶ Complete: Replace/overwrite older output

Apache Spark – Stream Analytics

Import all necessary functions and get a Spark Session:

```
from pyspark.sql.functions import \
    explode, split, col, desc, \
    window, current_timestamp
```

Create the stream reader to read from a network socket:

```
lines = spark.readStream \
    .format('socket') \
    .option('host', 'localhost') \
    .option('port', 9999) \
    .load()
```

`lines` is a `DStream`.

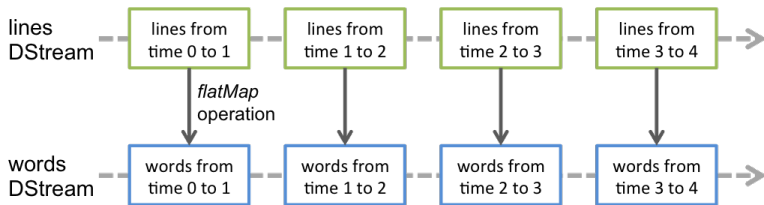
The stream reader opens a *client* socket, i.e. the socket must already be opened for writing.

Apache Spark – Stream Analytics

Define processing of lines:

```
words = lines.select(explode(split(col('value'), \
    '\\s')).alias('word'))
```

words is another DStream, connected to lines



[https:](https://spark.apache.org/docs/latest/img/streaming-dstream-ops.png)

[//spark.apache.org/docs/latest/img/streaming-dstream-ops.png](https://spark.apache.org/docs/latest/img/streaming-dstream-ops.png)

Define processing of words:

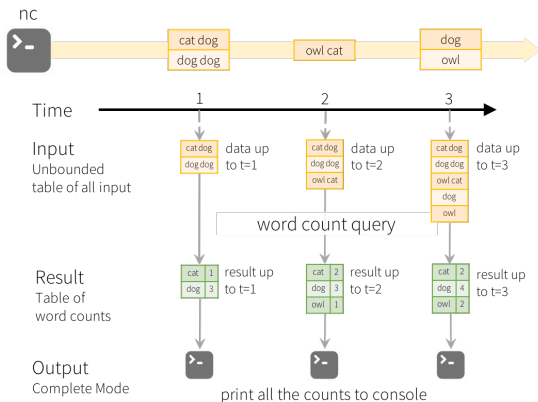
```
counts = words.groupBy('word') \
                .count() \
                .sort(desc('count'))
```

`counts` is another `Dstream`, connected to `words`

Define the output writer with output mode and processing trigger:

```
writer = counts.writeStream \
                .format('console') \
                .outputMode('complete') \
                .trigger(processingTime='5 second') \
                .option('checkpointLocation', \
                        'hdfs://localhost:9000/user/busi4720/')
```

Apache Spark – Stream Analytics



Model of the Quick Example

<https://spark.apache.org/docs/latest/img/structured-streaming-example-model.png>

First, open a server socket from the shell using `nc`:

```
nc -kl 9999
```

Start the processing by starting the writer. This returns a streaming query object that provides progress information. "`start()`" is a non-blocking operation:

```
streamingQuery = writer.start()
```

Get progress information through the `lastProgress` attribute of the query:

```
print(streamingQuery.lastProgress)
```

Stop the processing by calling `stop()` on the query object:

```
streamingQuery.stop()
```

Fault Tolerance

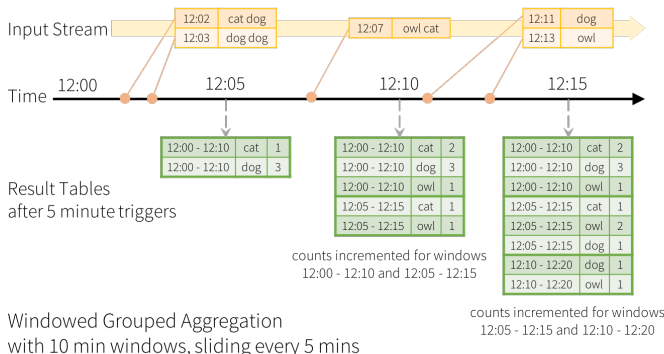
- ▶ Checkpointing of state
- ▶ "End-to-end exactly-once" guarantees
 - ▶ Replayable sources
 - ▶ Deterministic computations
 - ▶ Destination that can identify duplicates

Time Windowing

```
words = lines.select(explode(split(col('value'), \
    '\\s')).alias('word')) \
    .withColumn('eventTime', current_timestamp())

counts = words \
    .groupBy('word', \
        window('eventTime', '1 minute', '30 second')) \
    .count() \
    .sort(desc('count'))
```

Apache Spark – Stream Analytics



<https://spark.apache.org/docs/latest/img/structured-streaming-window.png>

- 1 Streaming ML Learning
 - ▶ Streaming Linear Regression
 - ▶ Streaming Logistic Regression
 - ▶ Streaming KMeans
- 2 Streaming ML Prediction
 - ▶ From off-line trained models

Quick Start

<https://spark.apache.org/docs/latest/quick-start.html>

SQL, DataFrames and Datasets

<https://spark.apache.org/docs/latest/sql-programming-guide.html>

Structured Streaming

<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>

Machine Learning

<https://spark.apache.org/docs/latest/ml-guide.html>

Last accessed on March 8, 2024