SCIENCE
PASSION
TECHNOLOGY

# Database Systems
# 12 Distributed Analytics

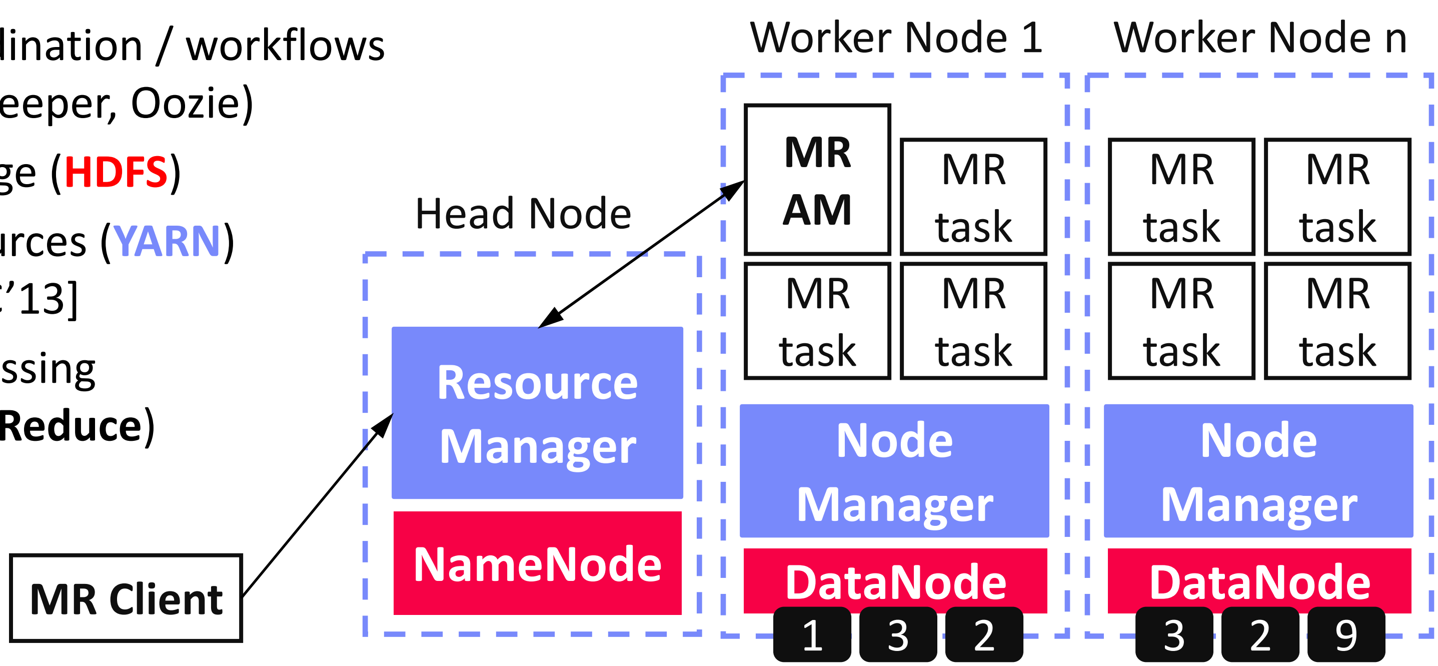
**Matthias Boehm, Arnab Phani**

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management

Last update: Jan 10, 2020

# Hadoop History and Architecture

- **Recap: Brief History**
  - Google's GFS [SOSP'03] + MapReduce [ODSI'04] → Apache Hadoop (2006)
  - Apache Hive (SQL), Pig (ETL), Mahout (ML), Giraph (Graph)
- **Hadoop Architecture / Eco System**
  - Management (Ambari)
  - Coordination / workflows (Zookeeper, Oozie)
  - Storage (**HDFS**)
  - Resources (**YARN**) [SoCC'13]
  - Processing (**MapReduce**)



# MapReduce – Programming Model

- **Overview Programming Model**
  - Inspired by functional programming languages
  - **Implicit parallelism** (abstracts distributed storage and processing)
  - Map function: key/value pair → set of intermediate key/value pairs
  - Reduce function: merge all intermediate values by key

- **Example** `SELECT Dep, count(*) FROM csv_files GROUP BY Dep`

| Name | Dep |
|---|---|
| X | CS |
| Y | CS |
| A | EE |
| Z | CS |

Collection of key/value pairs

```
map(Long pos, String line) {
  parts ← line.split(“,”)
  emit(parts[1], 1)
}
```

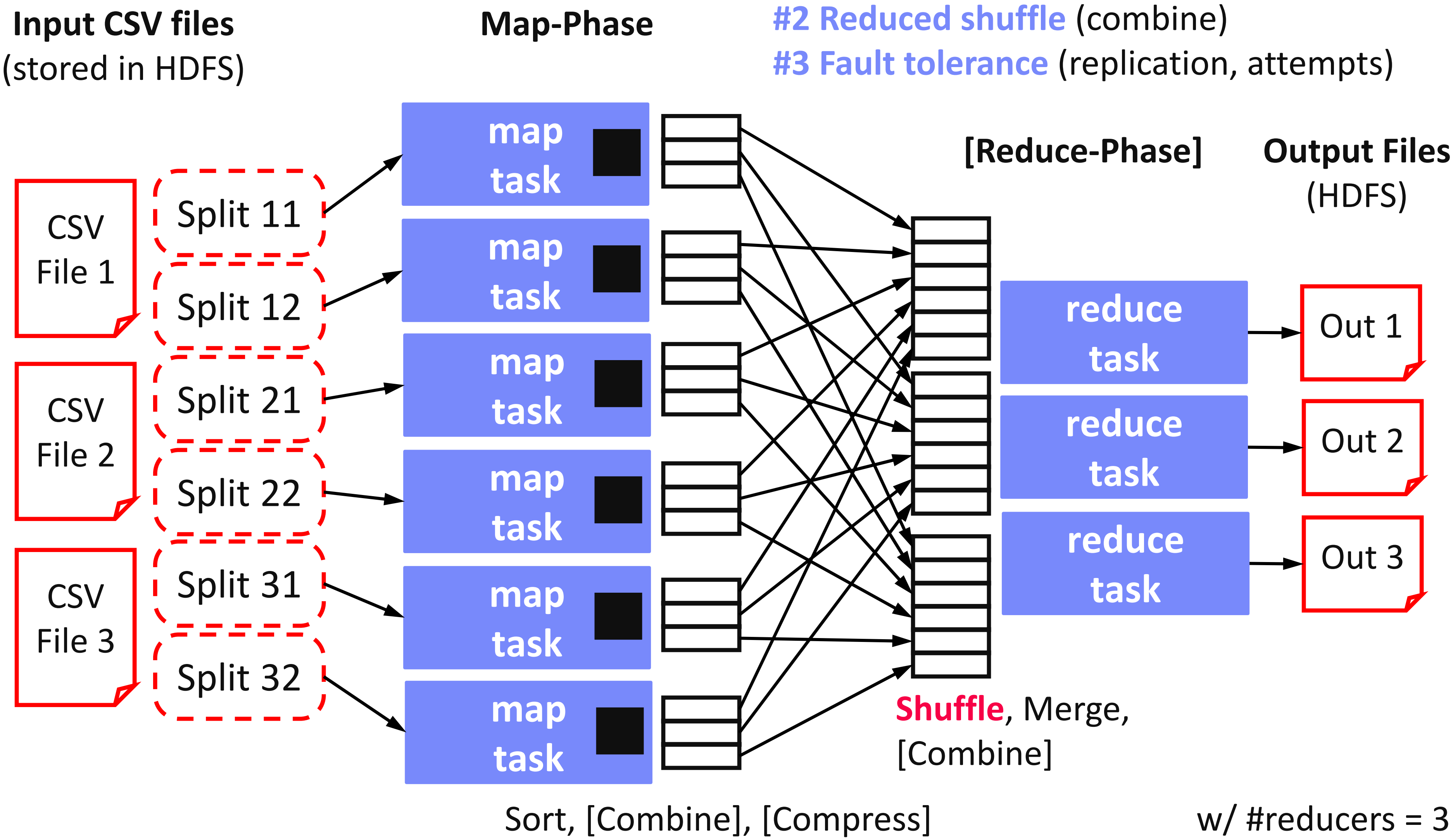
| | |
|---|---|
| CS | 1 |
| CS | 1 |
| EE | 1 |
| CS | 1 |

```
reduce(String dep,
       Iterator<Long> iter) {
  total ← iter.sum();
  emit(dep, total)
}
```

| | |
|---|---|
| CS | 3 |
| EE | 1 |

# MapReduce – Execution Model

**#1 Data Locality** (delay sched., write affinity)
**#2 Reduced shuffle** (combine)
**#3 Fault tolerance** (replication, attempts)



# Spark History and Architecture

- **Summary MapReduce**
  - Large-scale & fault-tolerant processing w/ UDFs and files ➔ **Flexibility**
  - Restricted functional APIs ➔ **Implicit parallelism and fault tolerance**
  - **Criticism**: #1 **Performance**, #2 **Low-level APIs**, #3 **Many different systems**

- **Evolution to Spark** (and Flink)
  - Spark [HotCloud'10] + RDDs [NSDI'12] → **Apache Spark** (2014)
  - **Design: standing executors with in-memory storage**, lazy evaluation, and fault-tolerance via RDD lineage
  - **Performance:** In-memory storage and fast job scheduling (100ms vs 10s)
  - **APIs:** Richer functional APIs and general computation DAGs, high-level APIs (e.g., DataFrame/Dataset), unified platform

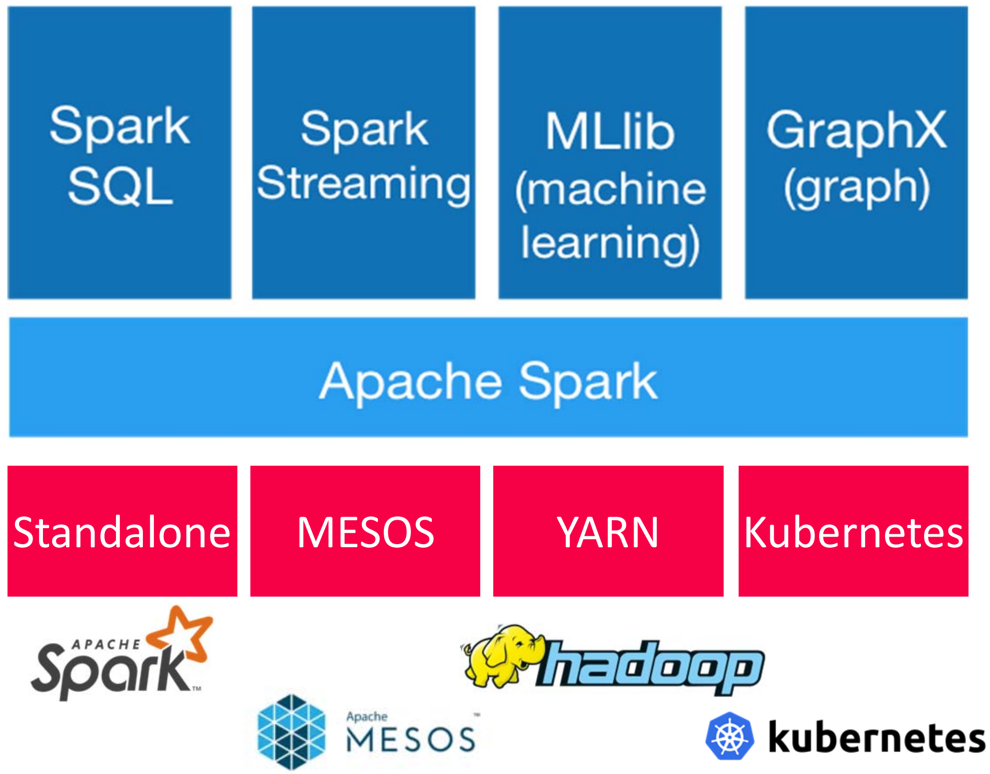
➔ **But many shared concepts/infrastructure**

- **Implicit parallelism through dist. collections** (data access, fault tolerance)
- Resource negotiators (YARN, Mesos, Kubernetes)
- HDFS and object store connectors (e.g., Swift, S3)

# Spark History and Architecture, cont.

- **High-Level Architecture**
  - **Different language bindings**: Scala, Java, Python, R
  - **Different libraries**: SQL, ML, Stream, Graph
  - Spark core (incl RDDs)
  - **Different cluster managers**: Standalone, Mesos, **Yarn**, **Kubernetes**
  - Different file systems/ formats, and data sources: **HDFS**, **S3**, SWIFT, **DBs**, **NoSQL**

[https://spark.apache.org/]




- **Focus on a unified platform for data-parallel computation**

# Resilient Distributed Datasets (RDDs)

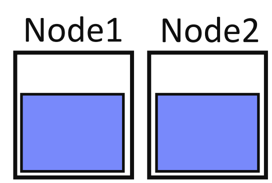
- **RDD Abstraction**
  - **Immutable**, partitioned **collections of key-value pairs**
  - **Coarse-grained** deterministic operations (transformations/actions)
  - Fault tolerance via lineage-based re-computation

```
JavaPairRDD
  <MatrixIndexes,MatrixBlock>
```

- **Operations**
  - Transformations: define new RDDs
  - Actions: return result to driver

| Type | Examples |
|---|---|
| Transformation (**lazy**) | `map, hadoopFile, textFile, flatMap, filter, sample, join, groupByKey, cogroup, reduceByKey, cross, sortByKey, mapValues` |
| Action | `reduce, save, collect, count, lookupKey` |

- **Distributed Caching**
  - Use fraction of worker **memory for caching**
  - Eviction at granularity of individual partitions
  - **Different storage levels** (e.g., mem/disk x serialization x compression)

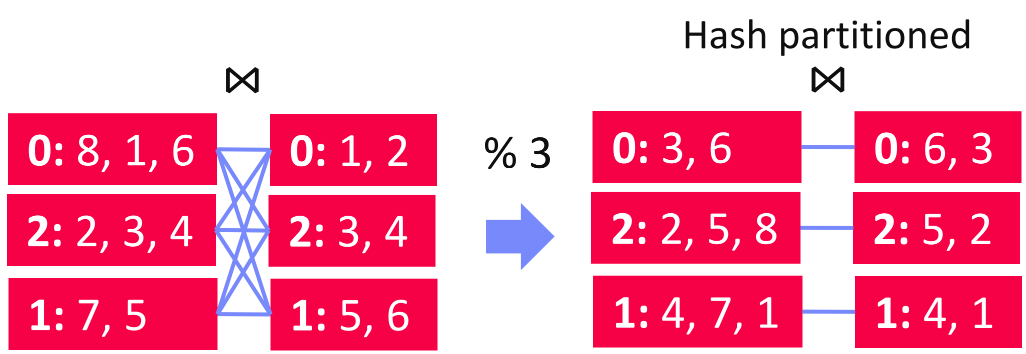


# Partitions and Implicit/Explicit Partitioning

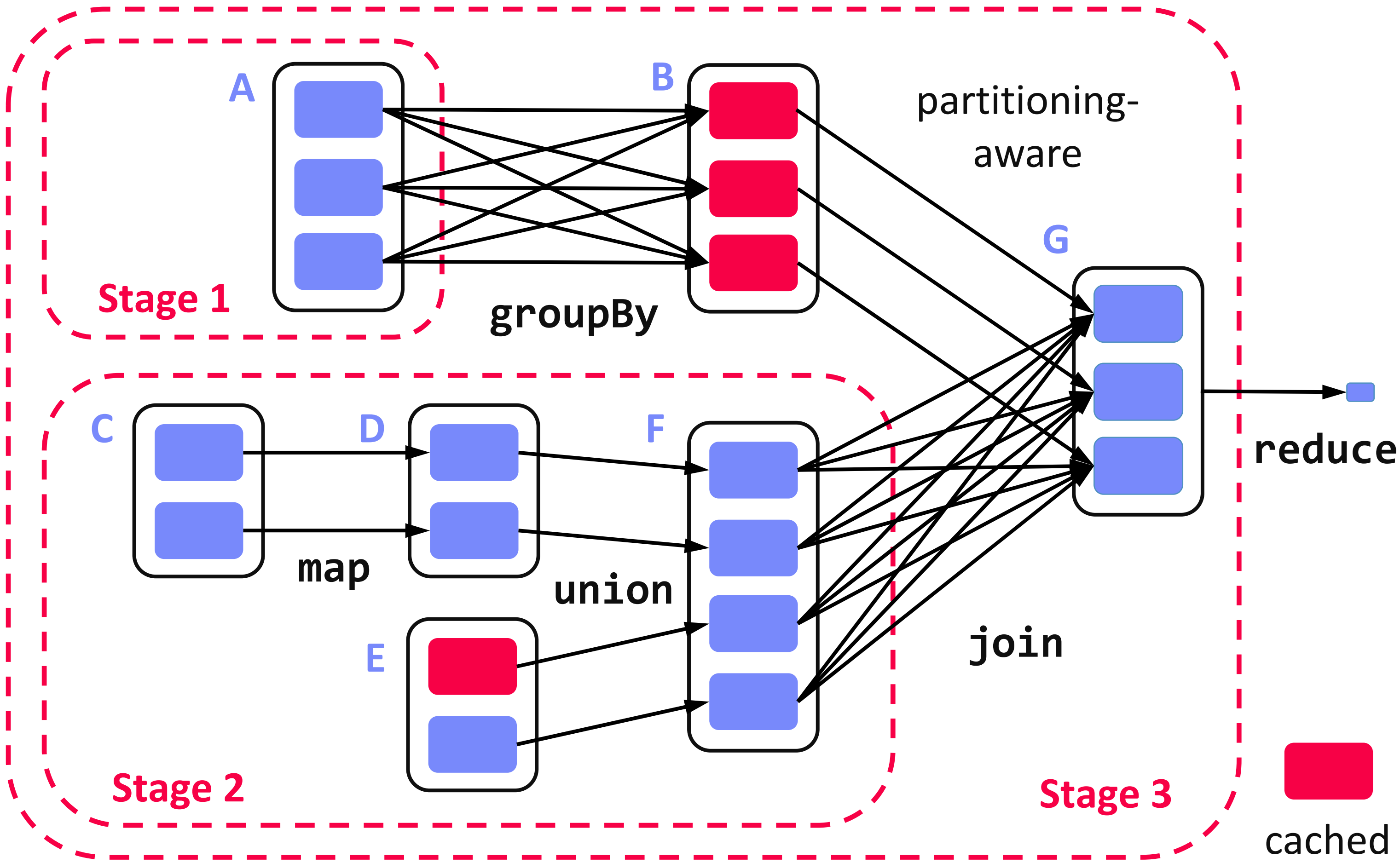
- **Spark Partitions**
  - Logical key-value collections are split into **physical partitions**
  - Partitions are granularity of **tasks, I/O** (HDFS blocks/files)**, shuffling, evictions**
- **Partitioning via Partitioners**
  - Implicitly on every data shuffling
  - Explicitly via `R.repartition(n)`

**Example Hash Partitioning:**
For all (k,v) of R:
pid = hash(k) % n

- **Partitioning-Preserving**
  - All operations that are guaranteed to keep keys unchanged (e.g. `mapValues()`, `mapPartitions()` w/ preservesPart flag)
- **Partitioning-Exploiting**
  - Join: R3 = R1.join(R2)
  - Lookups: `v = C.lookup(k)`



# Lazy Evaluation, Caching, and Lineage



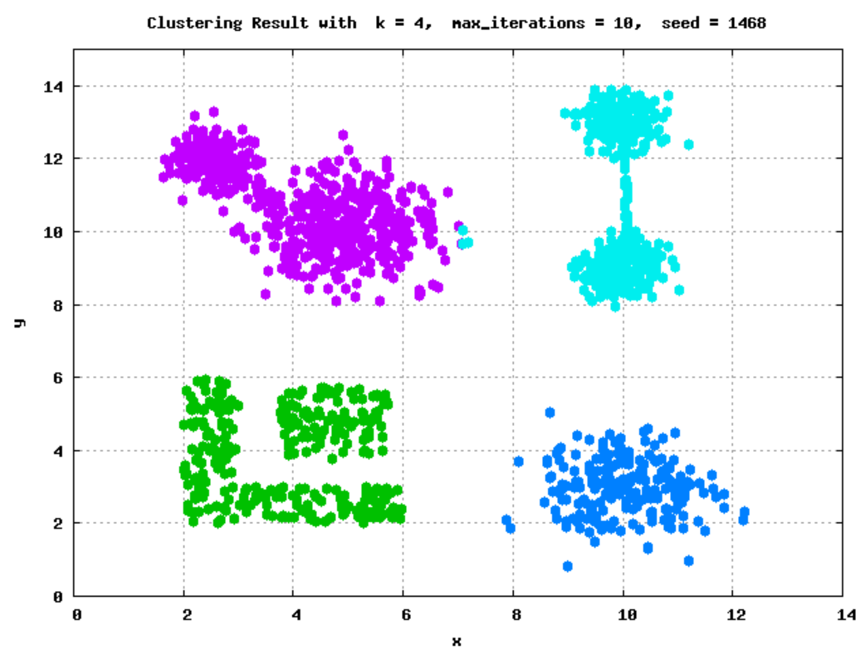

[Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. **NSDI 2012**]

# Example: k-Means Clustering

- **k-Means Algorithm**
  - Given dataset D and number of clusters k, find cluster centroids ("mean" of assigned points) that minimize within-cluster variance
  - Euclidean distance: **sqrt**(**sum**((**a**-**b**)^2))

- **Pseudo Code**

```
function Kmeans(D, k, maxiter) {
  C‘ = randCentroids(D, k);
  C = {};
  i = 0; //until convergence
  while( C‘ != C & i<=maxiter ) {
    C = C‘;
    i = i + 1;
    A = getAssignments(D, C);
    C‘ = getCentroids(D, A, k);
  }
  return C‘
}
```



# Example: K-Means Clustering in Spark

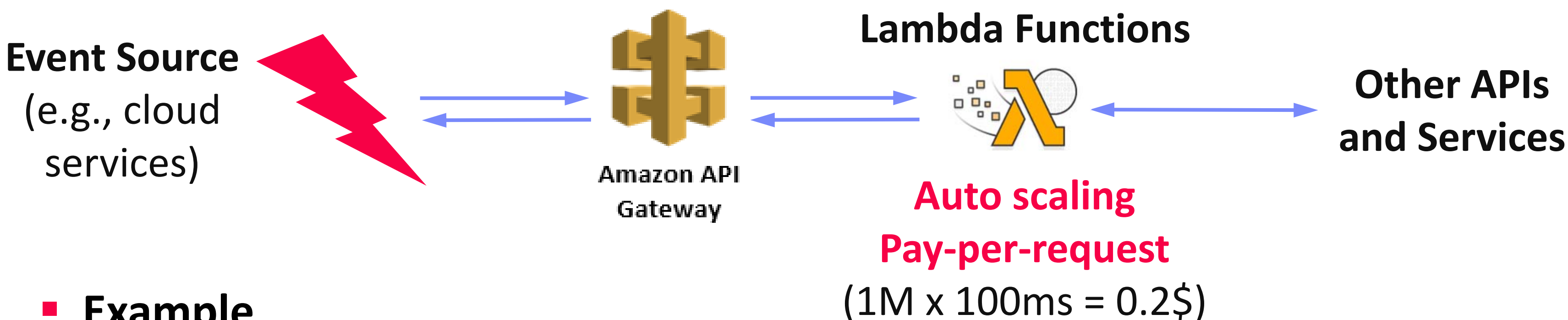
```
// create spark context (allocate configured executors)
JavaSparkContext sc = new JavaSparkContext();

// read and cache data, initialize centroids
JavaRDD<Row> D = sc.textFile("hdfs:/user/mboehm/data/D.csv")
    .map(new ParseRow()).cache(); // cache data in spark executors
Map<Integer,Mean> C = asCentroidMap(D.takeSample(false, k));

// until convergence
while( !equals(C, C2) & i<=maxiter ) {
    C2 = C; i++;
    // assign points to closest centroid, recompute centroid
    Broadcast<Map<Integer,Row>> bC = sc.broadcast(C)
    C = D.mapToPair(new NearestAssignment(bC))
        .foldByKey(new Mean(0), new IncComputeCentroids())
        .collectAsMap();
}

return C;
```

Note: Existing library algorithm
[https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala]

# Serverless Computing

[Joseph M. Hellerstein et al: Serverless Computing: One Step Forward, Two Steps Back. **CIDR 2019**]

- **Definition Serverless**
  - **FaaS:** functions-as-a-service (event-driven, stateless input-output mapping)
  - Infrastructure for deployment and auto-scaling of APIs/functions
  - Examples: **Amazon Lambda**, **Microsoft Azure Functions**, etc




- **Example**

```java
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class MyHandler implements RequestHandler<Tuple, MyResponse> {
    @Override
    public MyResponse handleRequest(Tuple input, Context context) {
       return expensiveStatelessComputation(input);
    }
}
```

# Exercise 4:
# Large-Scale Data Analysis

Published: Dec 31

Deadline: Jan 21

# Task 4.1 Apache Spark Setup

**4/25 points**

- **#1 Pick your Spark Language Binding**
  - Java, Scala, Python
- **#2 Install Dependencies**
  - Java: Maven
    **spark-core, spark-sql**
  - Python:
    **pip install pyspark**

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.11</artifactId>
  <version>2.4.3</version>
 </dependency>
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.4.3</version>
</dependency>
```

- **(#3 Win Environment)**
  - Download https://github.com/steveloughran/winutils/tree/master/hadoop-2.7.1/bin/winutils.exe
  - Create environment variable HADOOP_HOME="<some-path>/hadoop"

# Task 4.2 SQL Query Processing

5/25 points

- **Q09: Top 5 Cities by Route Departures**
  - Consider all their airports
  - Total number of route departures
  - Return (City Name, Number of departures)
  - Sorted in descending order of the number of routes
- **Q10: Frequently used Plane Types**
  - Plane types used on more than 2048 routes
  - Return (Plane type name, Number of routes it is used on)

# Task 4.2 SQL Query Processing, cont.

- **Expected Results with provided Schema and Data**

**Q09: Top 5 Cities by Route Departures**

```
   name    | count
-----------+-------
 London    |  1090
 Atlanta   |   760
 Paris     |   681
 Shanghai  |   603
 Beijing   |   600
(5 rows)
```

**Q10: Frequently used Plane Types**

```
      name        | count
------------------+-------
 Airbus A320      | 15406
 Airbus A319      |  7847
 Boeing 737       |  2751
 Boeing 737-800   | 10329
 Airbus A321      |  3611
(5 rows)
```

# Task 4.3 Query Processing via Spark RDDs

- **#1 Spark Context Creation**
  - Create a spark context sc w/ local master (`local[*]`)

**10/25 points**

- **#2 Implement Q09 via RDD Operations**
  - Implement Q09 self-contained in executeQ09RDD()
  - All reads should use `sc.textFile(fname)`
  - RDD operations only → stdout

See Spark online documentation for details

- **#3 Implement Q10 via RDD Operations**
  - Implement Q10 self-contained in executeQ10RDD()
  - All reads should use `sc.textFile(fname)`
  - RDD operations only → stdout

# Task 4.4 Query Processing via Spark SQL

- **#1 Spark Session Creation**
  - Create a spark session via a spark session builder and w. local master (`local[*]`)

**6/25 points**

- **#2 Implement Q09 via Dataset Operations**
  - Implement Q09 self-contained in executeQ09Dataset()
  - All reads should use `sc.read().format("csv")`
  - SQL or Dataset operations only → JSON

See Spark online documentation for details

- **#3 Implement Q10 via Dataset Operations**
  - Implement Q10 self-contained in executeQ10Dataset()
  - All reads should use `sc.read().format("csv")`
  - SQL or Dataset operations only → JSON

**→ SQL processing of high importance in modern data management**

# Task 4.4 Query Processing via Spark SQL, cont.

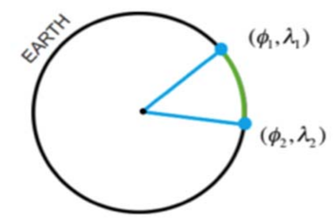
- **Optional: Explore Spark Web UI**
  - Web UI started even in local mode
  - Explore distributed jobs and stages
  - Explore effects of caching on repeated query processing
  - Explore statistics

```
INFO Utils: Successfully started
   service 'SparkUI' on port 4040.
INFO SparkUI: Bound SparkUI to 0.0.0.0, and
   started at http://192.168.108.220:4040
```

# Task 4.5 Extra Credit: SQL Query Processing

- **Q11: Longest route computed via Haversine distance** **5 points**
  - Longest route in km
  - Computed via Haversine distance (using longitude & latitude)
  - Return (Departure City Name, Arrival City Name, Distance in km)

$$\text{haversine}\left(\frac{d}{r}\right) = \text{haversine}(\phi_2 - \phi_1) + \cos(\phi_1)\cos(\phi_2)\text{haversine}(\lambda_2 - \lambda_1)$$




$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Where, Ø = latitude and ƛ = longitude

# Conclusions and Q&A

- **Summary 11/12 Distributed Storage/Data Analysis**
  - Cloud Computing Overview
  - Distributed Storage
  - Distributed Data Analytics

- **Next Lectures (Part B: Modern Data Management)**
  - **13 Data stream processing systems** [**Jan 20**]
  - **Jan 27: Q&A and exam preparation**