Apache Spark Introduction

2016. 05.

Hwang Joongwon

Index

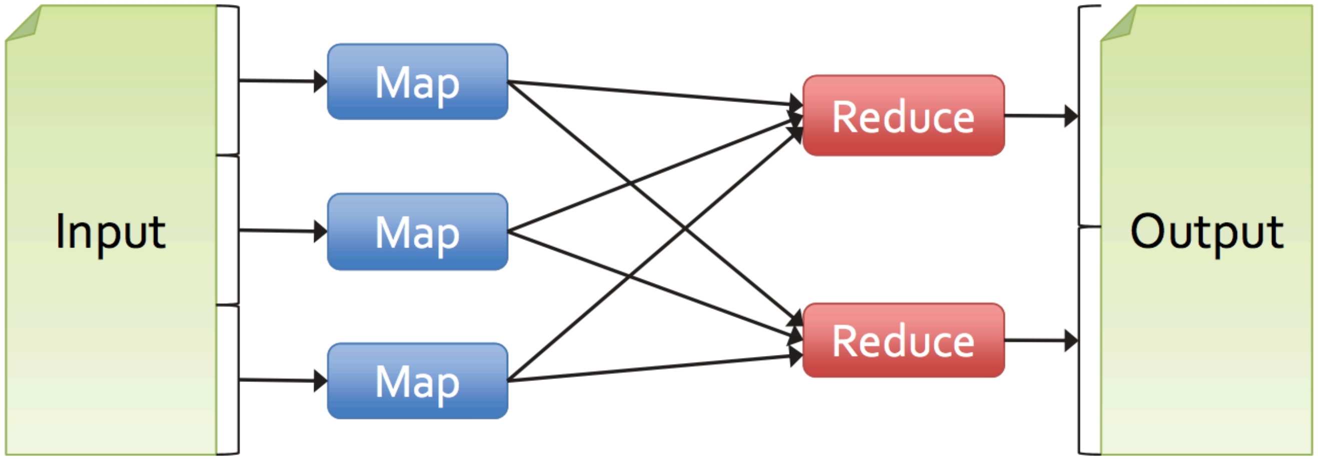
- Apache Spark Introduction
  - Motivation
  - Resilient Distributed Dataset (RDD)
  - Lineage chain for RDD
  - Lazyness Execution
  - RDD dependency
  - Caching
- Machine Learning in ApacheSpark
  - Machine Leaning library (MLlib)
  - Collaborative Fillltering(CF)
  - Alternative Least Square (ALS)
- Appendix

Apache Spark Introduction

Motivation
Resilient Distributed Dataset (RDD)
Lineage chain for RDD
Lazyness Execution
RDD dependency
Caching

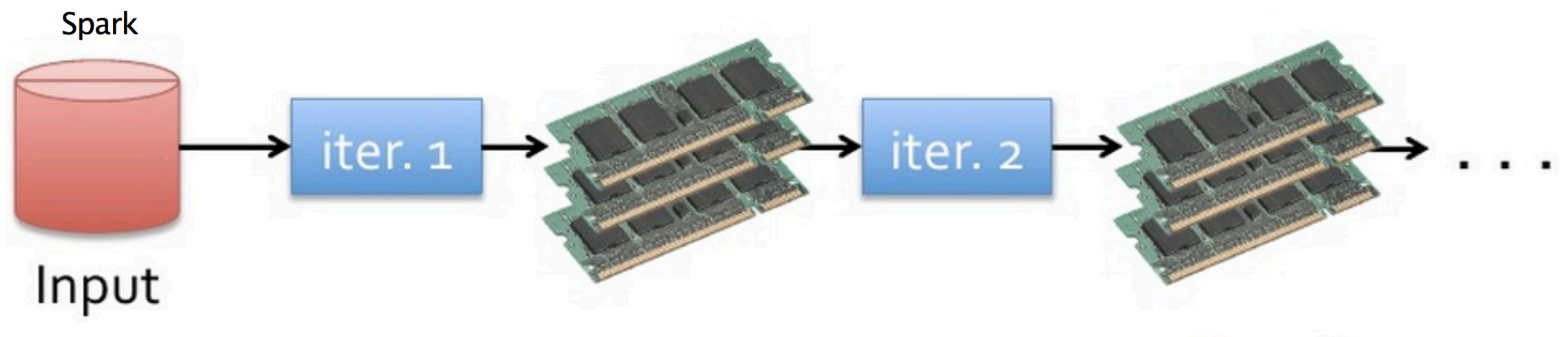
Motivation

- Current popular programming models for clusters transform data flowing from stable storage to stable storage.
  - E.g., MapReduce

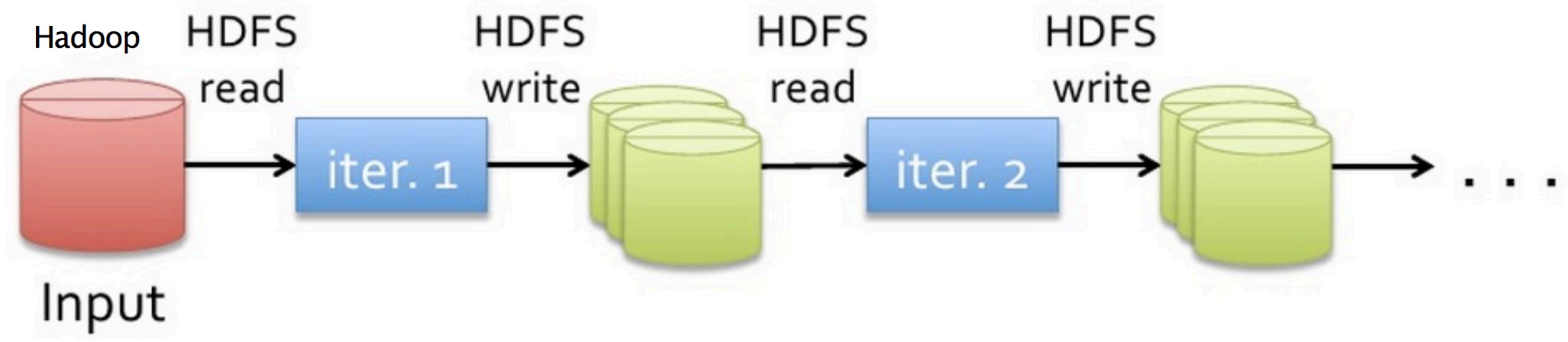


- Cons: Heavy latency occurs while repeatedly accessing dataset from storage.

Hadoop suffers from I/O overhead



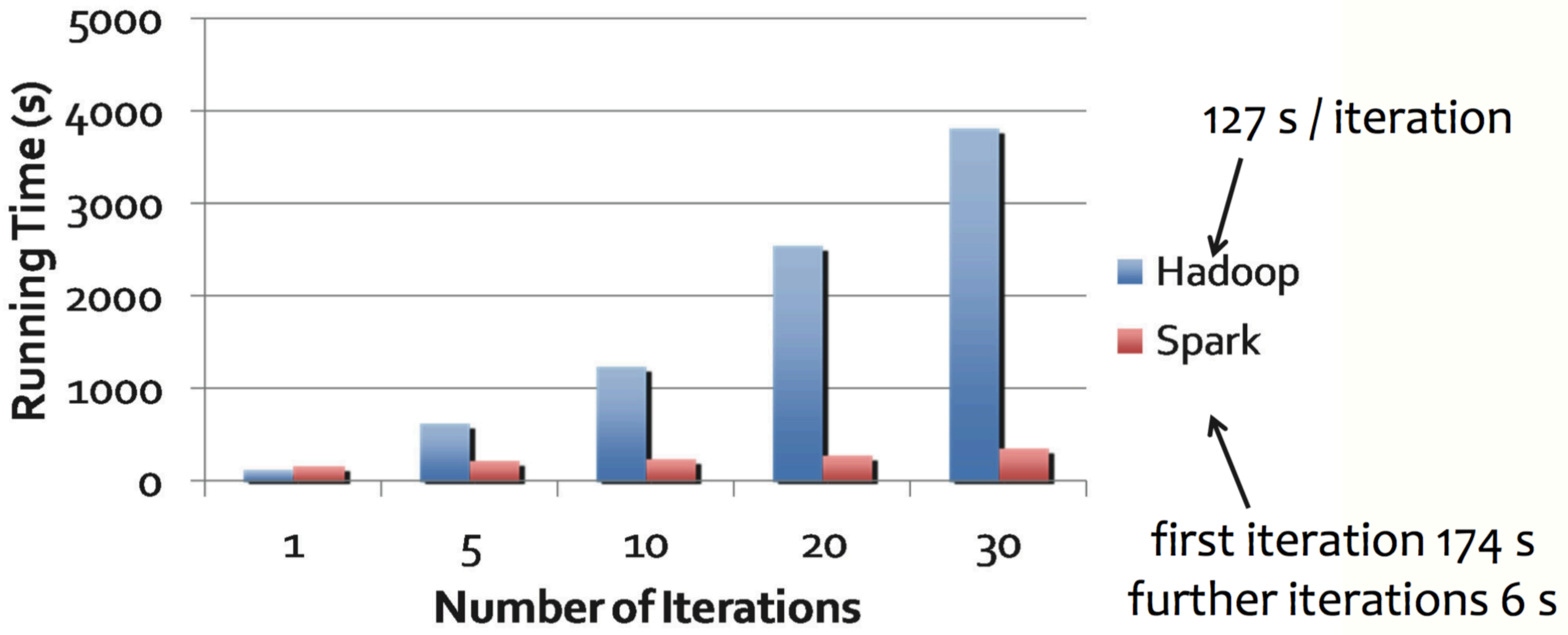
Vs



Source : http://www.slideshare.net/MrChrisJohnson/collaborative-filtering-with-spark

Spark performance

- Logisitic Regression performance



Source : http://www.mosharaf.com/wp-content/uploads/mosharaf-spark-bc-poster-spring10.pdf

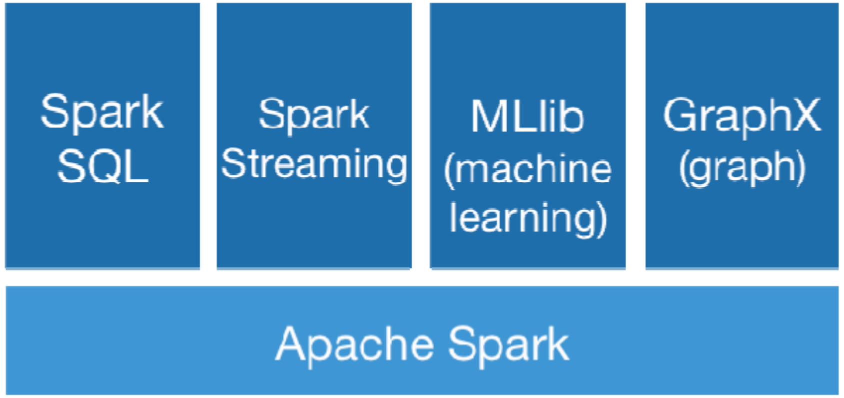
Apache Spark

- Fast and general engine for large-scale data processing engine.
- Running on Apache Hadoop, Mesos, and Amazon EC2.

- Main Features
  - General execution graphs
    - Low latency parallel scheduler that achieves high locality
  - Distributed dataset and memory abstractions
    - Based on Resilient Distributed Dataset(RDD)
    - Aggressive memory use
  - Support Scala / Java / Python

Apache Spark

- Aim to generality



Source: Apache Spark

- SparkR
- Apache Mahout
- Zeppelin

- Vigorous update

Version	Original release date	Latest version	Release date
0.5	2012-06-12	0.5.1	2012-10-07
0.6	2012-10-14	0.6.1	2012-11-16
0.7	2013-02-27	0.7.3	2013-07-16
0.8	2013-09-25	0.8.1	2013-12-19
0.9	2014-02-02	0.9.2	2014-07-23
1	2014-05-30	1.0.2	2014-08-05
1.1	2014-09-11	1.1.1	2014-11-26
1.2	2014-12-18	1.2.2	2015-04-17
1.3	2015-03-13	1.3.1	2015-04-17
1.4	2015-06-11	1.4.1	2015-07-15
1.5	2015-09-09	1.5.2	2015-11-09
1.6	2016-01-04	1.6.1	2016-03-09
2	2016	2.0.0	2016

Source: wikipedia

Resilient Distributed Dataset (RDD)

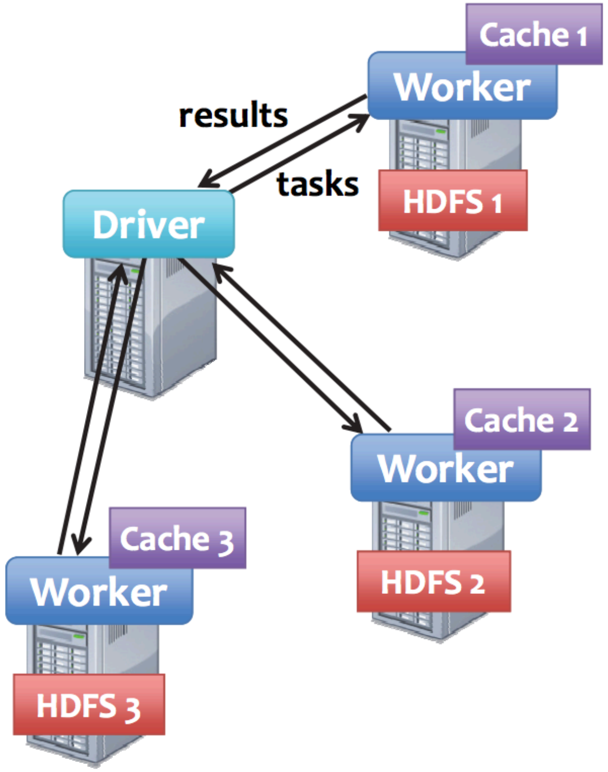
- is The alpha and Omega of Apache Spark
- is a Read-only collection of objects partitioned across a set of machines
  - Low maintenance cost
- is Created by transforming data in storage using data flow operators
- is Managed by 'lineage'
  - Contains information about how it can derived from parent RDD.
  - Can be rebuilt if a partition is lost.
- Can be cached for following parallel operations.

RDD example

- Goal : Counting lines containing “ERROR” in log file

- file = spark.textFile(“hdfs://...”)
- errors = file.filter(_.contains(“ERROR”))
- cachedErrs = errors.cache()

- cachedErrs.filter(_.contains(“foo”)).count
- cachedErrs.filter(_.contains(“bar”)).count



Caching reusable datasets ::
Much Faster running time !!

RDD operations

RDD transformation

- map()
- filter()
- sample()
- intersection()
- repartition()

…

RDD to other RDD

RDD actions

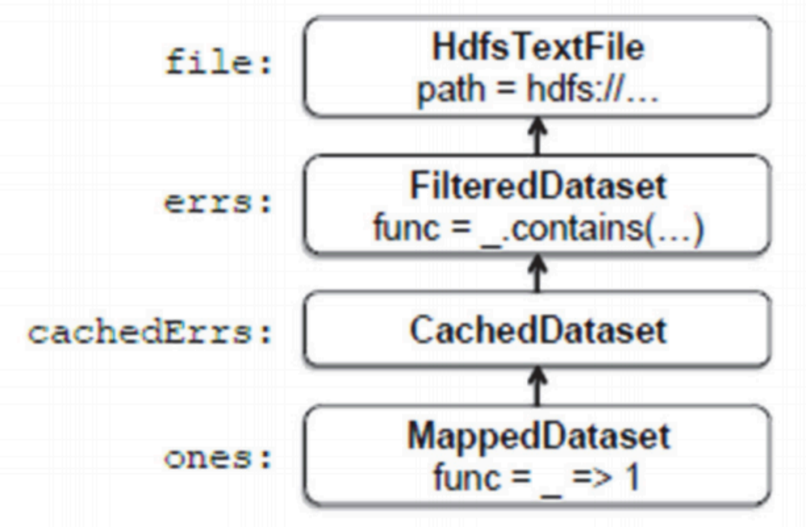
- reduce()
- collect()
- count()
- saveAsFile()
- foreach()

…

Actually produce output and return final value to the driver program (master)

Lineage chain for RDD

```
file = spark.textFile("hdfs://...")
errors = file.filter(_.contains("ERROR"))
cachedErrs = errors.cache()
ones = cachedErrs.map(_ => 1)
count = ones.reduce(_+_)
```



- Each RDD contains a pointer to its parent RDD and information about how the parent was transformed.
- Guarantees Fault tolerance

Lazyness Execution

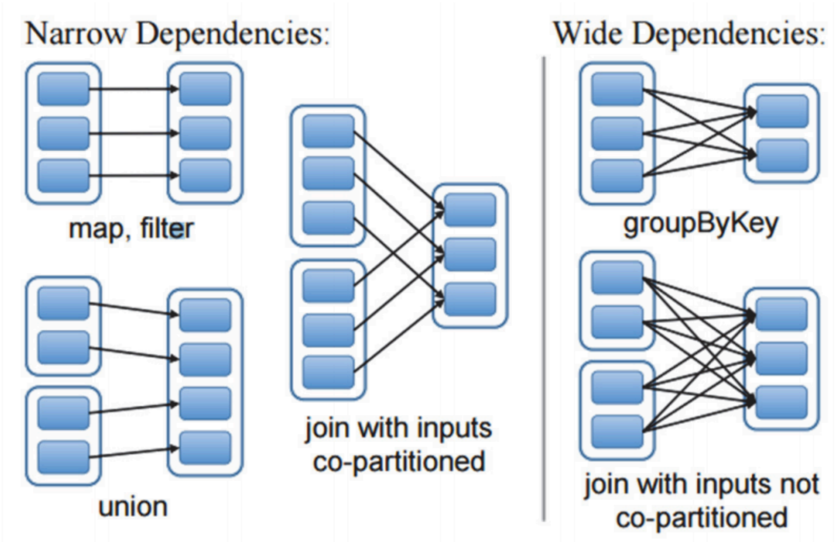
- Lazyness

`file = spark.textFile(“hdfs://...”)`	
`errors = file.filter(_.contains(“ERROR”))`	: never materialized (do nothing)
`ones = cachedErrs.map(_ => 1)`	: never materialized (do nothing)
`count = ones.reduce(_+_)`	: Each worker node scans input blocks in a streaming manner to evaluate ones and count

- RDD is never materialized until parallel actions are executed.
- Effect of reducing the # of passes it has to take over the data by grouping operations together : Chaining multiple operation on data becomes much more easier
- RDD is discarded from memory after use.

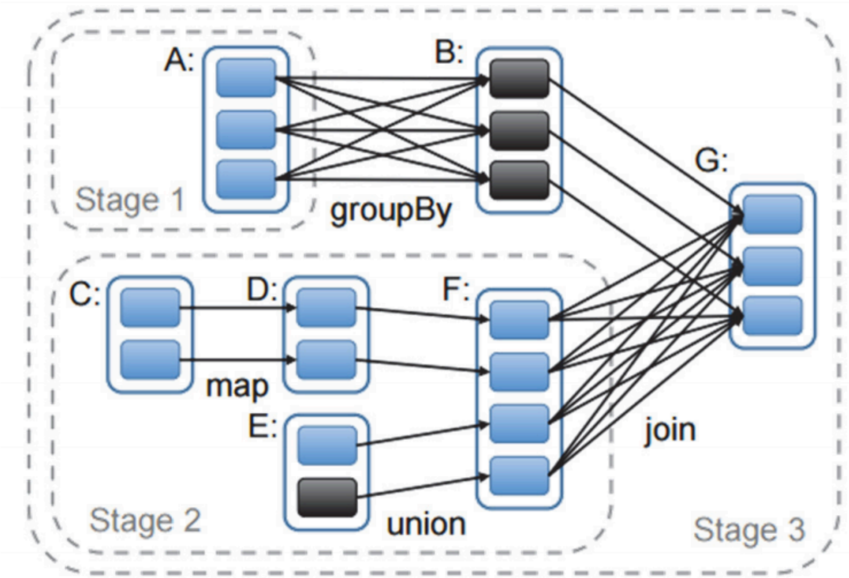
RDD operations

- Dependencies



- Narrow: fast, local processing
- Wide: communication over network

- Job stages



  - Black box: already in memory

- To run an action on G, only the stage 2 and 3 need to be performed.

Caching

- cache() and persist() method
- Memory / Storage caching
- RDD is cached after first action
- If not enough memory – cache as much as possible
- Future actions are performed on cached partitions
- Useful in iterative algorithms

Level	Space	CPU	In Memory	On disk
MEMORY_ONLY	High	Low	Y	N
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK	High	Medium	Some	Some
MEMORY_AND_DISK_SER	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

Source : http://dataottam.com/2015/12/22/what-is-the-role-of-rdds-in-apache-spark-part-1/

Machine Learning in ApacheSpark

- Machine Leaning library (MLlib)
- Collaborative Fillltering(CF)
- Alternative Least Square (ALS)

Machine Learning Library

- Classfication and Regression
  - logistic regression, support vector machine(SVM), naive bayse, decision tree, generalized linear regression, etc
- Clustering
  - latent drichlet allocation (LDA), k-means, gaussian mixture, etc
- Demensionality reduction
  - singular vector decomposition (SVD), principle comnent ananlysis (PCA)
- Collabarative filtering
  - alternative least square (ALS)
- Optimization
  - stochastic gradient descent, etc
- Etc

Collaborative Filltering

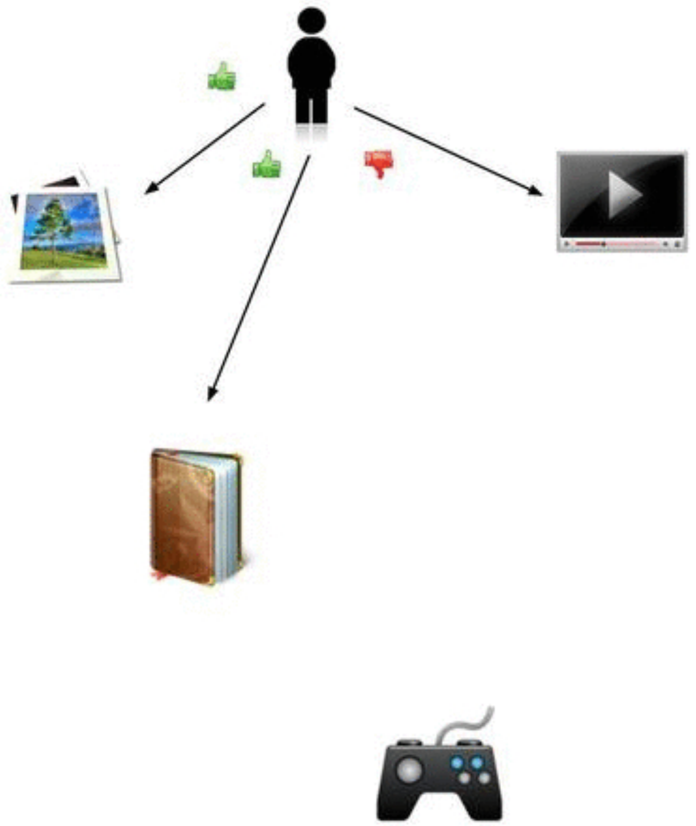
- is a technique used by some recommender systems.
- is a method of making automatic predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating).

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

Collaborative Filltering Example (Cont)

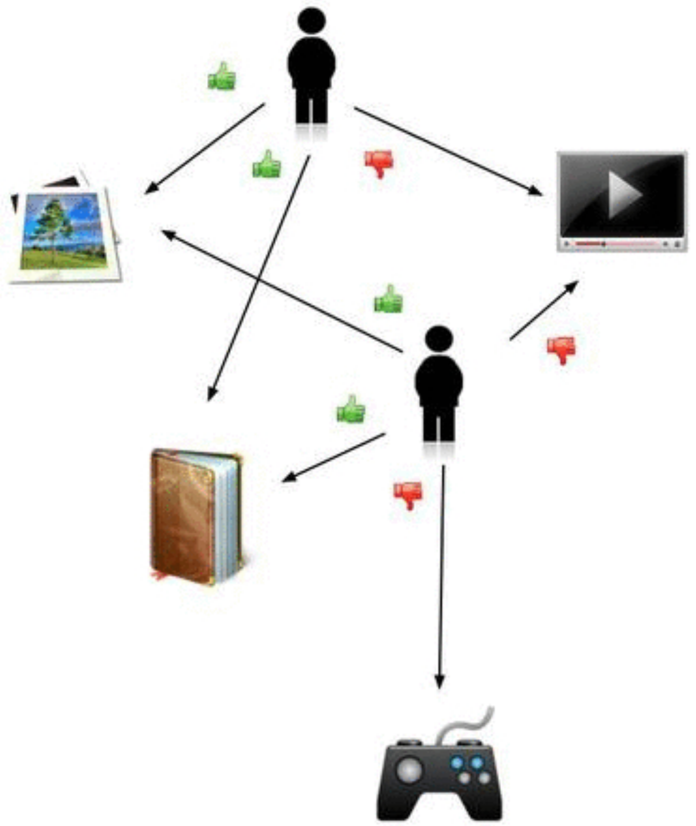
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



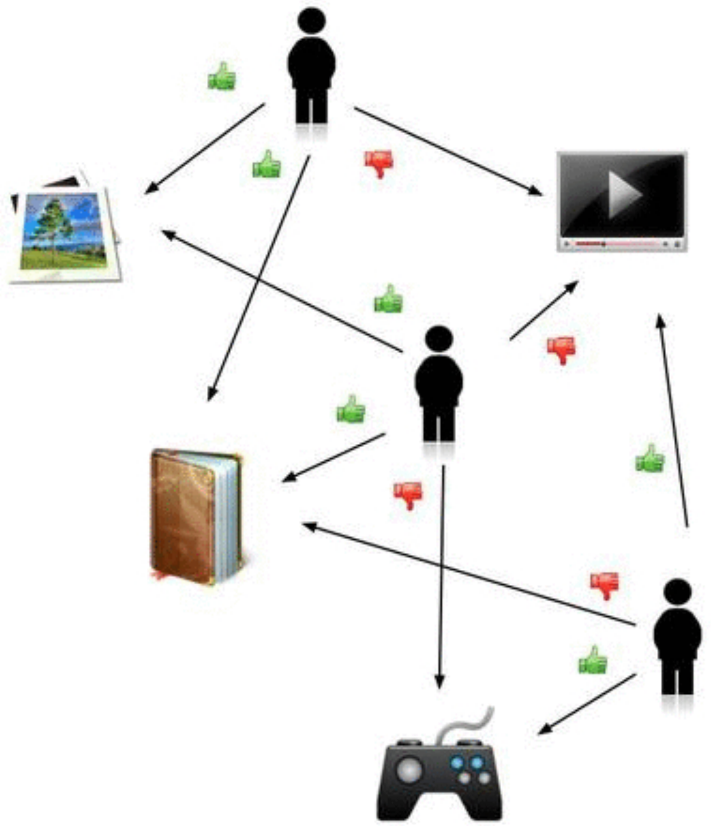
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



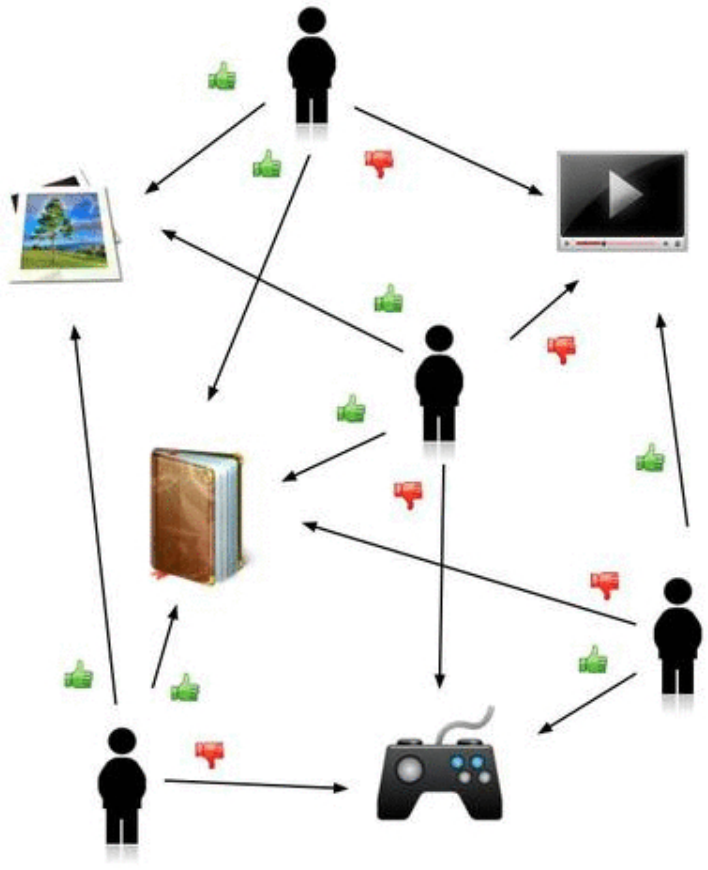
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



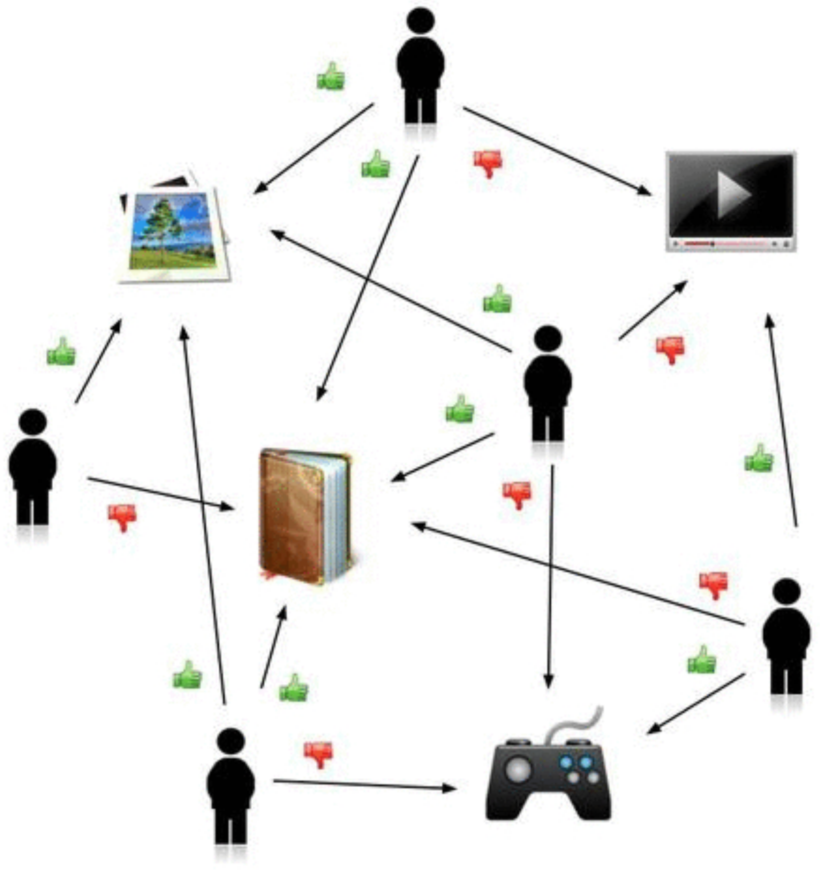
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Fillltering Example (Cont)



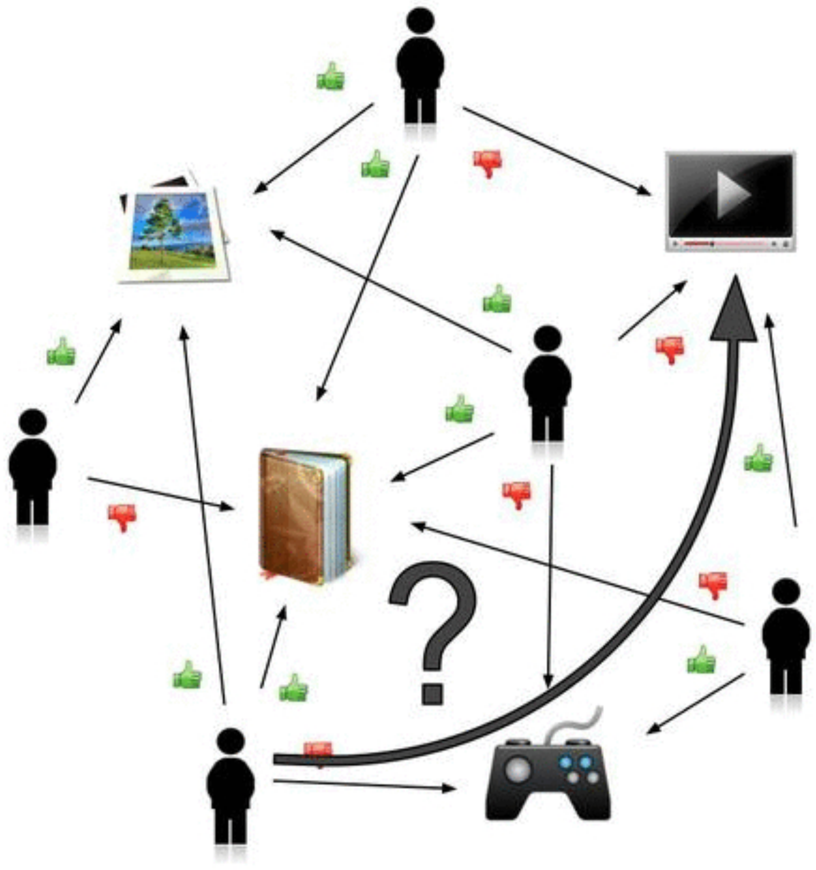
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Fillltering Example (Cont)



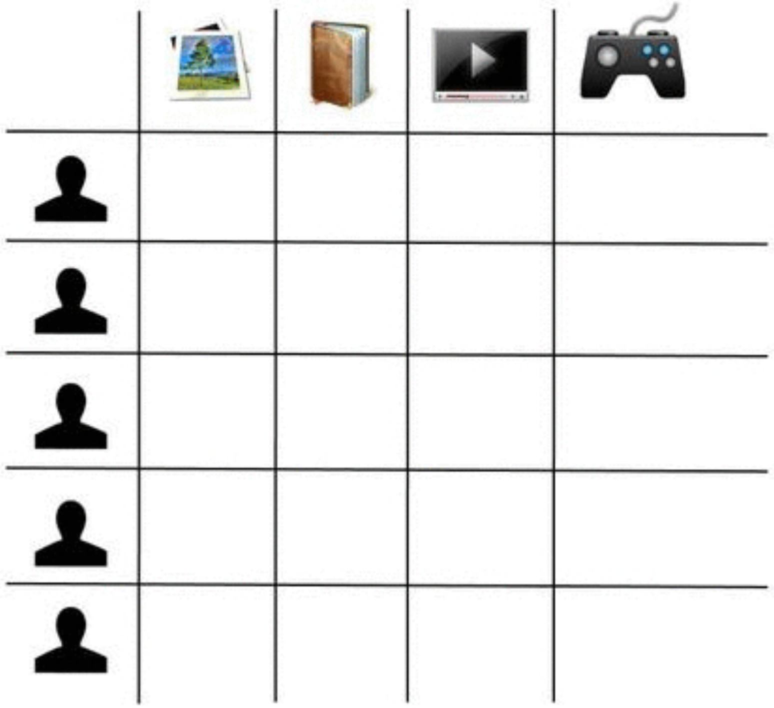
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



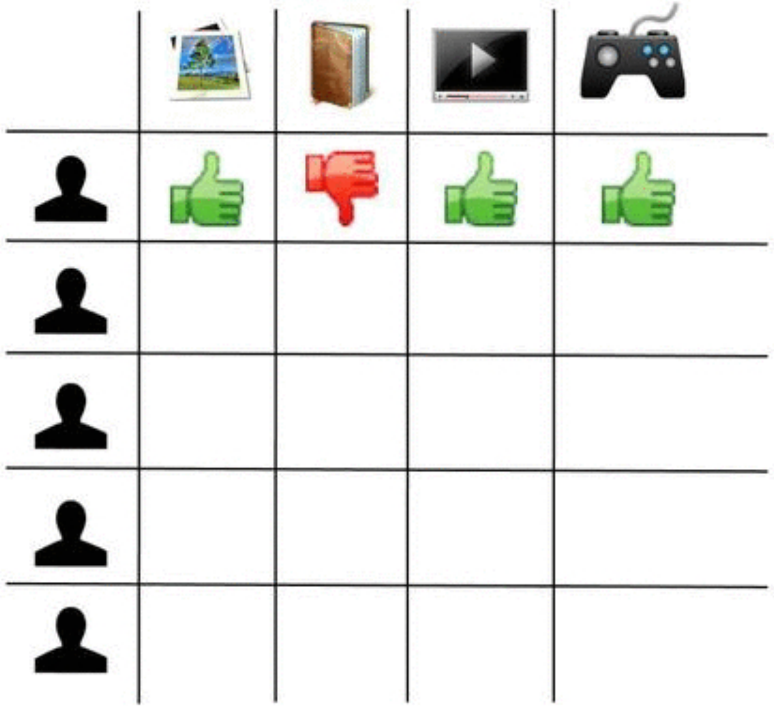
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



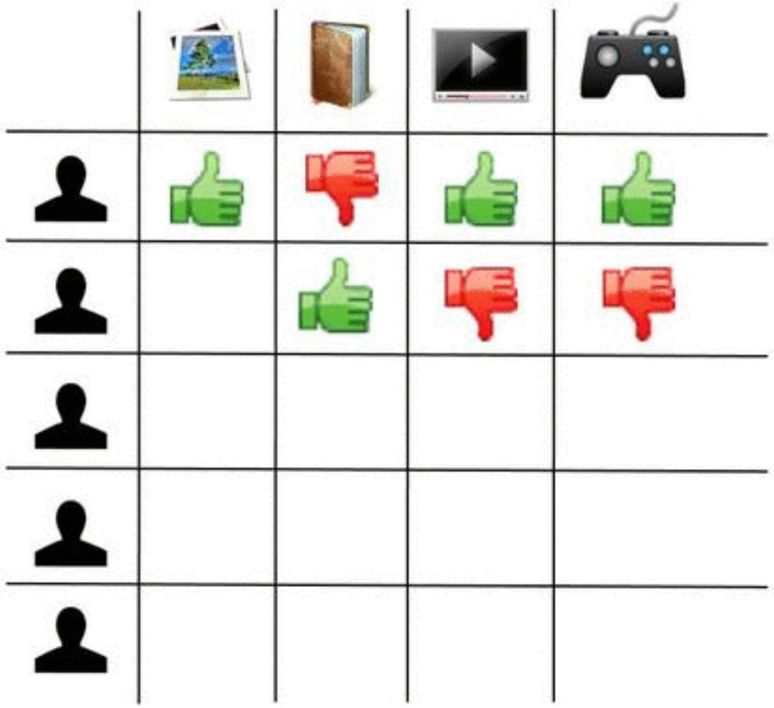
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Fillltering Example (Cont)



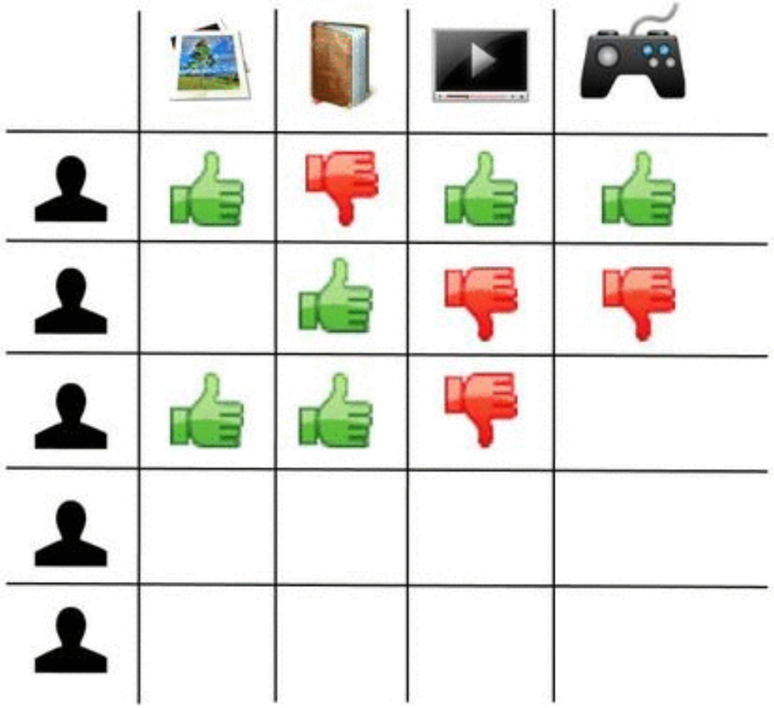
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



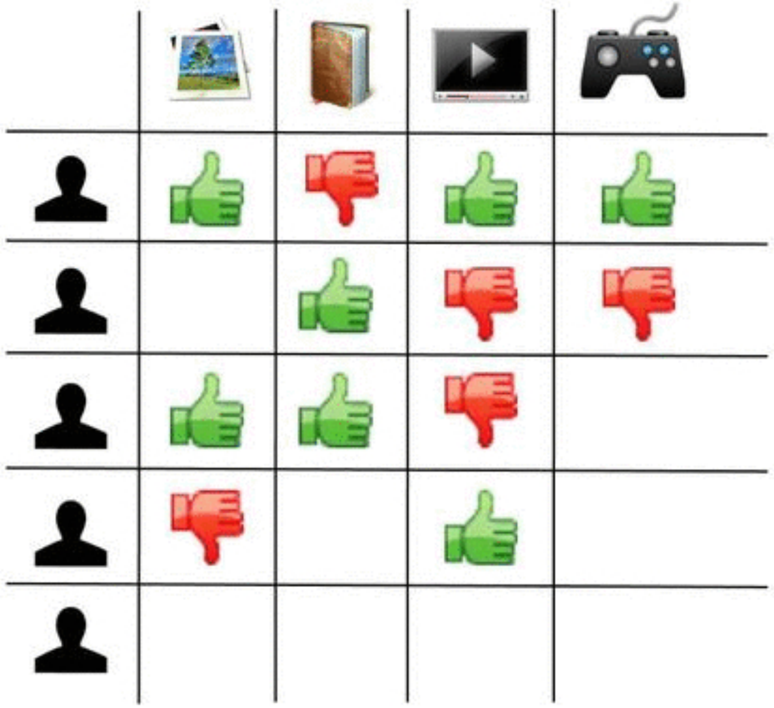
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



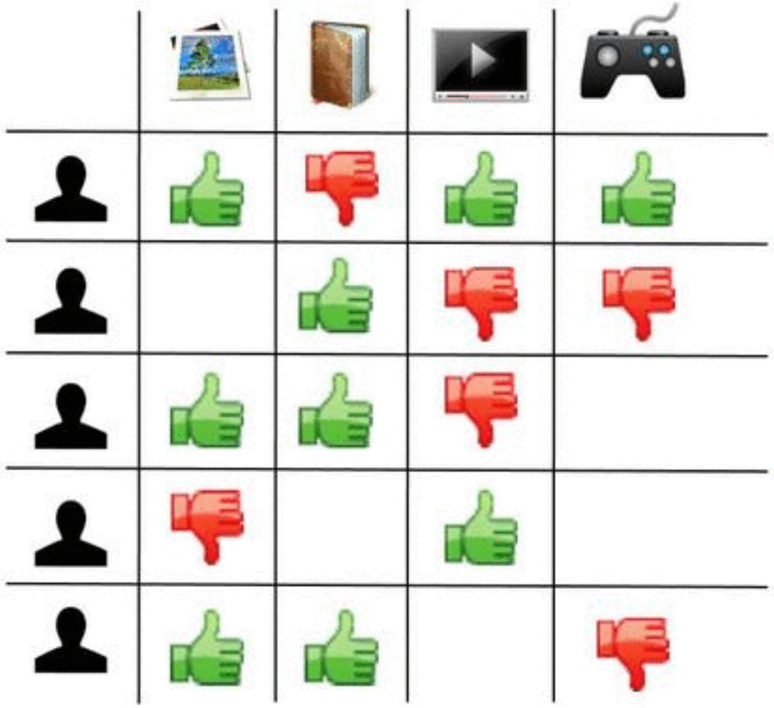
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



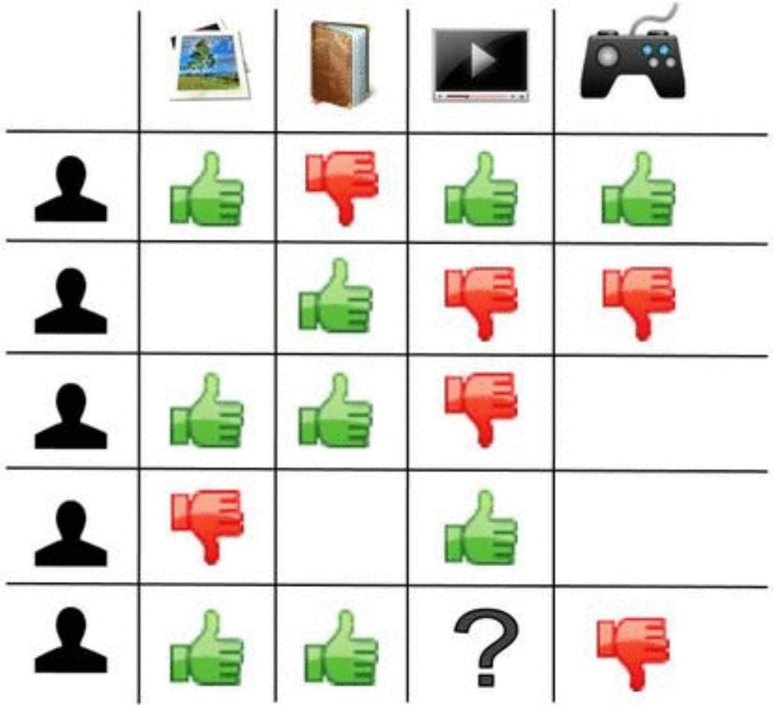
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



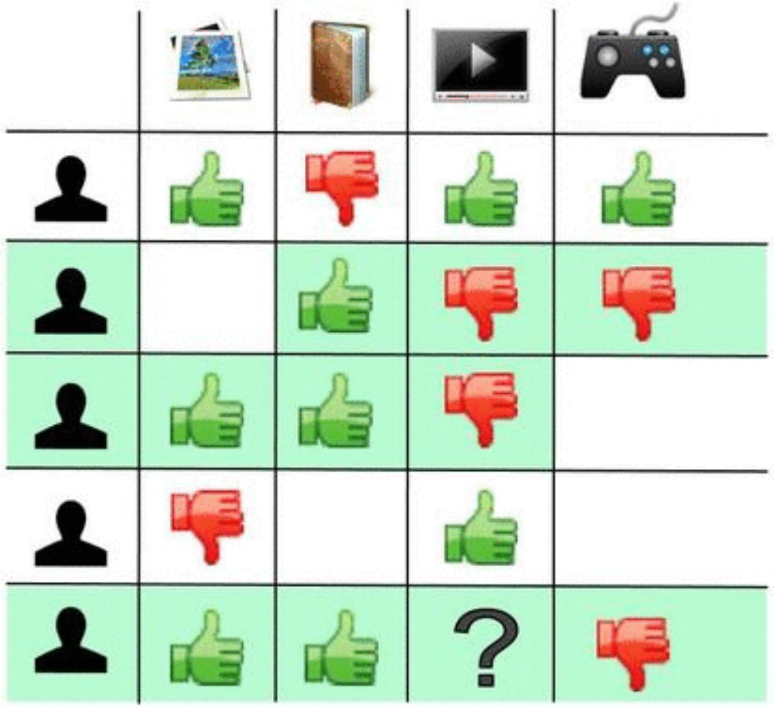
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Fillltering Example (Cont)



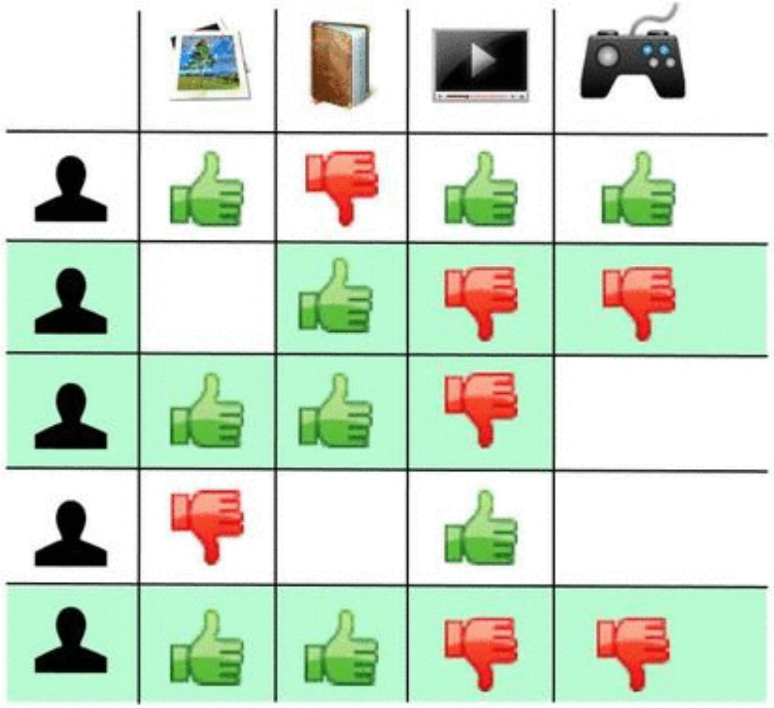
Source : https://en.wikipedia.org/wiki/Collaborative_filtering

Collaborative Filltering Example (Cont)



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

Collaborative Filltering Example (Cont)



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

Collaborative Fillltering Example (Cont)

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

Explicit Matirx Factorization

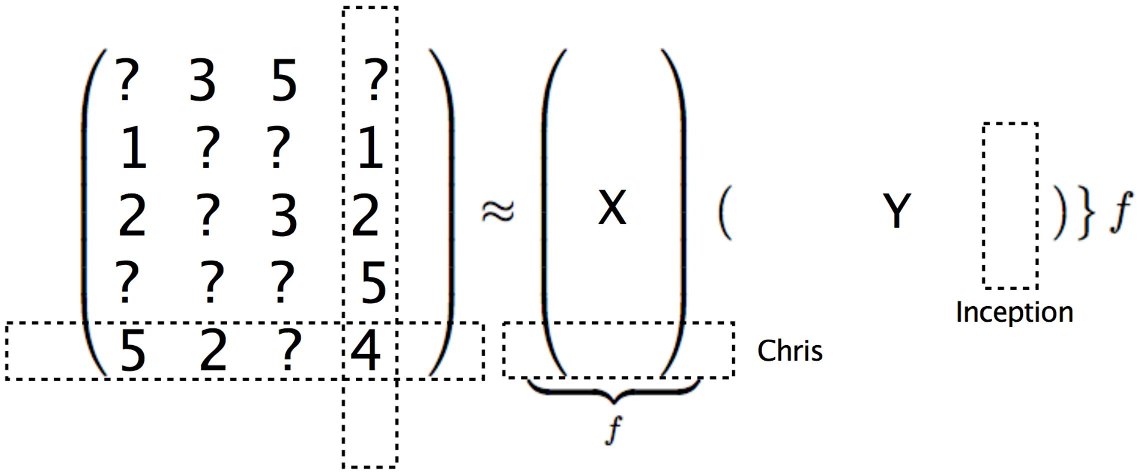
- Users explicitly rate a subset of movie catalog
- Goal : predict how users will rate new movie

$$\begin{array}{cccccc}
 & \multicolumn{4}{c}{\text{Movies}} & \\
 & ? & 3 & 5 & ? & \\
 & 1 & ? & ? & 1 & \\
\text{Users} & 2 & ? & 3 & 2 & \\
 & ? & ? & ? & 5 & \\
 & 5 & 2 & ? & 4 & \text{Chris} \\
 & & & & \text{Inception} &
\end{array}$$

Source : http://www.slideshare.net/MrChrisJohnson/collaborative-filtering-with-spark

Explicit Matrix factorization

- Approximate ratings matrix by product low-dimensional user and movie matrices
- Minimize RMSE (root mean square error)



$$min_{x,y} \sum_{u,i}(r_{ui} - x_u^T y_i - b_u - b_i)^2 - \lambda \left(\sum_u ||x_u||^2 + \sum_i ||y_i||^2 \right)$$

- r_{ui} = user $u's$ ratings for movie i
- x_u = user $u's$ latent factor vector
- x_i = item $i's$ latent factor vector
- b_u = bias for user u
- b_i = bias for item i
- λ = regulization parameter

Source : http://www.slideshare.net/MrChrisJohnson/collaborative-filtering-with-spark

Implicit Matrix Factorization

- Replace stream counts with binary bites
  - 1 = streamed, 0 = never streamed
- Minimise weighted RMSE (root mean square error) using a function of stream counts as weights

$$\begin{pmatrix}1&0&0&0&1&0&0&1\\0&0&1&0&0&1&0&0\\1&0&1&0&0&0&1&1\\0&1&0&0&0&1&0&0\\0&0&1&0&0&1&0&0\\1&0&0&0&1&0&0&1\end{pmatrix} \approx \underbrace{\begin{pmatrix} \\ \\ X \\ \\ \\ \end{pmatrix}}_{f} \left(\qquad Y \qquad \right)\Big\} f$$

$$min_{x,y} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i - b_u - b_i)^2 - \lambda \left(\sum_u ||x_u||^2 + \sum_i ||y_i||^2 \right)$$

- $p_{ui} = 1$ if user u streamed track i else 0
- $c_{ui} = 1 + ar_{ui}$
- $x_u =$ user $u's$ latent factor vector
- $x_i =$ i item $i's$ latent factor vector
- $b_u =$ bias for user u
- $b_i =$ bias for item i
- $\lambda =$ regulization parameter

Source : http://www.slideshare.net/MrChrisJohnson/collaborative-filtering-with-spark

Alternative Least Squares – example(1)

```scala
package org.apache.spark.examples.mllib
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.MatrixFactorizationModel
import org.apache.spark.mllib.recommendation.Rating

object RecommendationExample {
    def main(args: Array[String]): Unit = {
        val conf = new SparkConf().setAppName("CollaborativeFilteringExample")
        val sc = new SparkContext(conf)

        // Load and parse the data
        val data = sc.textFile("data/mllib/als/test.data")
        val ratings = data.map(_.split(',') match { case Array(user, item, rate) =>
            Rating(user.toInt, item.toInt, rate.toDouble)
        })
```

```
1,1,5.0
1,2,1.0
1,3,5.0
1,4,1.0
2,1,5.0
2,2,1.0
2,3,5.0
2,4,1.0
3,1,1.0
3,2,5.0
3,3,1.0
3,4,5.0
4,1,1.0
4,2,5.0
4,3,1.0
4,4,5.0
```

Input data(user, product, rating)

Source : http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html#collaborative-filtering

Alternative Least Squares – example(2)

```
// Build the recommendation model using ALS
val rank = 10
val numIterations = 10
val model = ALS.train(ratings, rank, numIterations, 0.01)

// Evaluate the model on rating data
val usersProducts = ratings.map { case Rating(user, product, rate) =>
    (user, product)
}
val predictions =
    model.predict(usersProducts).map { case Rating(user, product, rate) =>
        ((user, product), rate)
    }
val ratesAndPreds = ratings.map { case Rating(user, product, rate) =>
    ((user, product), rate)
}.join(predictions)
val MSE = ratesAndPreds.map { case ((user, product), (r1, r2)) =>
    val err = (r1 - r2)
    err * err
}.mean()
println("Mean Squared Error = " + MSE)
```

Source : http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html#collaborative-filtering

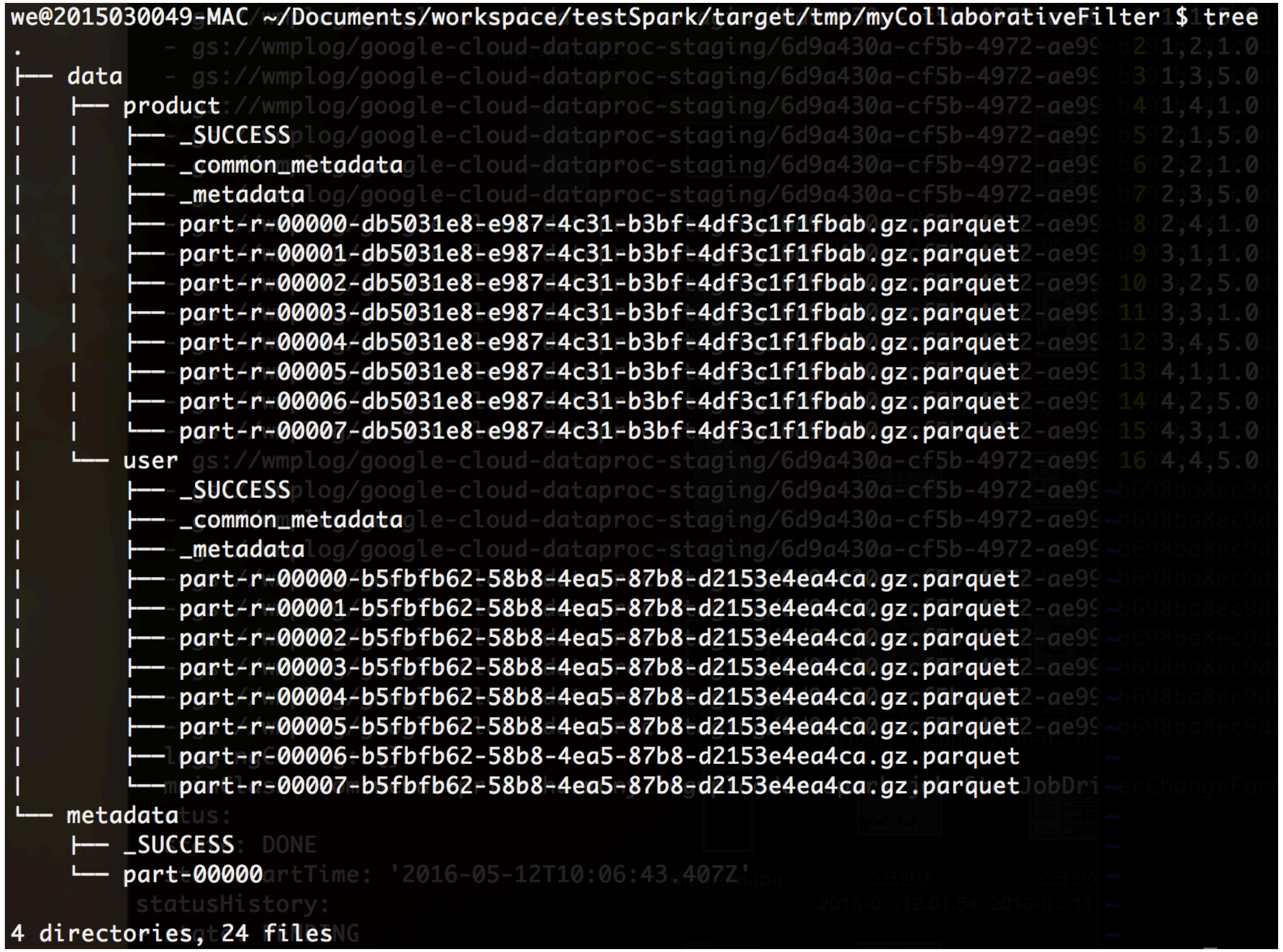
Alternative Least Squares – example(3)

```
        // Save and load model
        model.save(sc, "target/tmp/myCollaborativeFilter")
        val sameModel = MatrixFactorizationModel.load(sc, "target/tmp/myCollaborativeFilter")
    }
}
```

Source : http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html#collaborative-filtering

Alternative Least Squares – example(4)

- result RMSE : Mean Squared Error = 4.977986740610271E-6
- result Model



Appendix

- Spark cluster modes

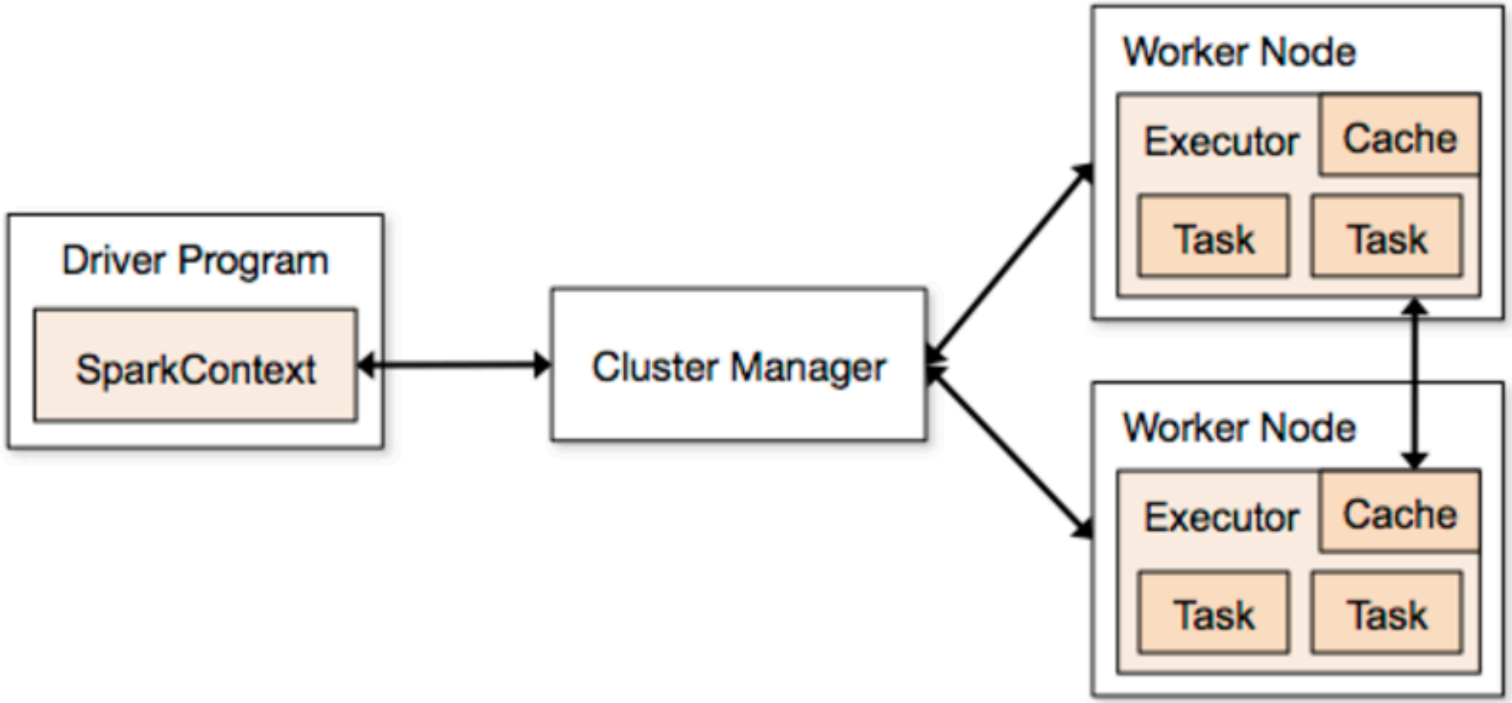
Spark cluster modes

- On stand alone cluster
  - a simple cluster manager included with Spark that makes it easy to set up a cluster

- On Hadoop Yarn cluster
  - the resource manager in Hadoop 2

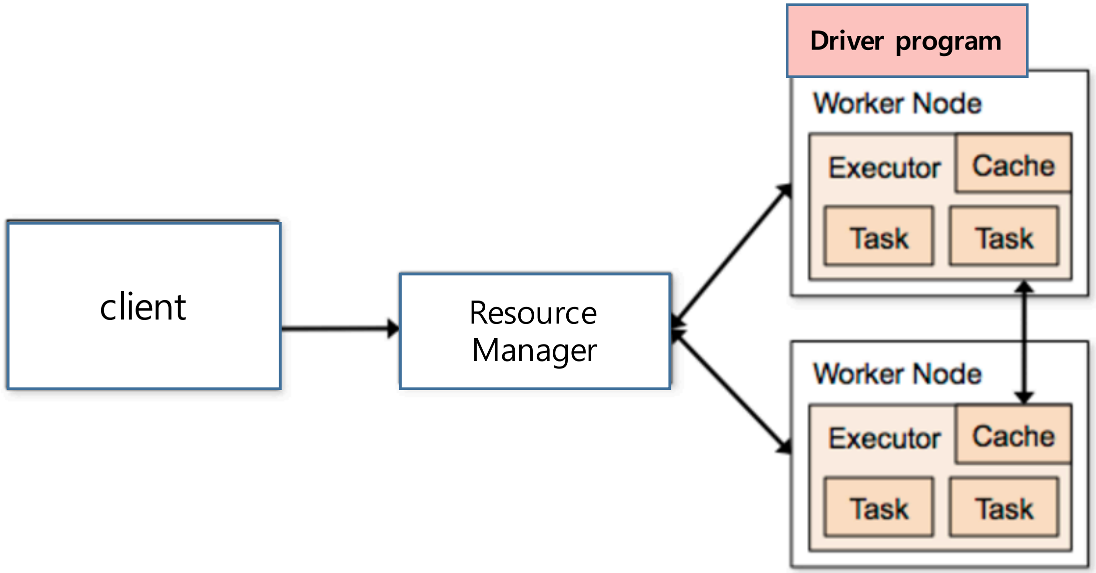
- On Mesos cluster
  - a general cluster manager that can also run Hadoop MapReduce and service applications

Stand alone mode

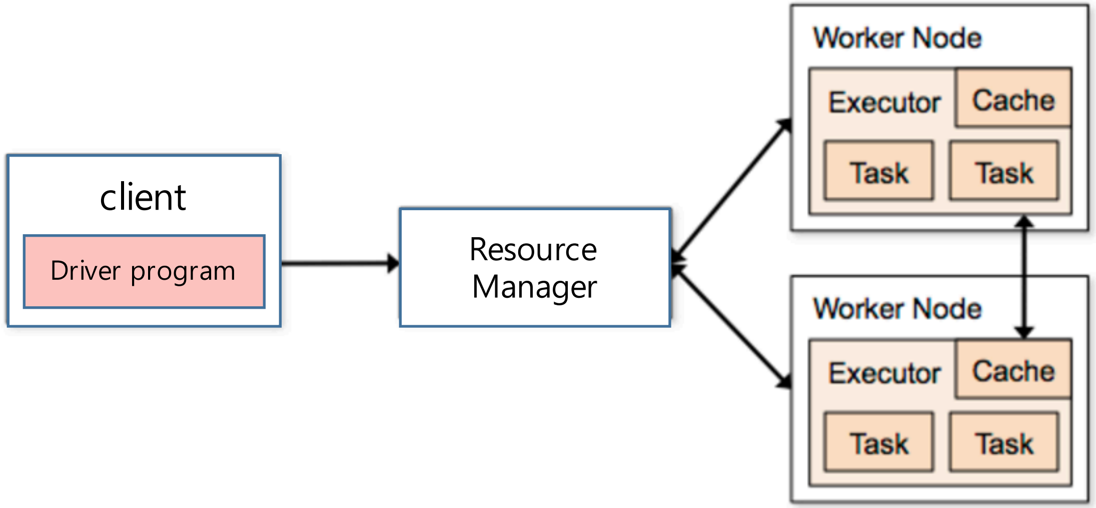


Hadoop Yarn mode

- 2 modes
  - Cluster mode

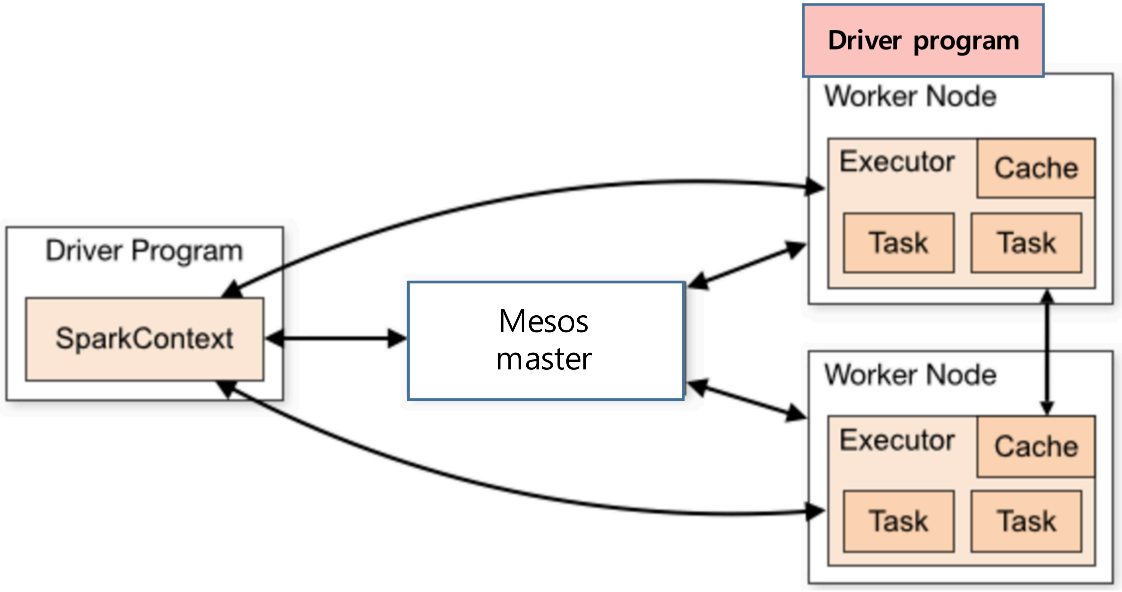


  - Client mode



Mesos mode

- 2 modes
  - Cluster mode
    - a Spark Mesos framework is launched directly on the client machine and waits for the driver output



  - Client mode
    - the client can find the results of the driver from the Mesos Web UI