Introduction to Spark

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the '**framework of spark**', Resilient Distributed Datasets (RDDs) and also discuss some of its applications such as: **Page rank** and **GraphX.**

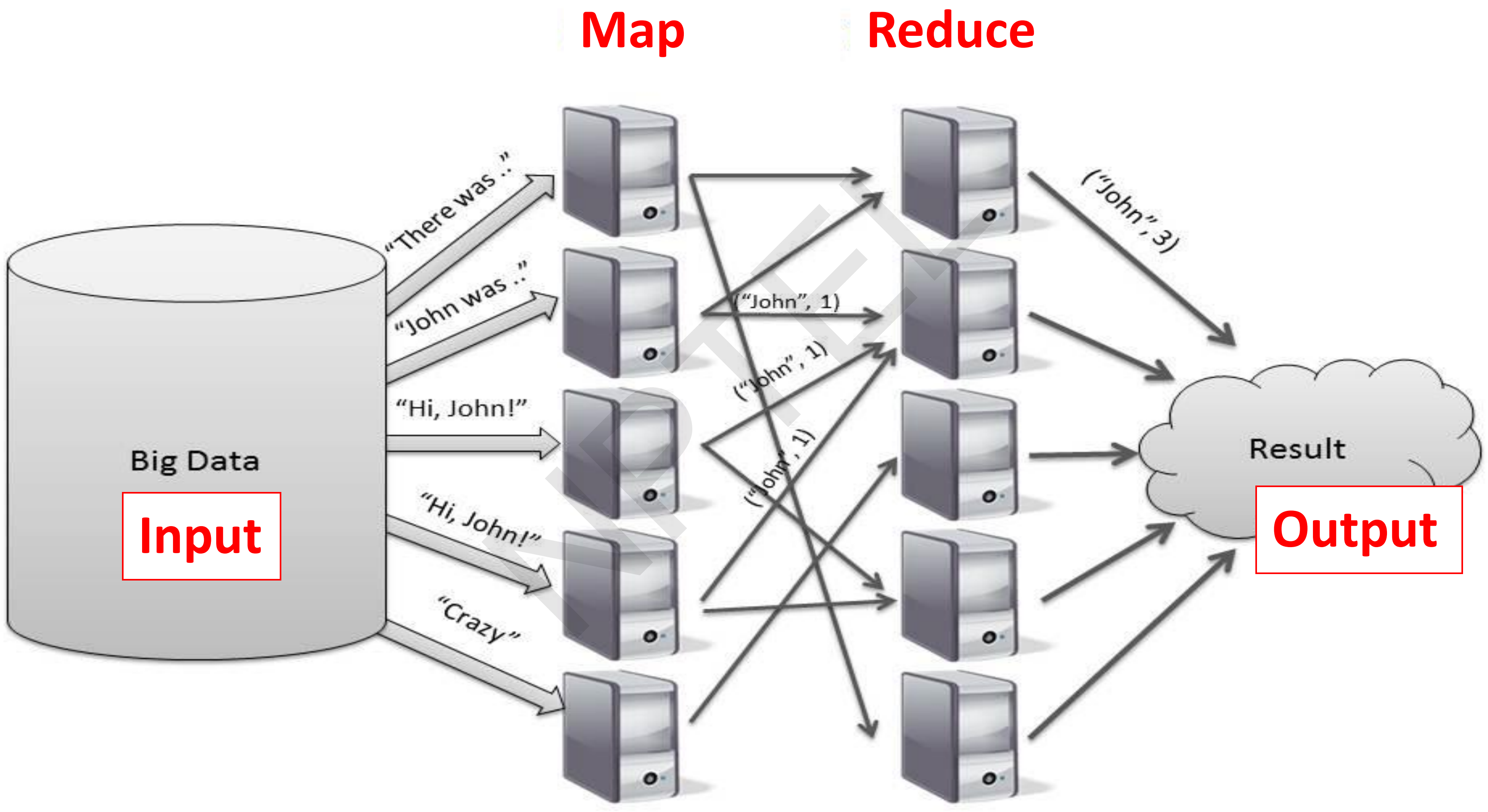
Need of Spark

- **Apache Spark** is a big data analytics framework that was originally developed at the University of California, Berkeley's AMPLab, in 2012. Since then, it has gained a lot of attraction both in academia and in industry.

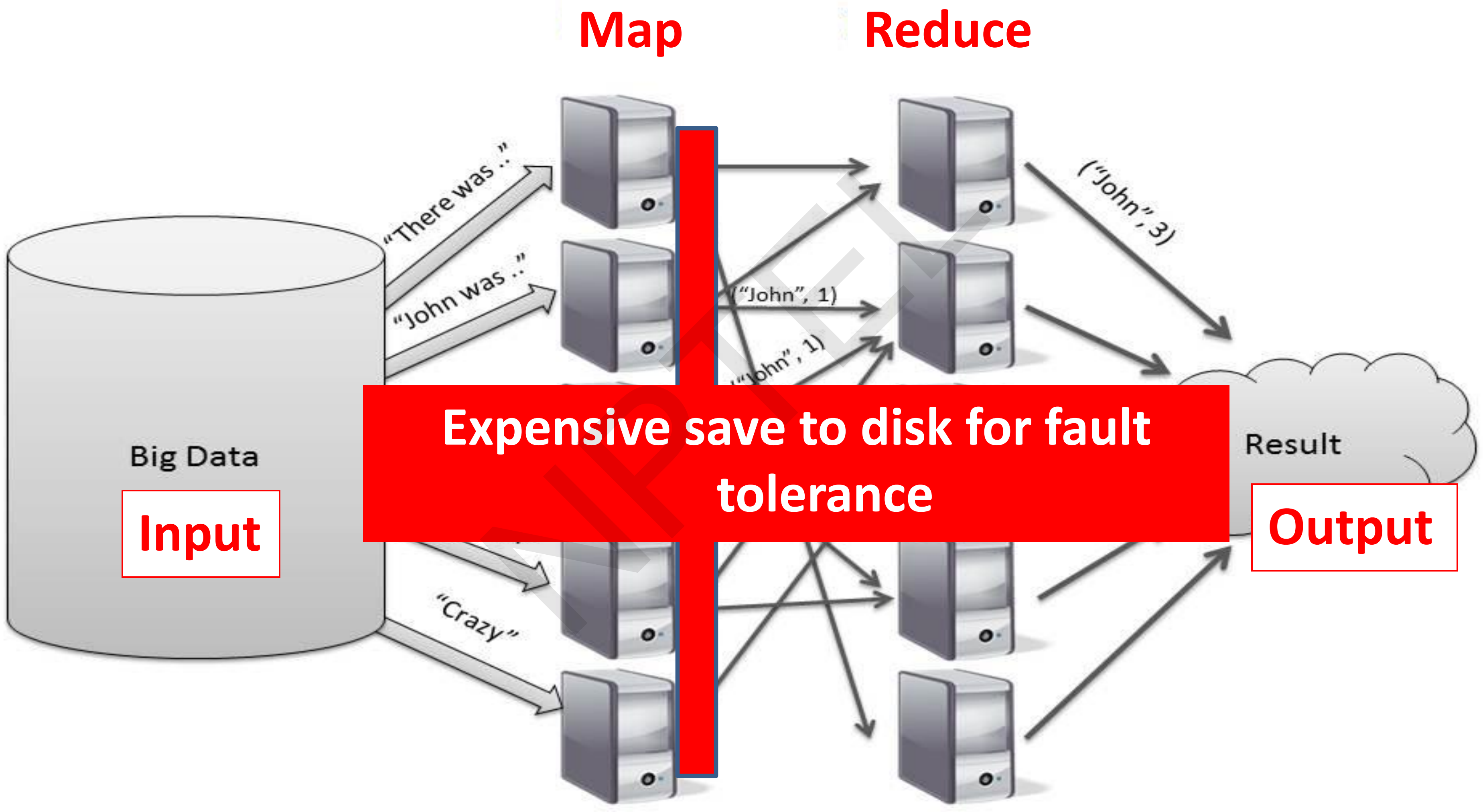
- It is an another system for big data analytics

- **Isn't MapReduce good enough?**
  - **Simplifies batch processing on large commodity clusters**

Need of Spark



Need of Spark



Expensive save to disk for fault tolerance

Need of Spark

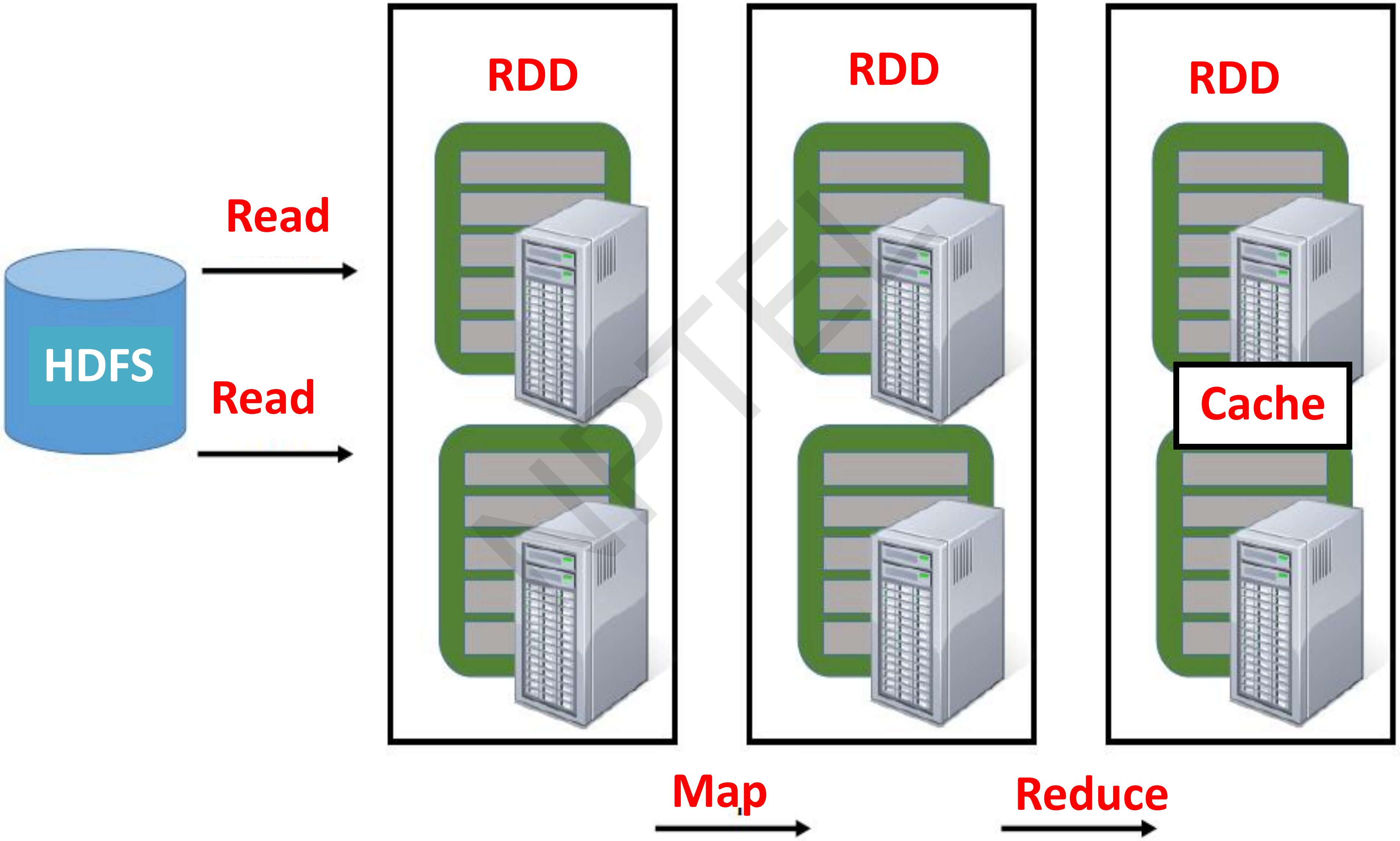
- MapReduce can be expensive for some applications e.g.,
    - **Iterative**
    - **Interactive**
- Lacks efficient data sharing
- Specialized frameworks did evolve for different programming models
    - **Bulk Synchronous Processing (Pregel)**
    - **Iterative MapReduce (Hadoop) ….**

Solution: Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs)

- Immutable, partitioned collection of records
- Built through coarse grained transformations (map, join …)
- Can be cached for efficient reuse

Need of Spark



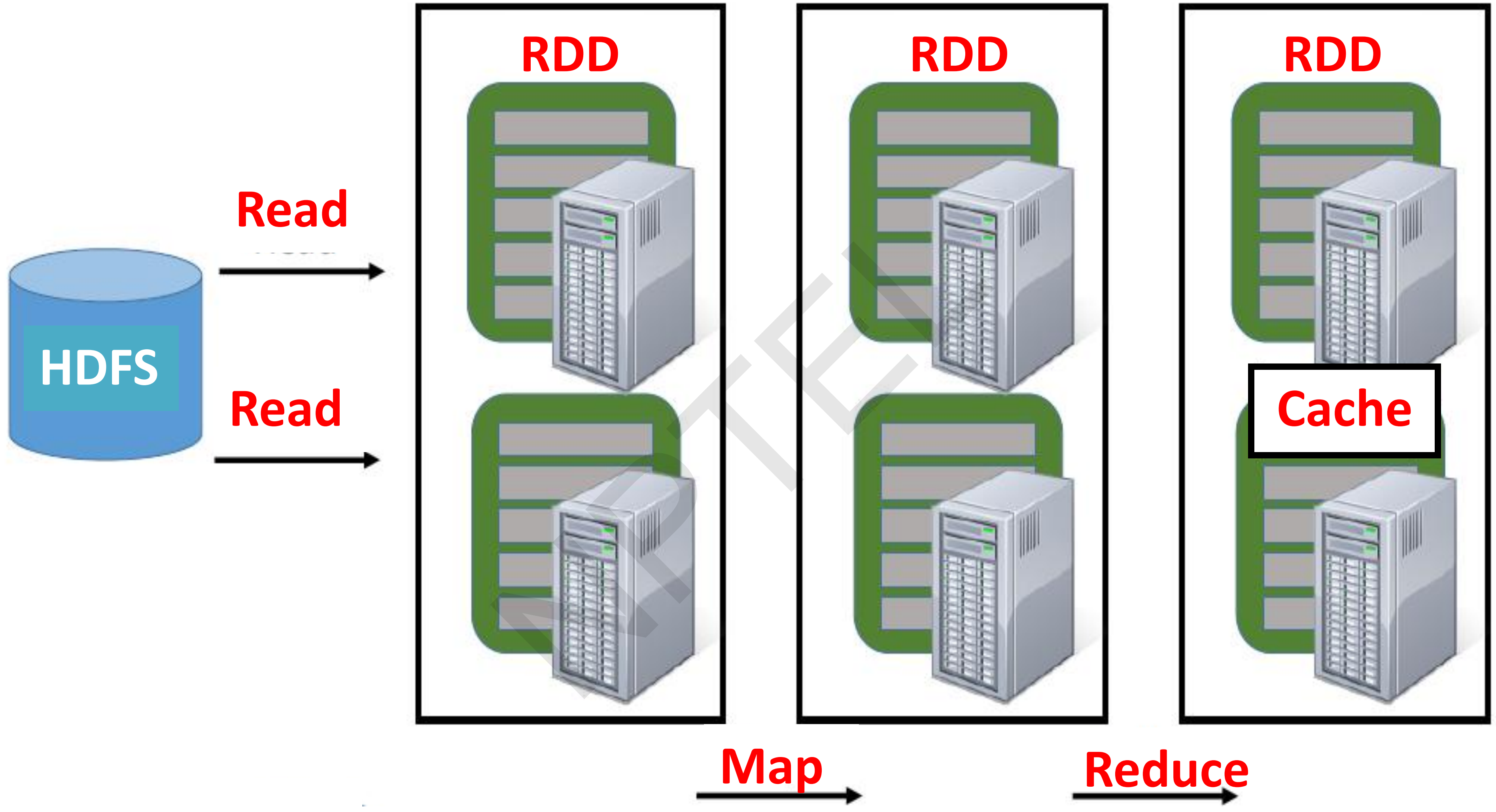
Solution: Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs)

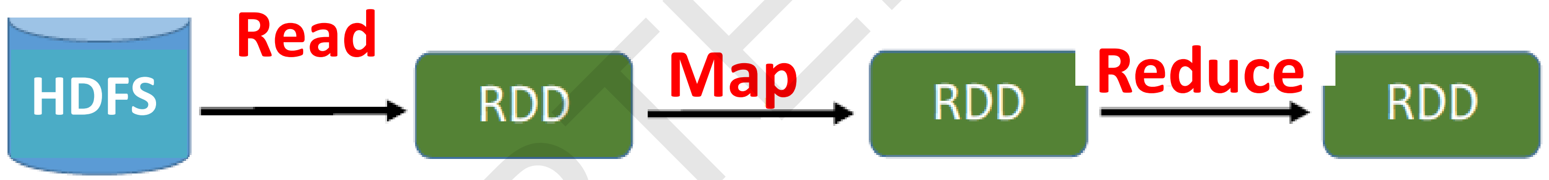
- Immutable, partitioned collection of records
- Built through coarse grained transformations (map, join …)

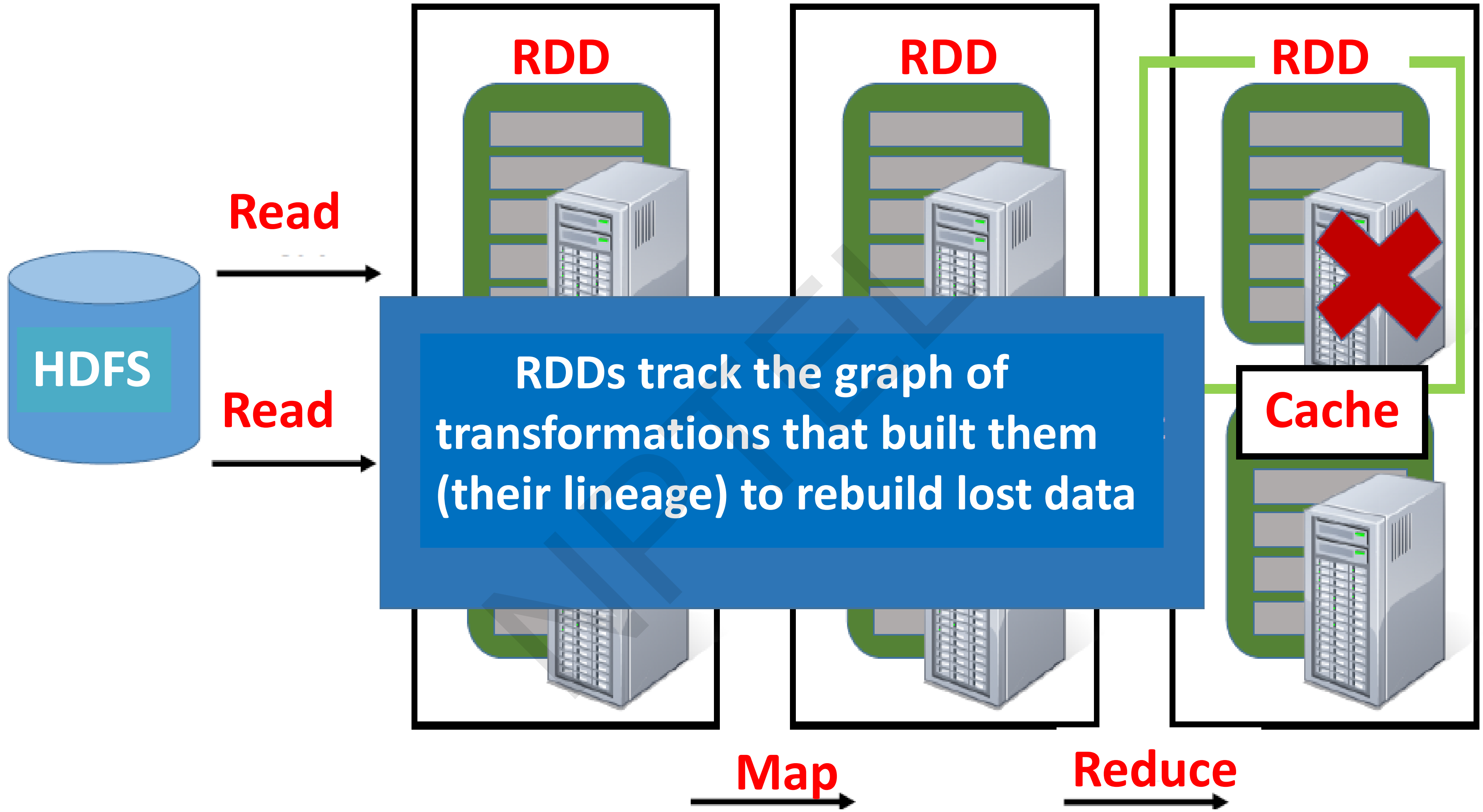
Fault Recovery?

- Lineage!
  - Log the coarse grained operation applied to a partitioned dataset
  - Simply recompute the lost partition if failure occurs!
  - No cost if no failure


RDD
RDD
RDD
Read
HDFS
Read
Cache
Map
Reduce

Lineage



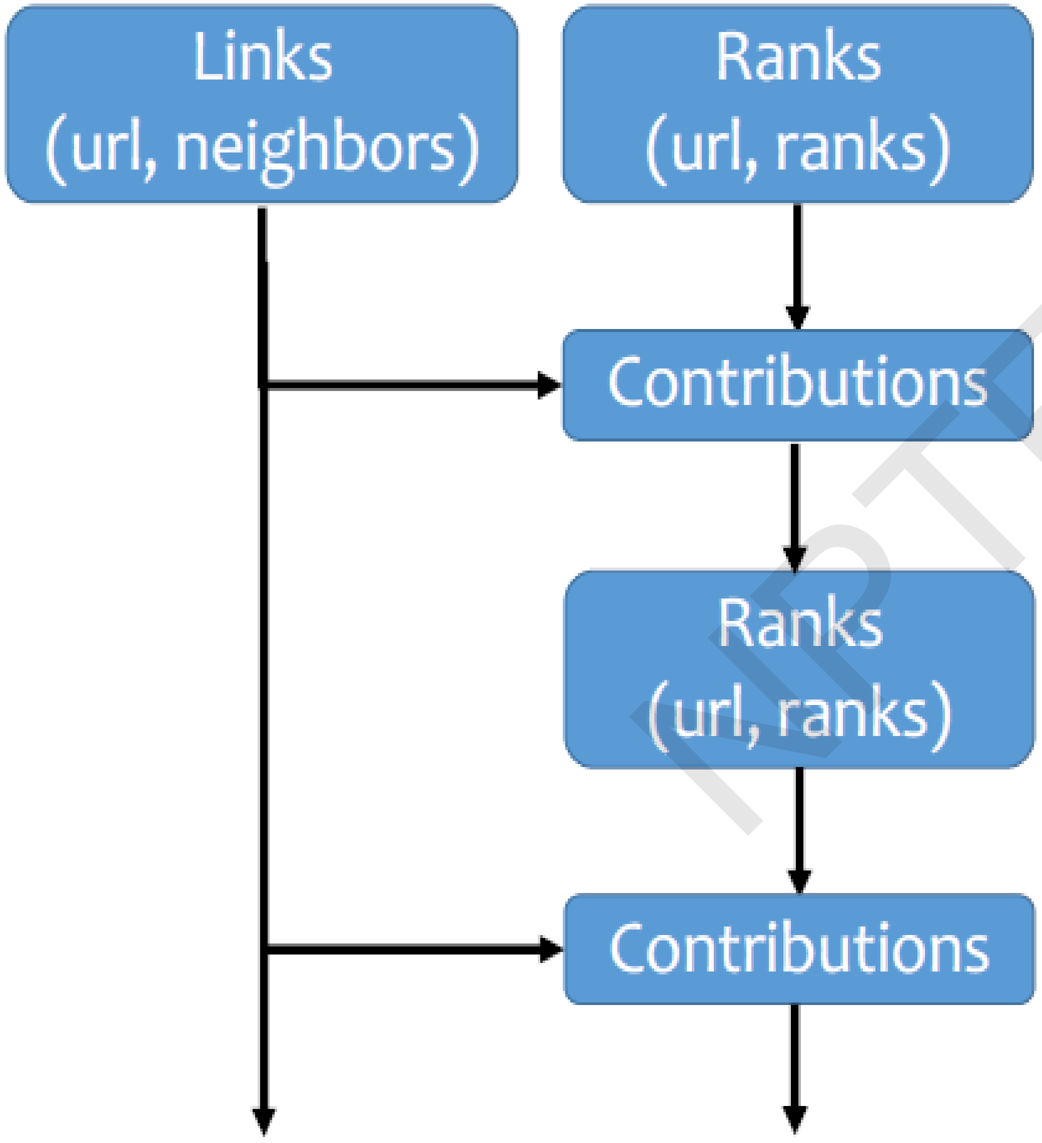


RDDs track the graph of transformations that built them (their lineage) to rebuild lost data

What can you do with Spark?

- **RDD operations**
  - Transformations e.g., filter, join, map, group-by …
  - Actions e.g., count, print …
- **Control**
  - **Partitioning:** Spark also gives you control over how you can partition your RDDs.
  - **Persistence:** Allows you to choose whether you want to persist RDD onto disk or not.

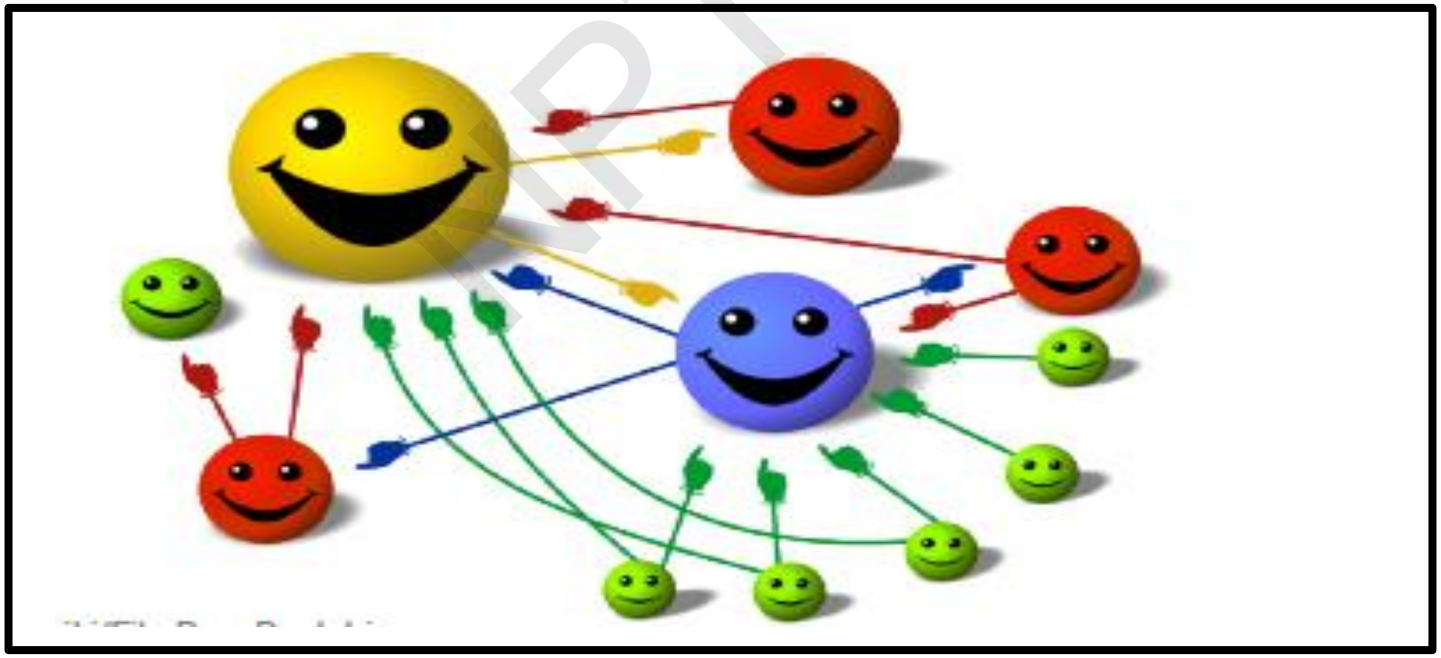
Partitioning: PageRank



- Joins take place repeatedly
- Good partitioning reduces shuffles

Example: PageRank

- Give pages ranks (scores) based on links to them
- Links from many pages $\Rightarrow$ high rank
- Links from a high-rank page $\Rightarrow$ high rank

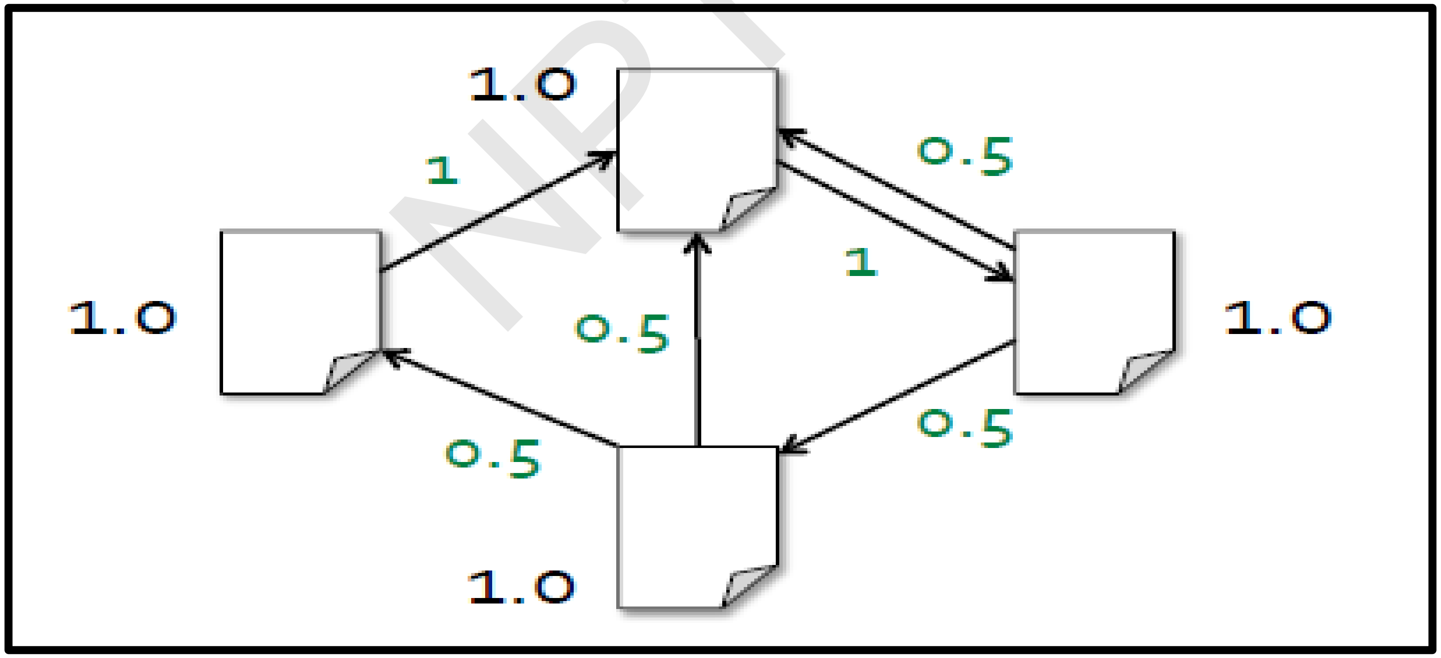


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute **$rank_p$/ |$neighbors_p$|** to its neighbors

Step-3 Set each page's rank to **0.15 + 0.85 x contributions**

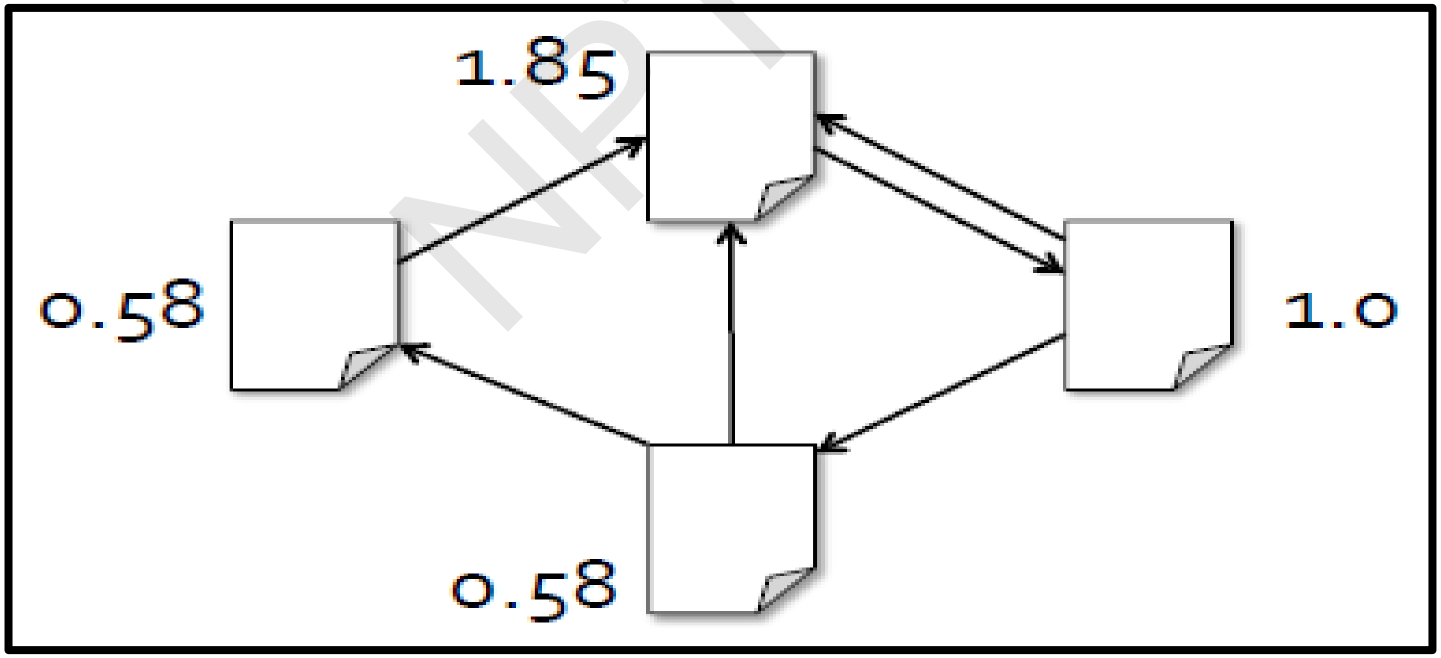


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute **$rank_p$/ $|neighbors_p|$** to its neighbors

Step-3 Set each page's rank to **0.15 + 0.85 x contributions**

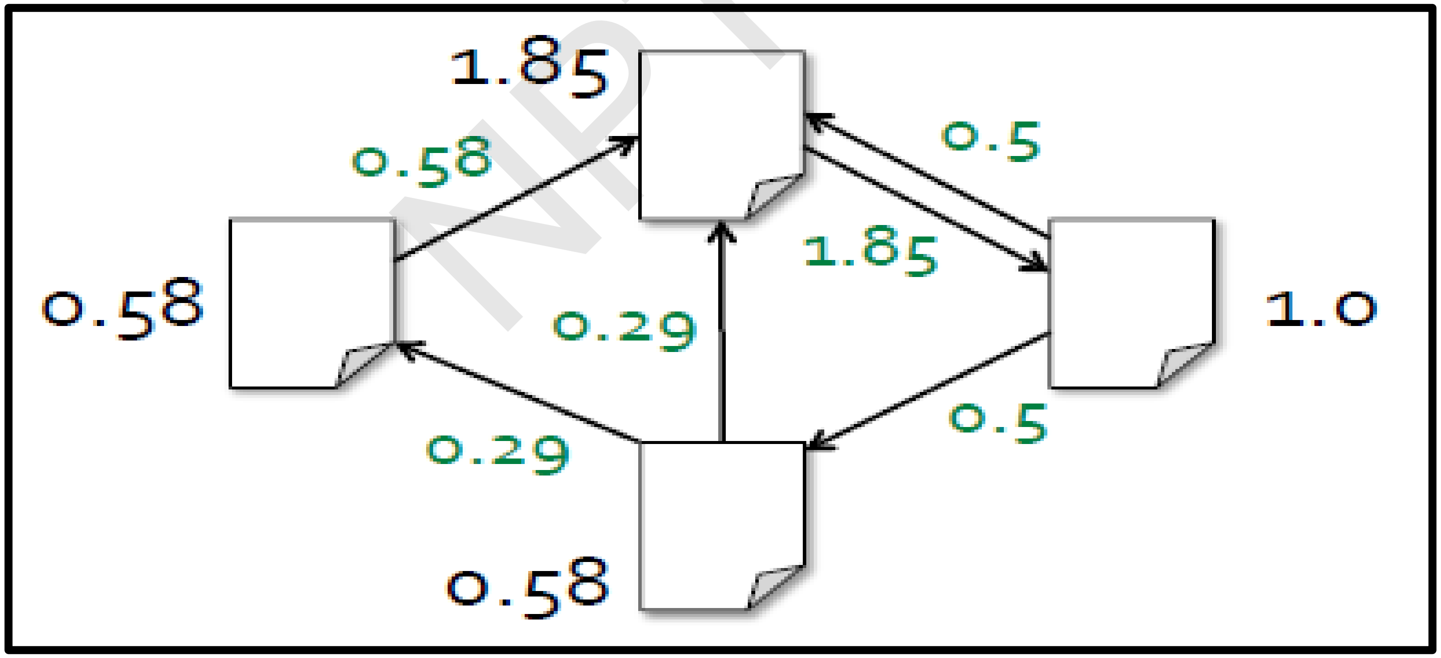


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute **$rank_p$/ |$neighbors_p$|** to its neighbors

Step-3 Set each page's rank to **0.15 + 0.85 x contributions**

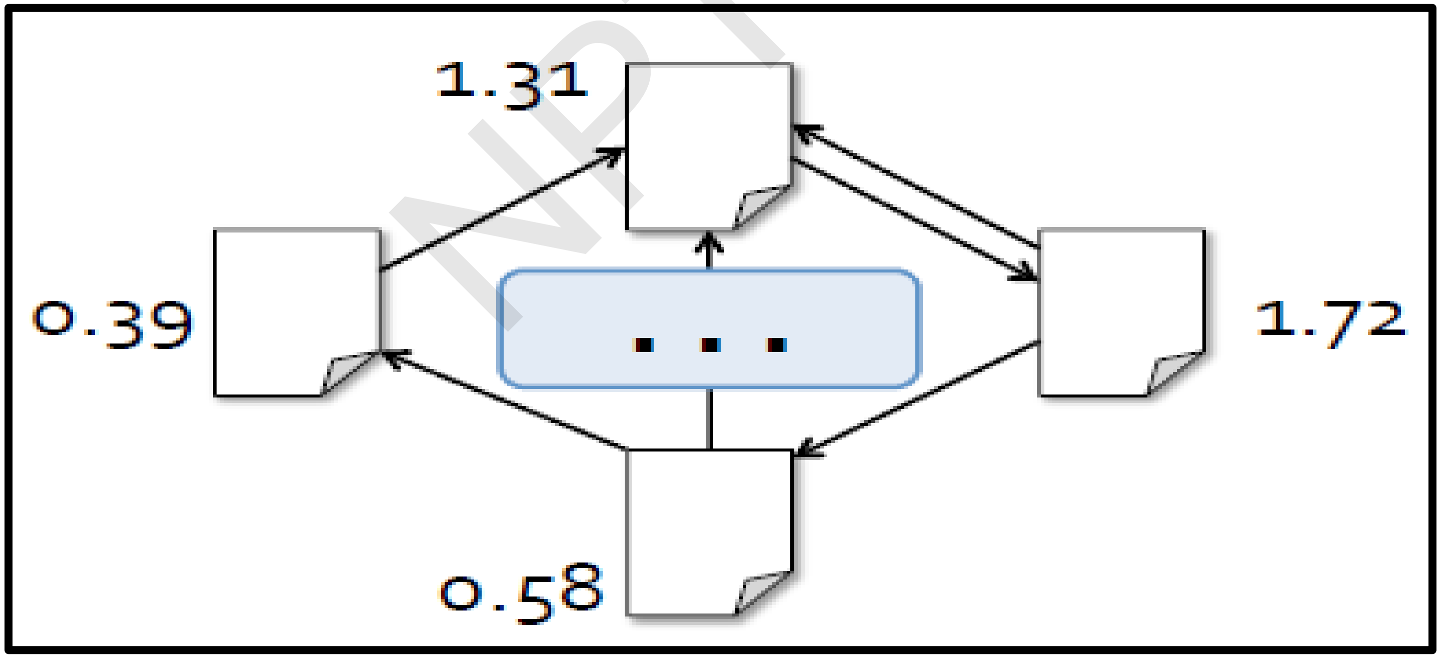


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute **$rank_p / |neighbors_p|$** to its neighbors

Step-3 Set each page's rank to **0.15 + 0.85 x contributions**

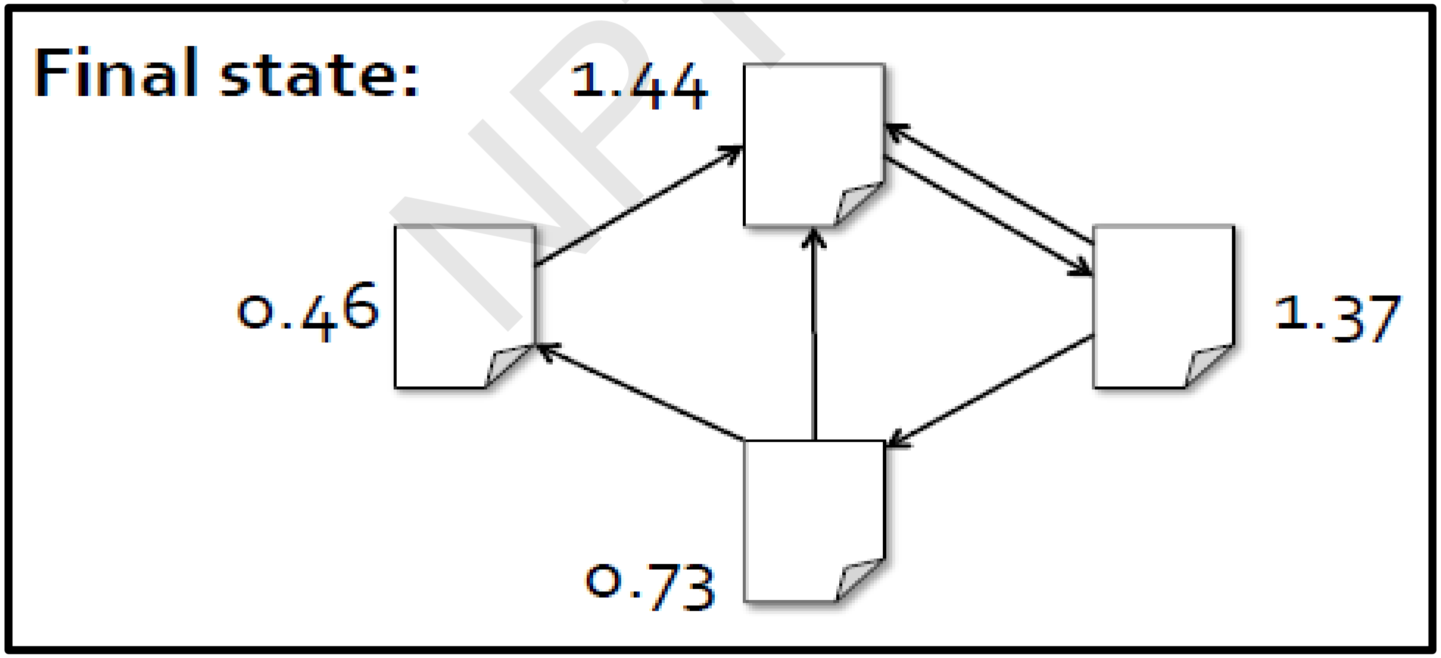


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute **$rank_p / |neighbors_p|$** to its neighbors

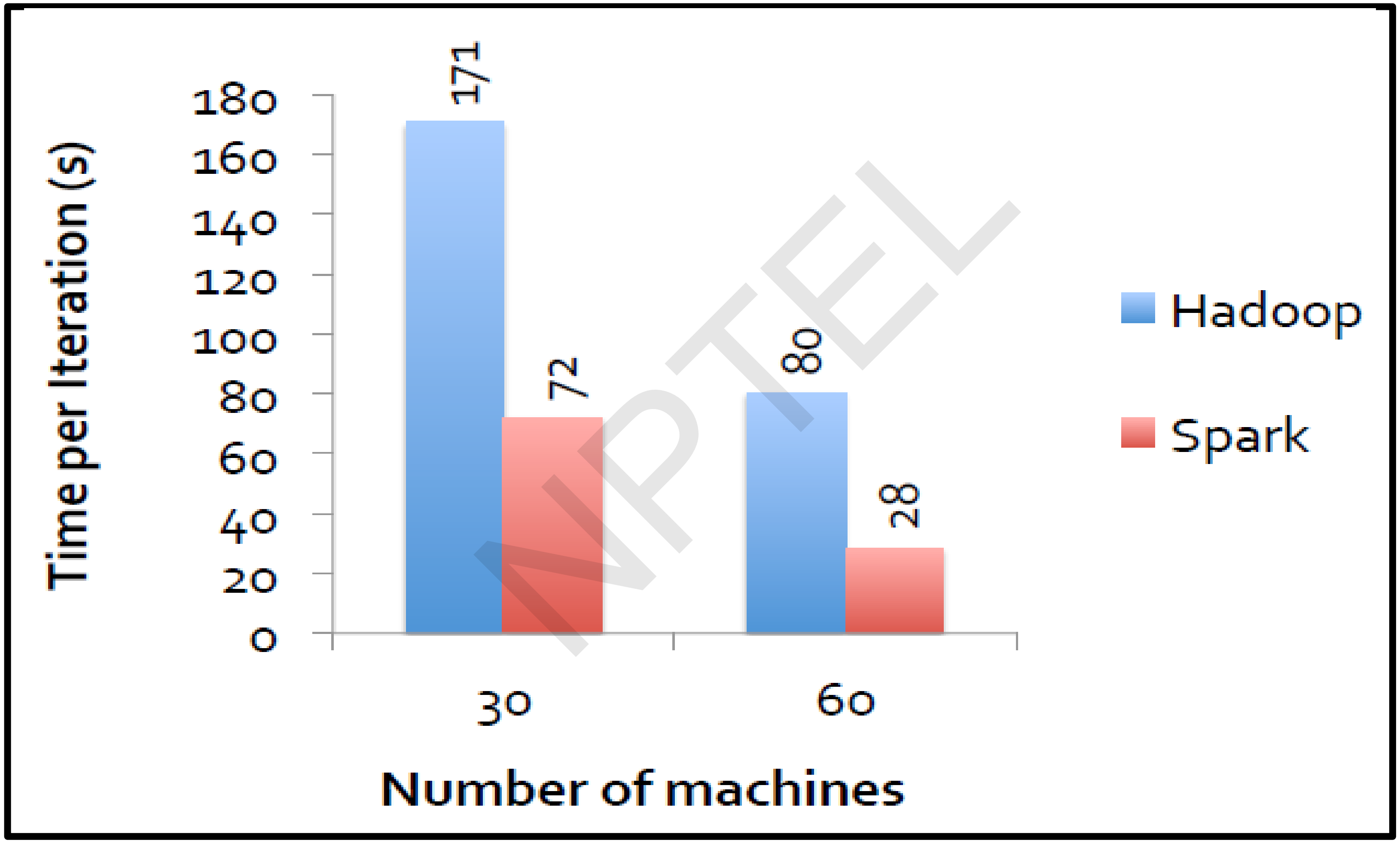
Step-3 Set each page's rank to **0.15 + 0.85 x contributions**



Spark Program

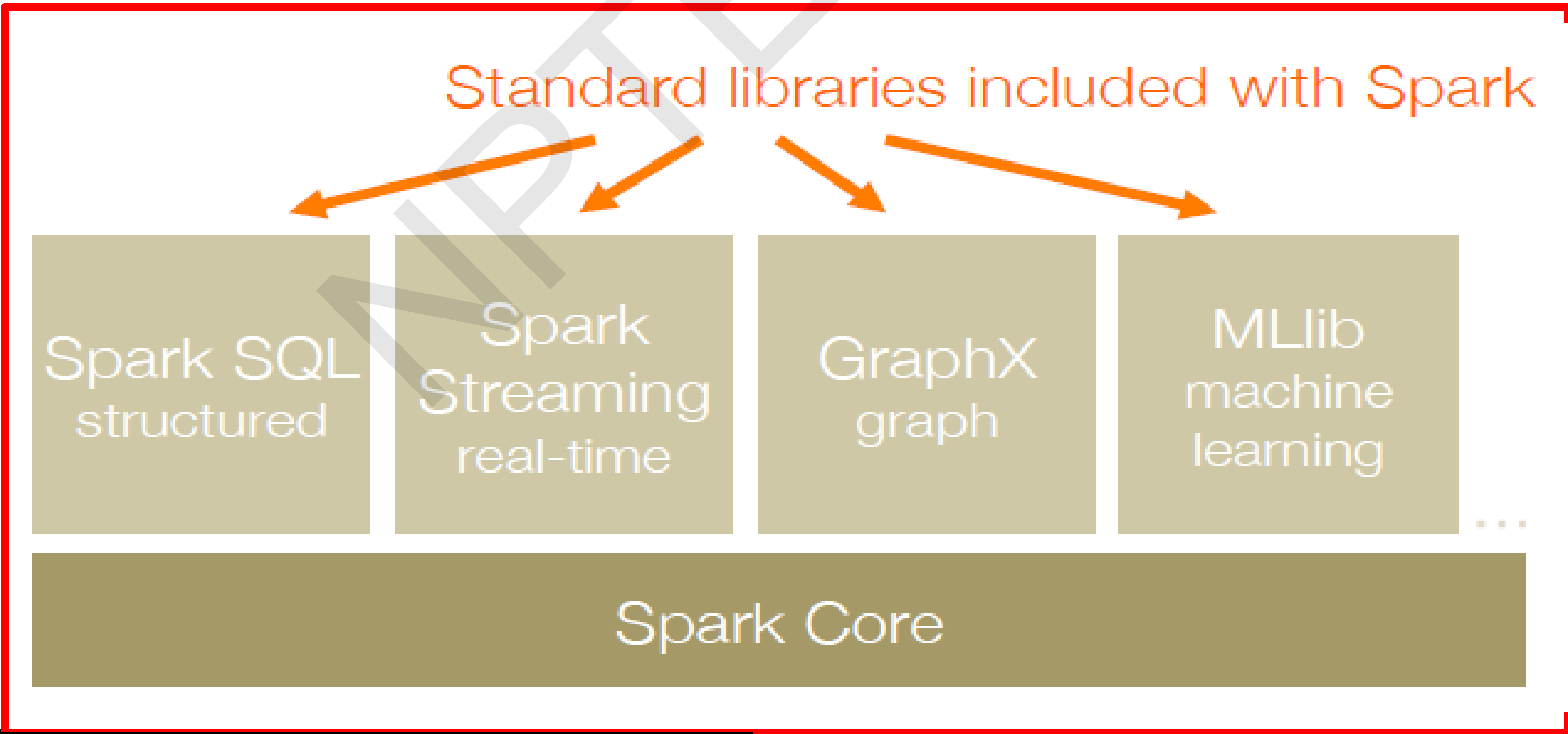
```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
    val contribs = links.join(ranks).flatMap {
        case (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
     }
     ranks = contribs.reduceByKey (_ + _)
                               .mapValues (0.15 + 0.85 * _)
  }
  ranks.saveAsTextFile(...)
```

PageRank Performance

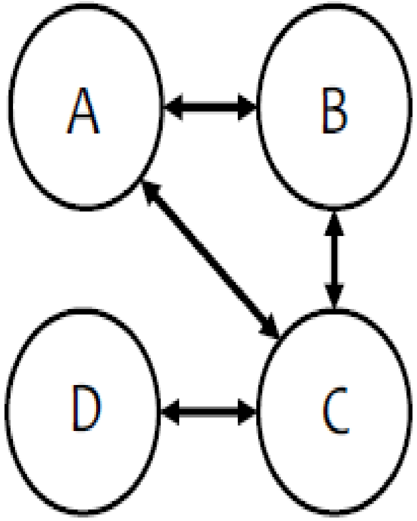


Generality

- RDDs allow unification of different programming models
    - Stream Processing
    - Graph Processing
    - Machine Learning

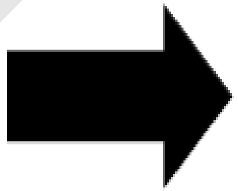


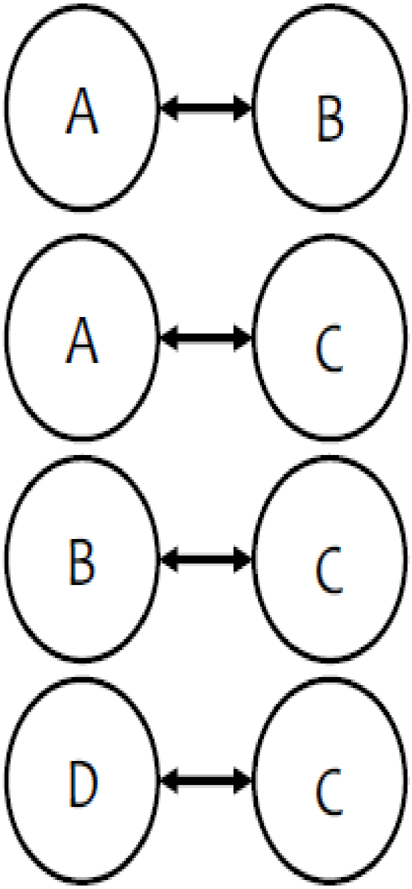
Gather-Apply-Scatter on GraphX



Vertices	Neighbors
A	B
A	C
B	C
D	C

Triplets

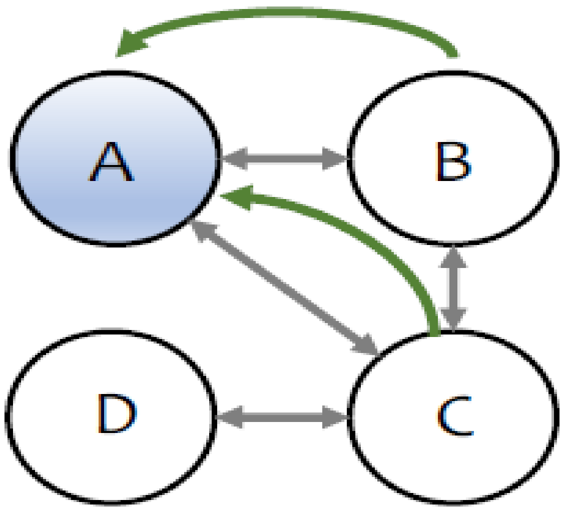




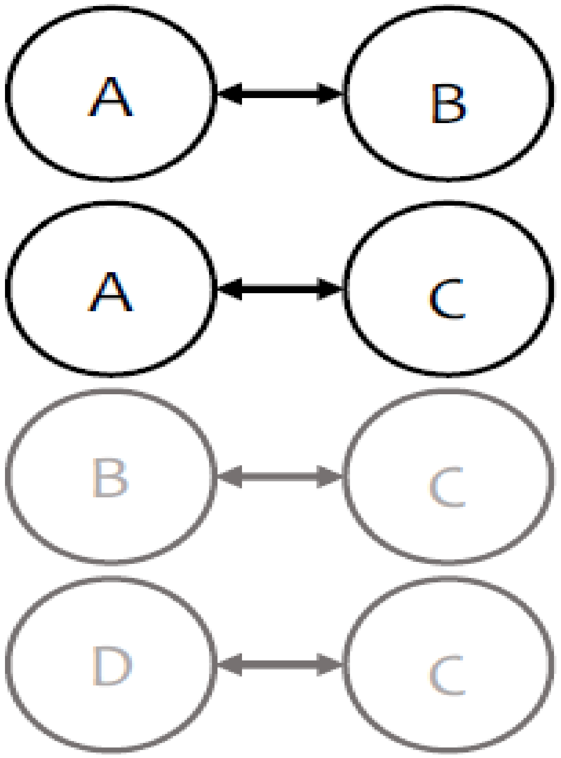
Graph Represented In a Table

Triplets

Gather-Apply-Scatter on GraphX

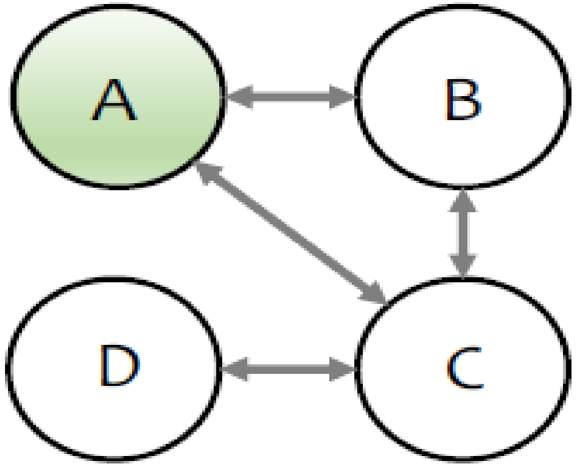


Gather at A

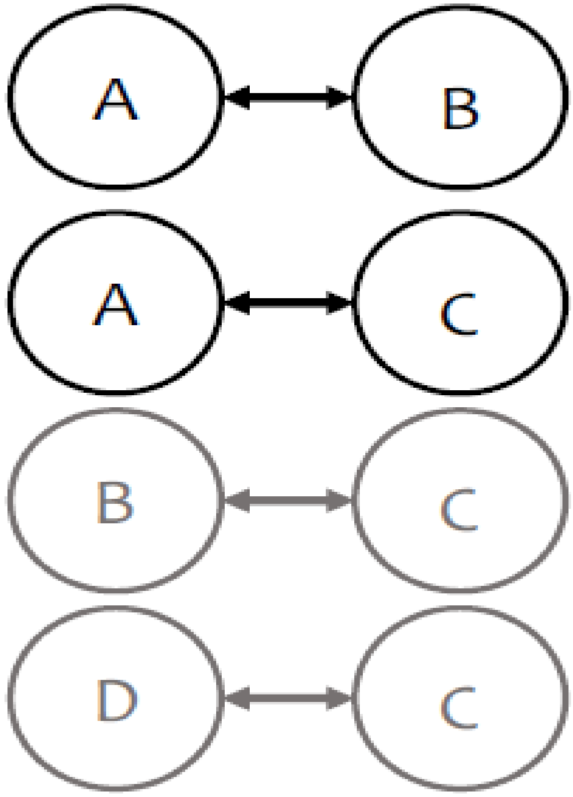


Group-By A

Gather-Apply-Scatter on GraphX

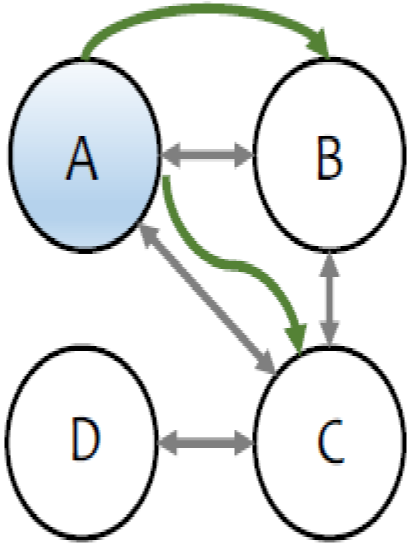


Apply

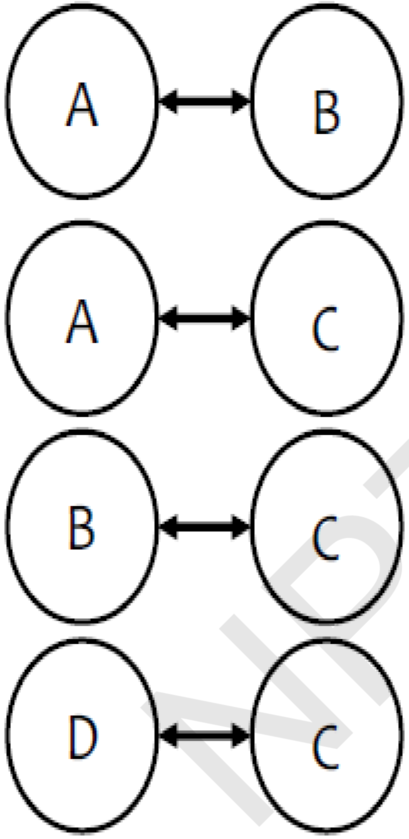


Map

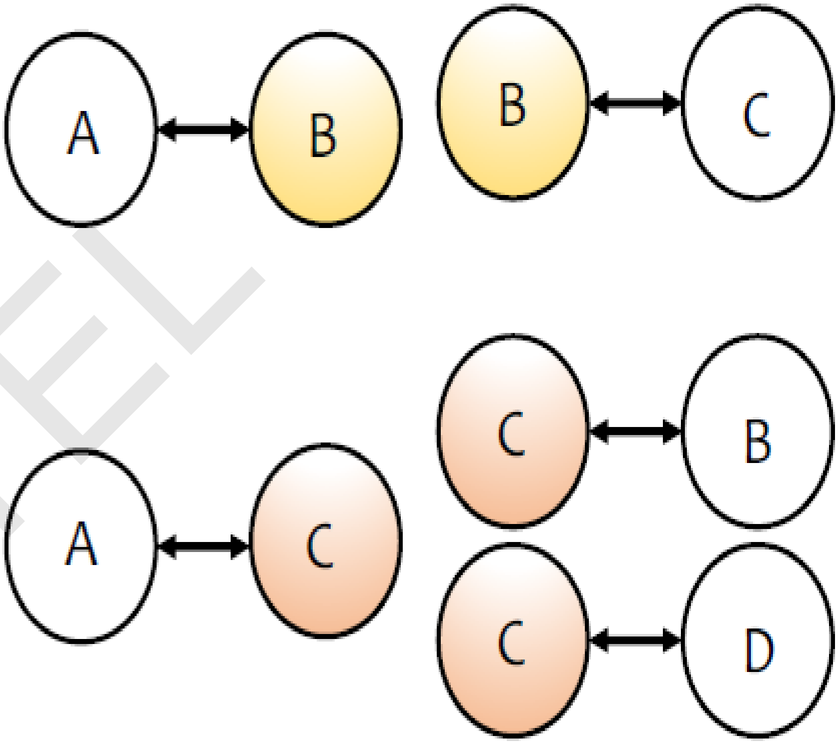
Gather-Apply-Scatter on GraphX



Scatter



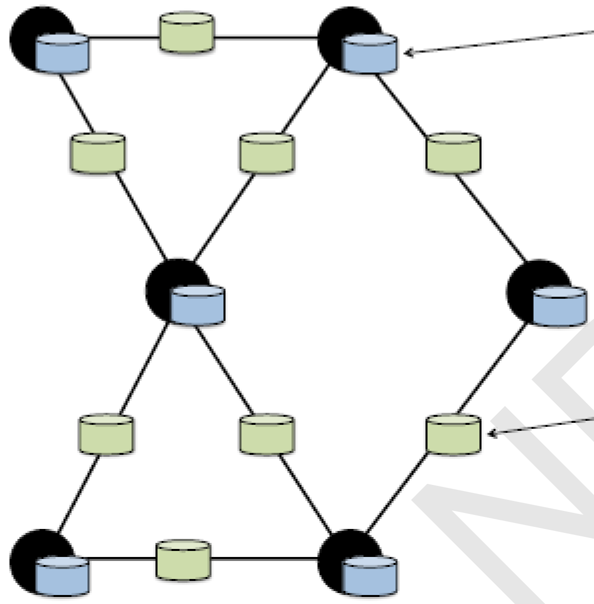
Triplets



Join

The GraphX API

Graphs Property



Vertex Property:

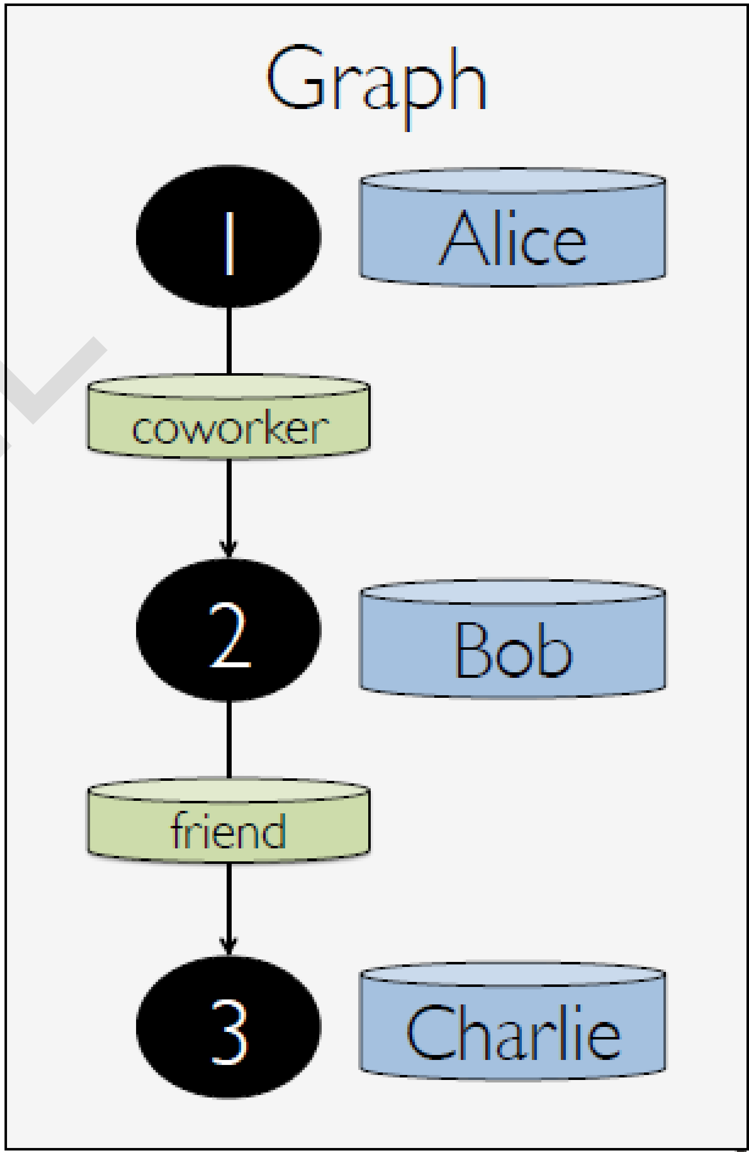
- User Profile
- Current PageRank Value

Edge Property:

- Weights
- Relationships
- Timestamps

Creating a Graph (Scala)

```
type VertexId = Long

val vertices: RDD[(VertexId, String)] =
  sc.parallelize(List(
    (1L, “Alice”),
    (2L, “Bob”),
    (3L, “Charlie”)))

class Edge[ED](
  val srcId: VertexId,
  val dstId: VertexId,
  val attr: ED)

val edges: RDD[Edge[String]] =
  sc.parallelize(List(
    Edge(1L, 2L, “coworker”),
    Edge(2L, 3L, “friend”)))

val graph = Graph(vertices, edges)
```



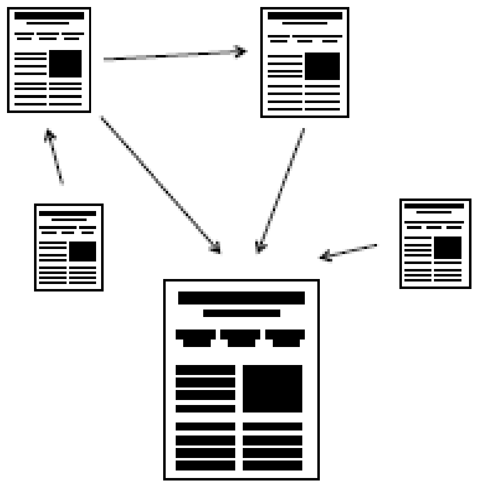
Graph Operations (Scala)

```
class Graph[VD, ED] {
    // Table Views -----------------------
    def vertices: RDD[(VertexId, VD)]
    def edges: RDD[Edge[ED]]
    def triplets: RDD[EdgeTriplet[VD, ED]]
    // Transformations ------------------------------------
    def mapVertices[VD2](f: (VertexId, VD) => VD2): Graph[VD2, ED]
    def mapEdges[ED2](f: Edge[ED] => ED2): Graph[VD2, ED]
    def reverse: Graph[VD, ED]
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
    // Joins ----------------------------------------
    def outerJoinVertices[U, VD2]
        (tbl: RDD[(VertexId, U)])
        (f: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
    // Computation ----------------------------------
    def aggregateMessages[A](
        sendMsg: EdgeContext[VD, ED, A] => Unit,
        mergeMsg: (A, A) => A): RDD[(VertexId, A)]
```

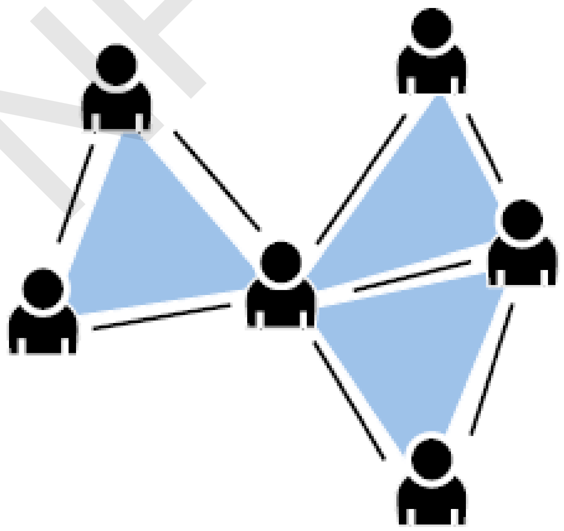
Built-in Algorithms (Scala)

```
    // Continued from previous slide
    def pageRank(tol: Double): Graph[Double, Double]
    def triangleCount(): Graph[Int, ED]
    def connectedComponents(): Graph[VertexId, ED]
    // ...and more: org.apache.spark.graphx.lib
}
```

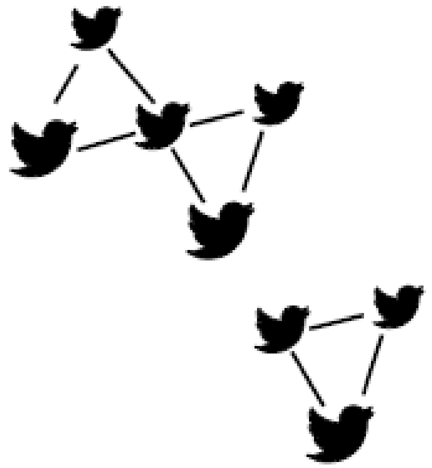
PageRank



Triangle Count

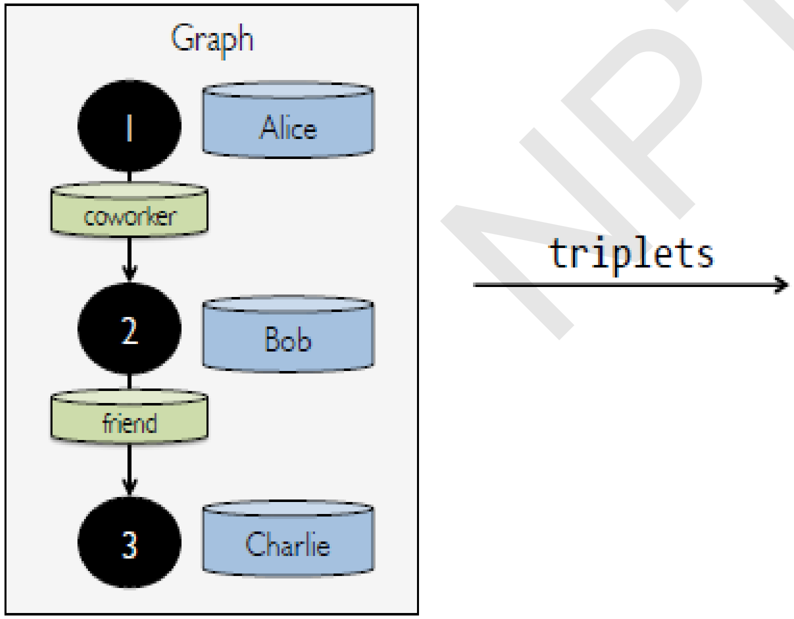


Connected Components



The triplets view

```
class Graph[VD, ED] {
  def triplets: RDD[EdgeTriplet[VD, ED]]
}

class EdgeTriplet[VD, ED](
  val srcId: VertexId, val dstId: VertexId, val attr: ED,
  val srcAttr: VD, val dstAttr: VD)
```

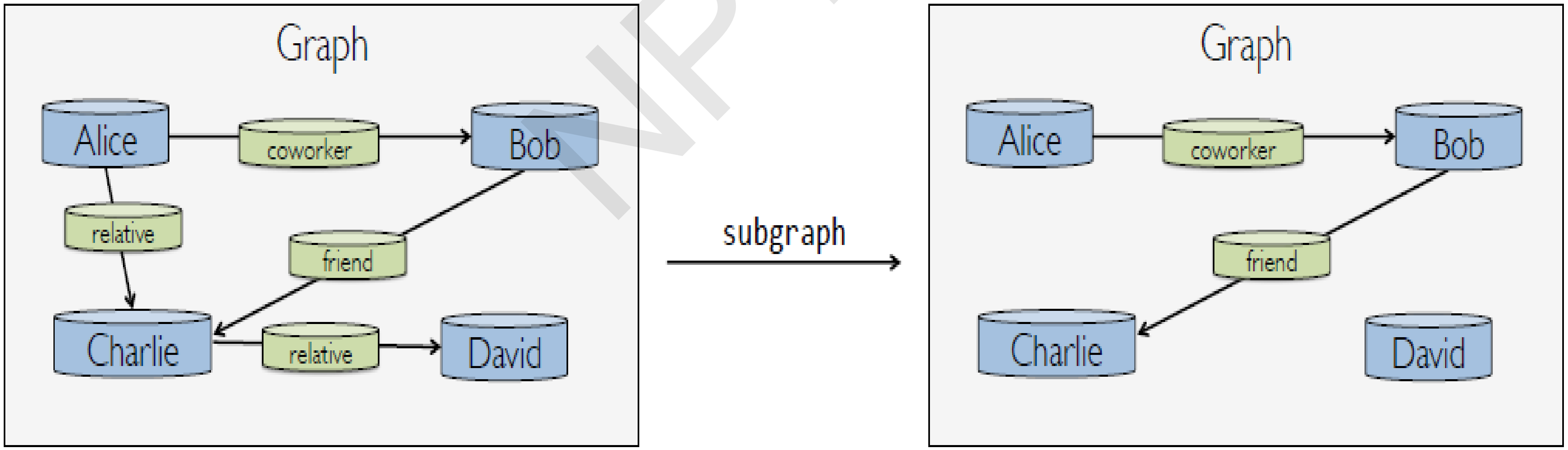


RDD

srcAttr	dstAttr	attr
Alice	coworker	Bob
Bob	friend	Charlie

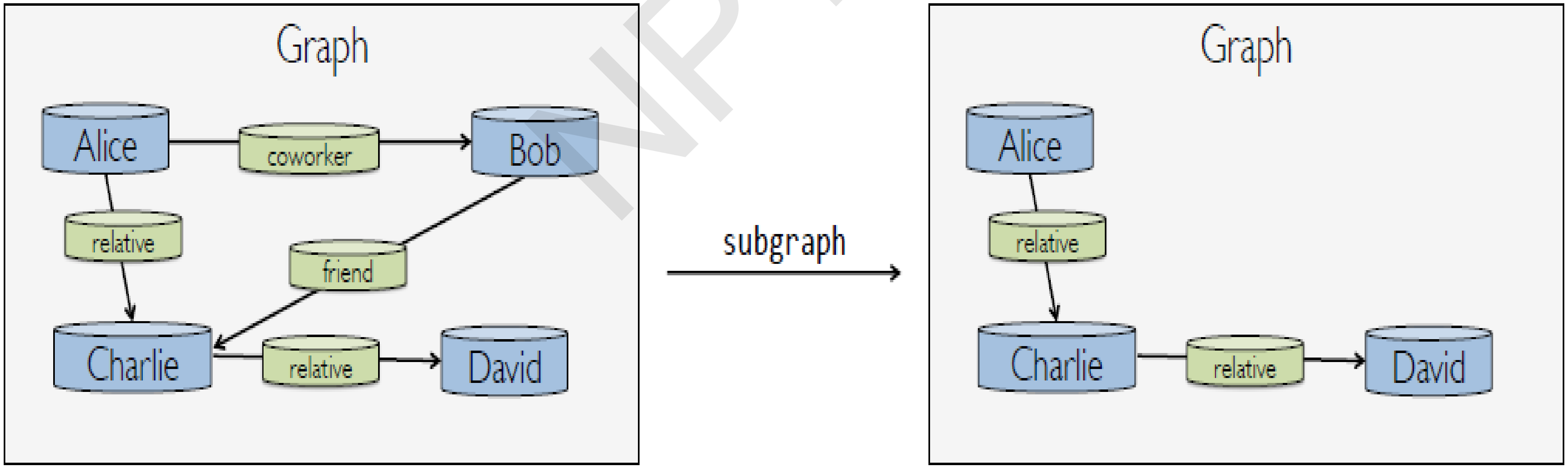
The subgraph transformation

```
class Graph[VD, ED] {
  def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
}

graph.subgraph(epred = (edge) => edge.attr != "relative")
```



The subgraph transformation

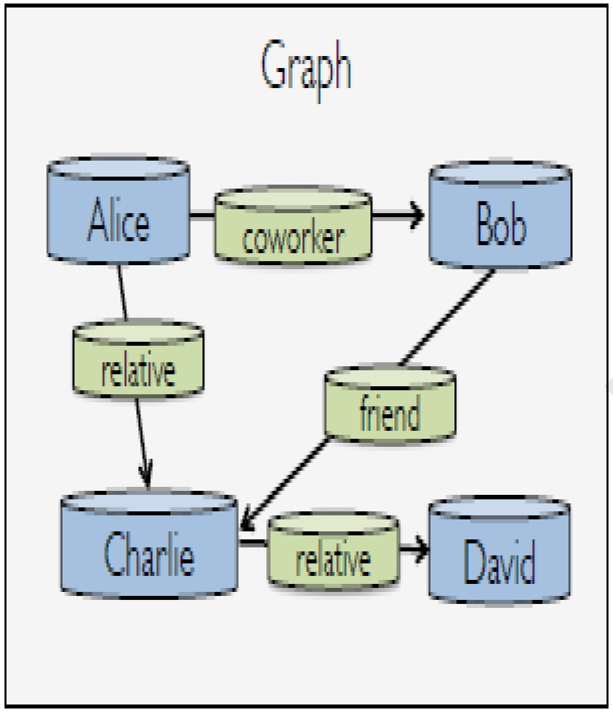
```
class Graph[VD, ED] {
  def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
}

graph.subgraph(vpred = (id, name) => name != "Bob")
```



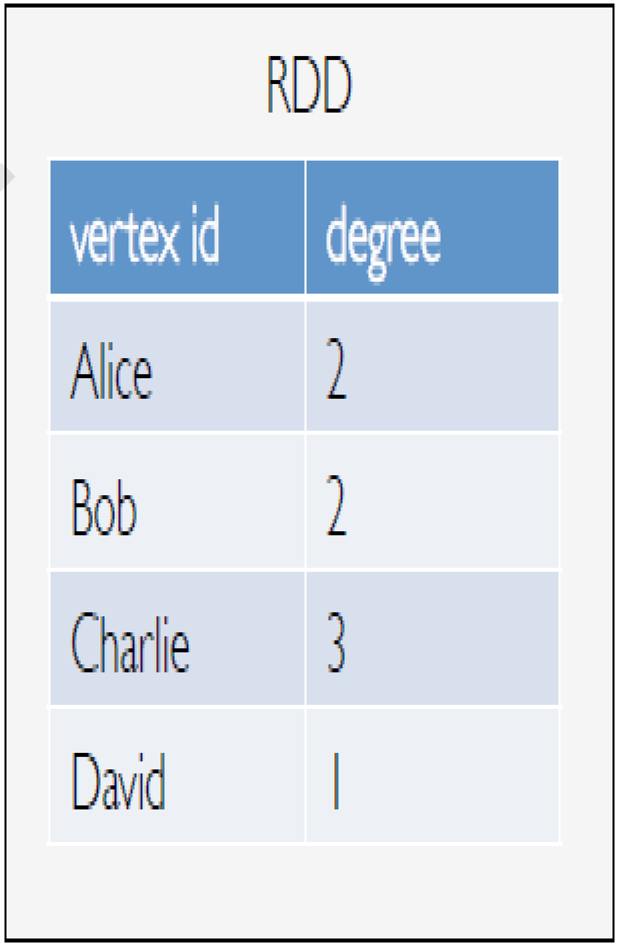
Computation with aggregateMessages

```
class Graph[VD, ED] {
  def aggregateMessages[A](
    sendMsg: EdgeContext[VD, ED, A] => Unit,
    mergeMsg: (A, A) => A): RDD[(VertexId, A)]
}

class EdgeContext[VD, ED, A](
    val srcId: VertexId, val dstId: VertexId, val attr: ED,
    val srcAttr: VD, val dstAttr: VD) {
  def sendToSrc(msg: A)
  def sendToDst(msg: A)
}

graph.aggregateMessages(
  ctx => {
    ctx.sendToSrc(1)
    ctx.sendToDst(1)
  },
  _ + _)
```

Computation with aggregateMessages

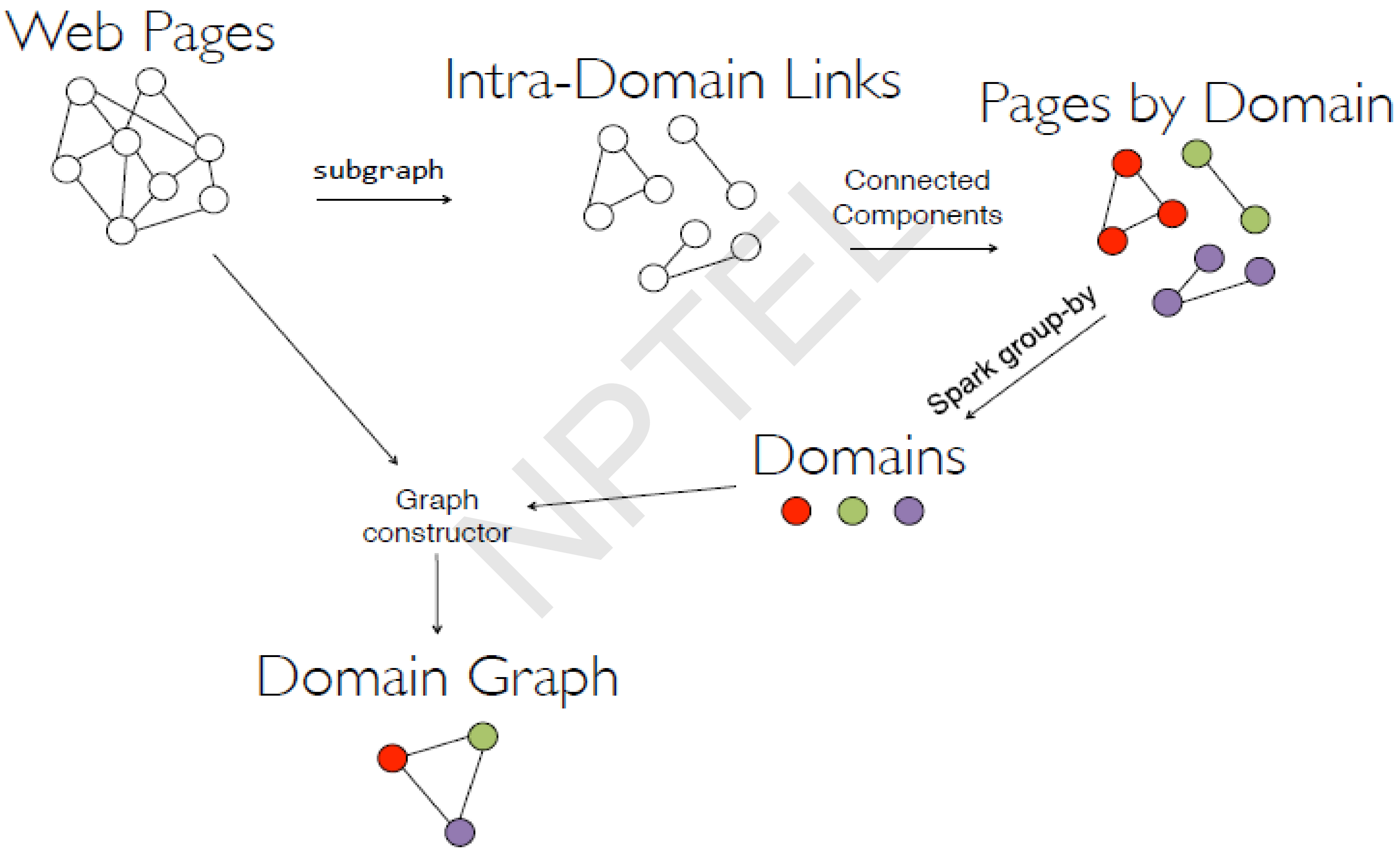


aggregateMessages →


RDD

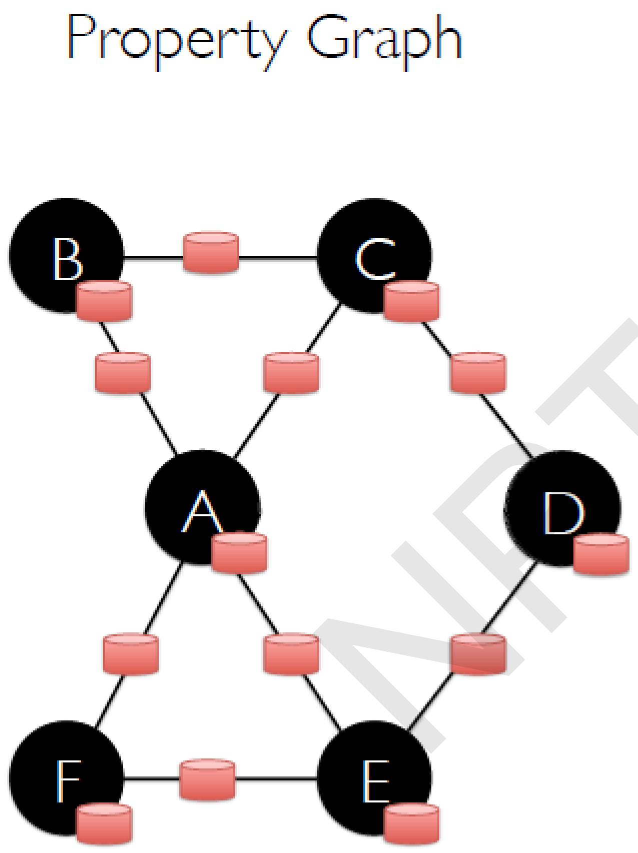
vertex id	degree
Alice	2
Bob	2
Charlie	3
David	1

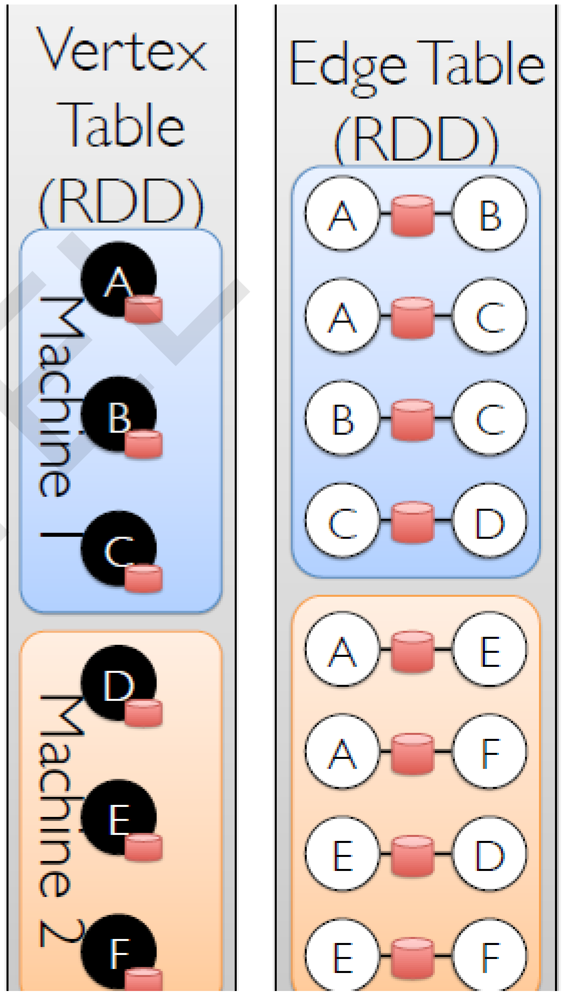
Example: Graph Coarsening



How GraphX Works

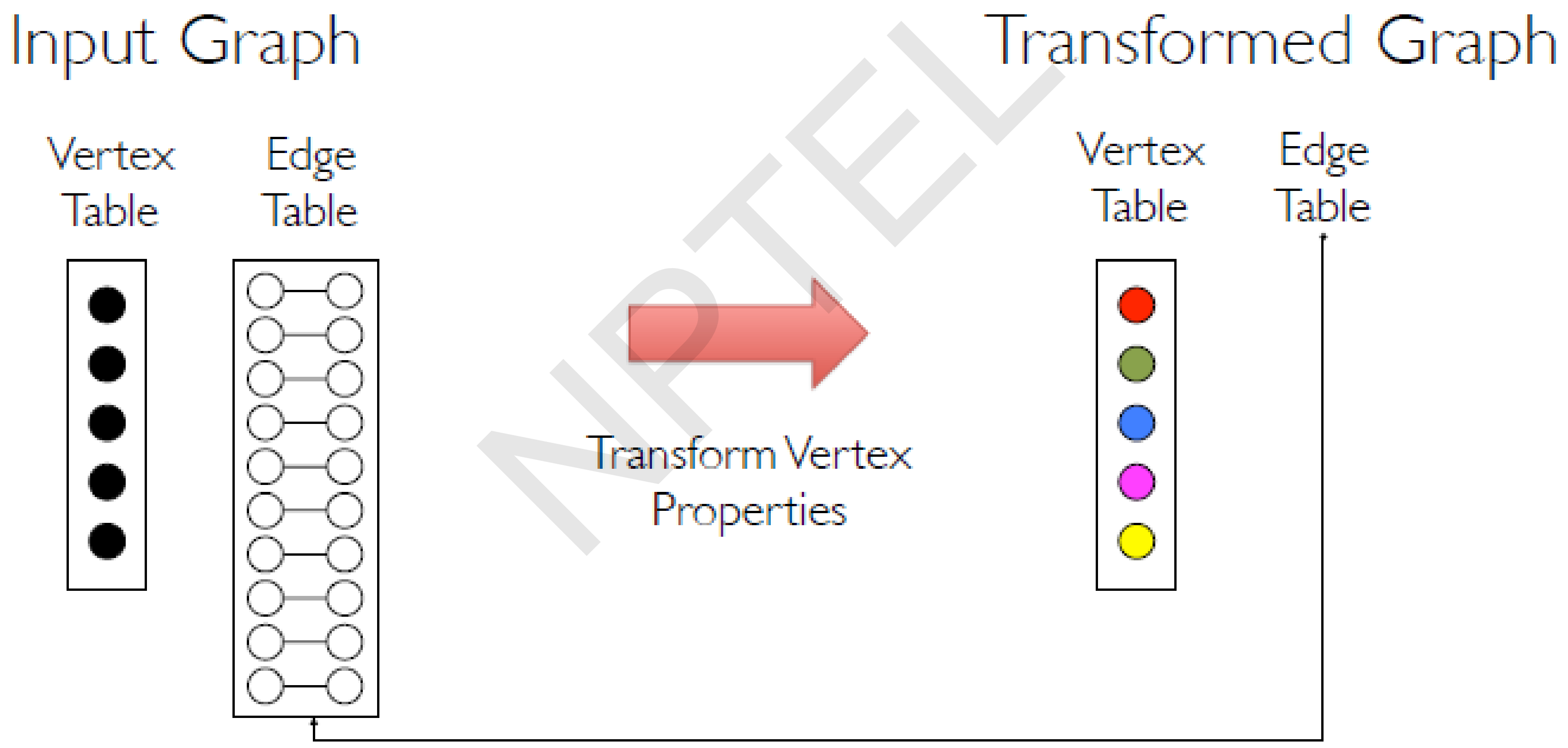
Storing Graphs as Tables



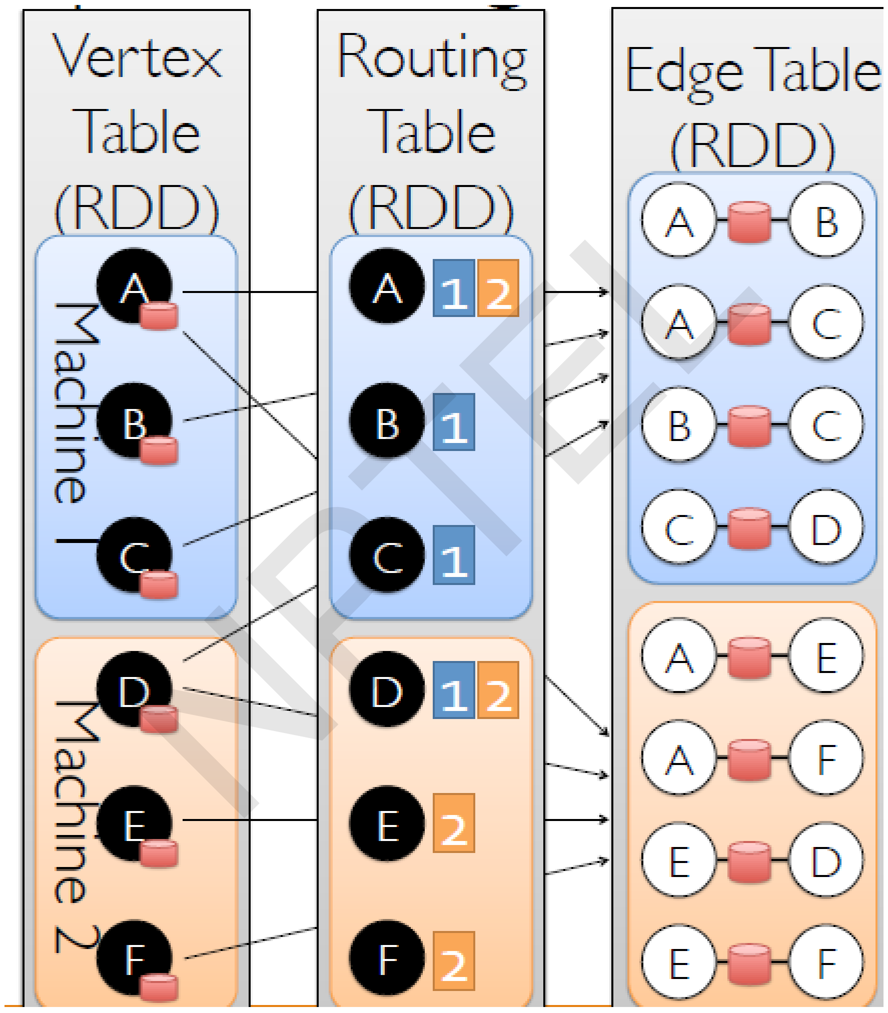


Simple Operations

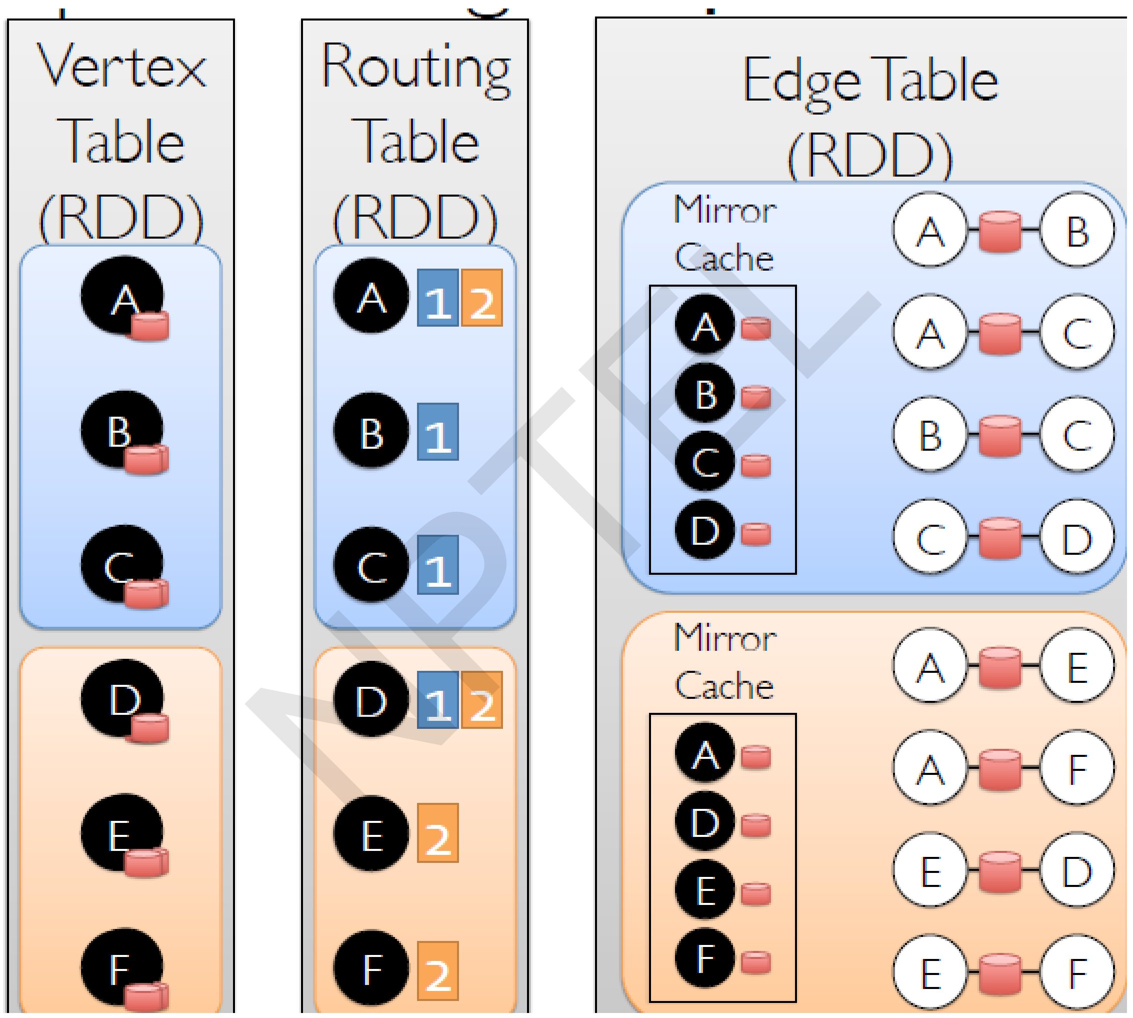
Reuse vertices or edges across multiple graphs



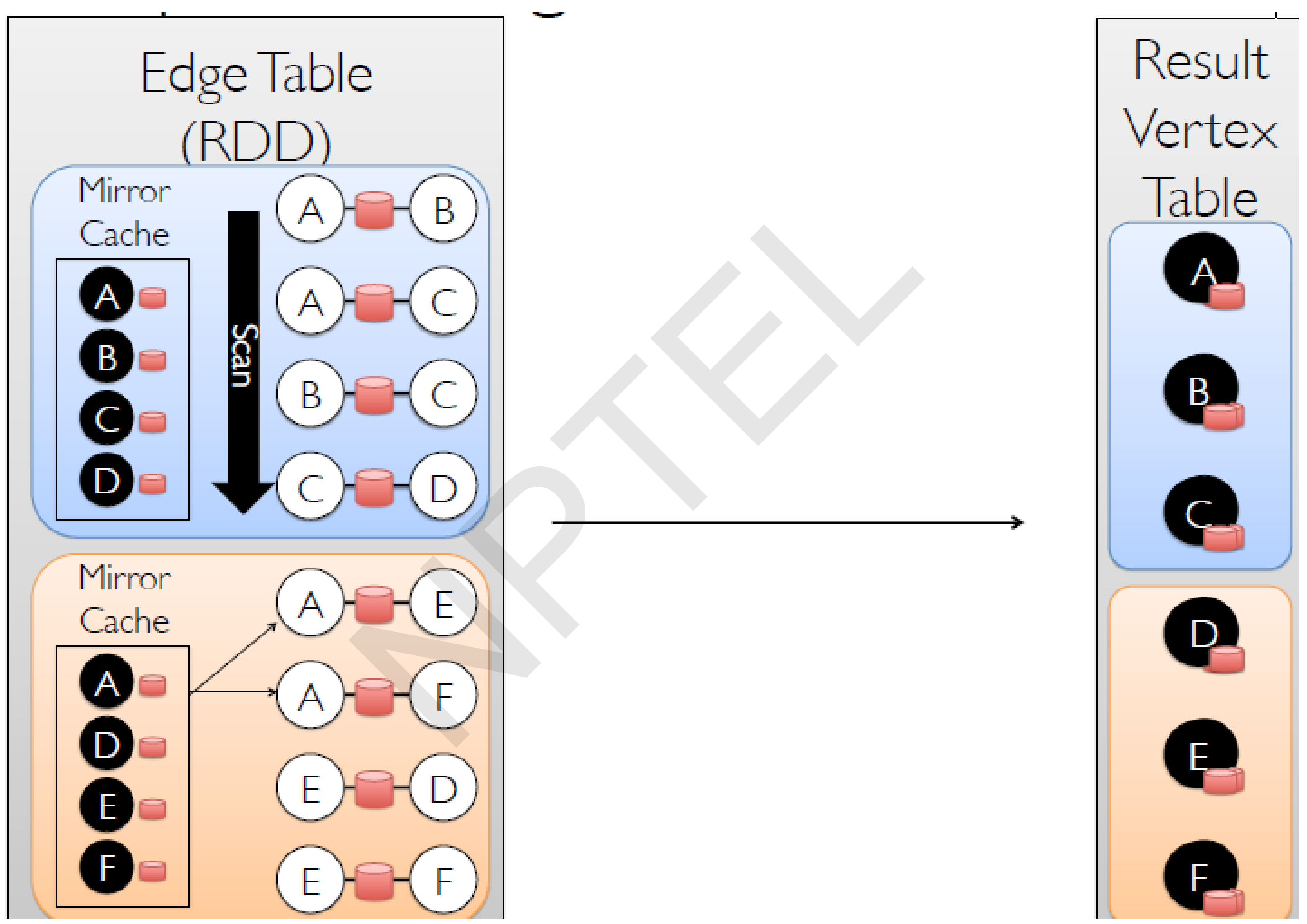
Implementing triplets



Implementing triplets



Implementing aggregateMessages



Future of GraphX

1. Language support

a) Java API

b) Python API: collaborating with Intel, SPARK-3789

2. More algorithms

a) LDA (topic modeling)

b) Correlation clustering

3. Research

a) Local graphs

b) Streaming/time-varying graphs

c) Graph database–like queries

Other Spark Applications

i. Twitter spam classification

ii. EM algorithm for traffic prediction

iii. K-means clustering

iv. Alternating Least Squares matrix factorization

v. In-memory OLAP aggregation on Hive data

vi. SQL on Spark

Reading Material

- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica

"Spark: Cluster Computing with Working Sets"

- Matei Zaharia, Mosharaf Chowdhury et al.

"Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing"

https://spark.apache.org/

Conclusion

- RDDs (Resilient Distributed Datasets (RDDs) provide a simple and efficient programming model
- Generalized to a broad set of applications
- Leverages coarse-grained nature of parallel algorithms for failure recovery

Introduction to Kafka

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

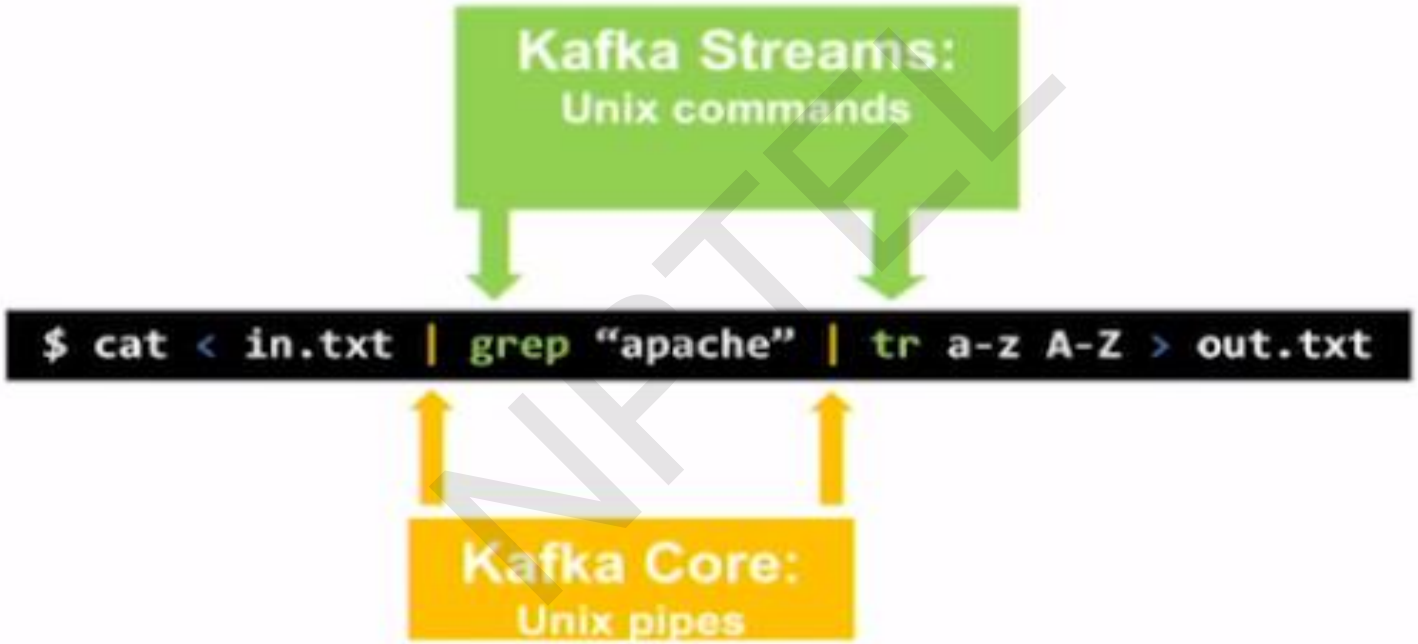
- Define Kafka
- Describe some use cases for Kafka
- Describe the Kafka data model
- Describe Kafka architecture
- List the types of messaging systems
- Explain the importance of brokers

Batch vs. Streaming

Batch

Streaming

1. Apache Kafka: a Streaming Data Platform Unix Pipelines Analogy



```
$ cat < in.txt | grep "apache" | tr a-z A-Z > out.txt
```

- Kafka Core: is the distributed, durable equivalent of Unix pipes. Use it to connect and compose your large-scale data applications.
- Kafka Streams are the commands of your Unix pipelines. Use it to transform data stored in Kafka.
- Kafka Connect is the I/O redirection in your Unix pipelines. Use it to get your data into and out of Kafka.

Introduction: Apache Kafka

- **Kafka is a high-performance, real-time messaging system. It is an open source tool and is a part of Apache projects.**

- The characteristics of Kafka are:

  1. **It is a distributed and partitioned messaging system.**
  2. **It is highly fault-tolerant**
  3. **It is highly scalable.**
  4. **It can process and send millions of messages per second to several receivers.**

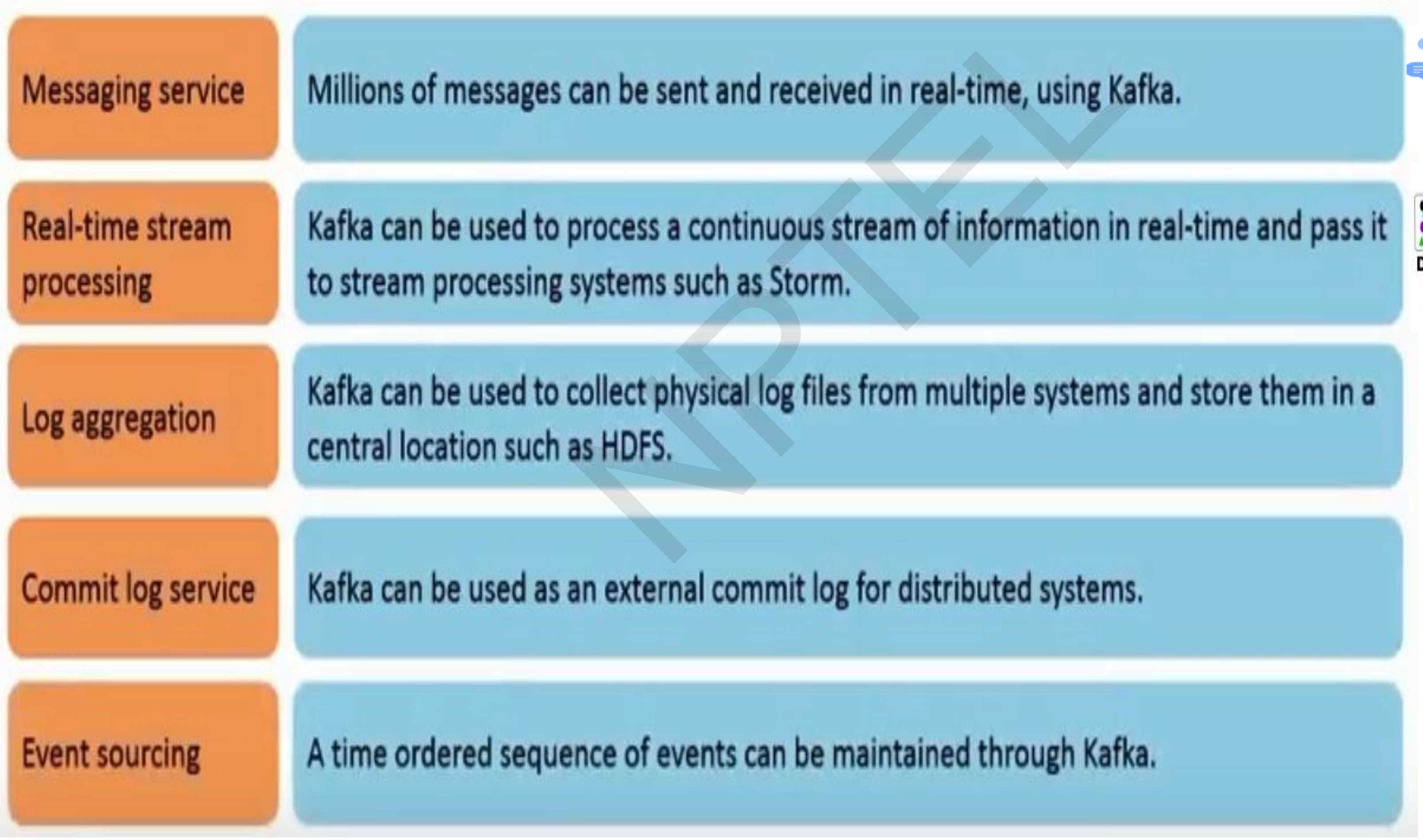
Kafka History

- Apache Kafka was originally developed by LinkedIn and later, handed over to the open source community in early 2011.

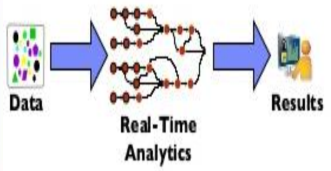
  - It became a main Apache project in October, 2012.
  - A stable Apache Kafka version 0.8.2.0 was release in Feb, 2015.
  - A stable Apache Kafka version 0.8.2.1 was released in May, 2015, which is the latest version.

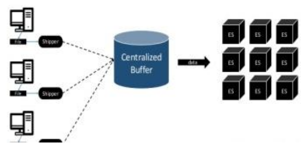
Kafka Use Cases

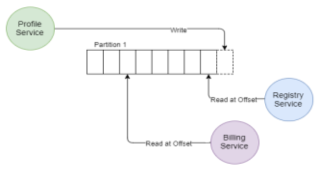
- Kafka can be used for various purposes in an organization, such as:

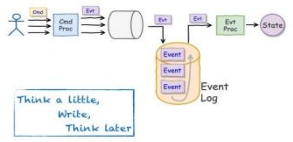


Messaging service	Millions of messages can be sent and received in real-time, using Kafka.
Real-time stream processing	Kafka can be used to process a continuous stream of information in real-time and pass it to stream processing systems such as Storm.
Log aggregation	Kafka can be used to collect physical log files from multiple systems and store them in a central location such as HDFS.
Commit log service	Kafka can be used as an external commit log for distributed systems.
Event sourcing	A time ordered sequence of events can be maintained through Kafka.





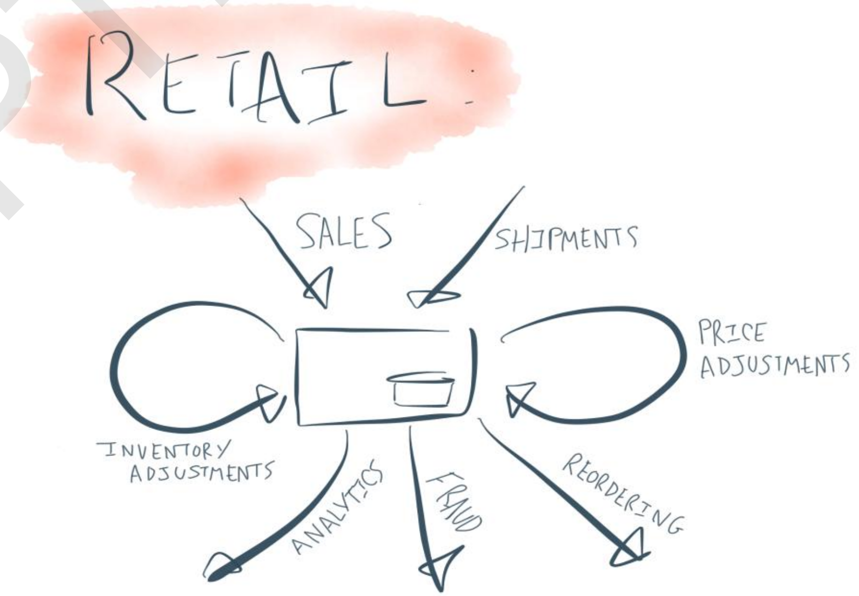




Apache Kafka: a Streaming Data Platform

- Most of **what a business does** can be thought as **event streams.** They are in a
  - **Retail system**: orders, shipments, returns, …
  - **Financial system**: stock ticks, orders, …
  - **Web site**: page views, clicks, searches, …
  - **IoT**: sensor readings, …

and so on.

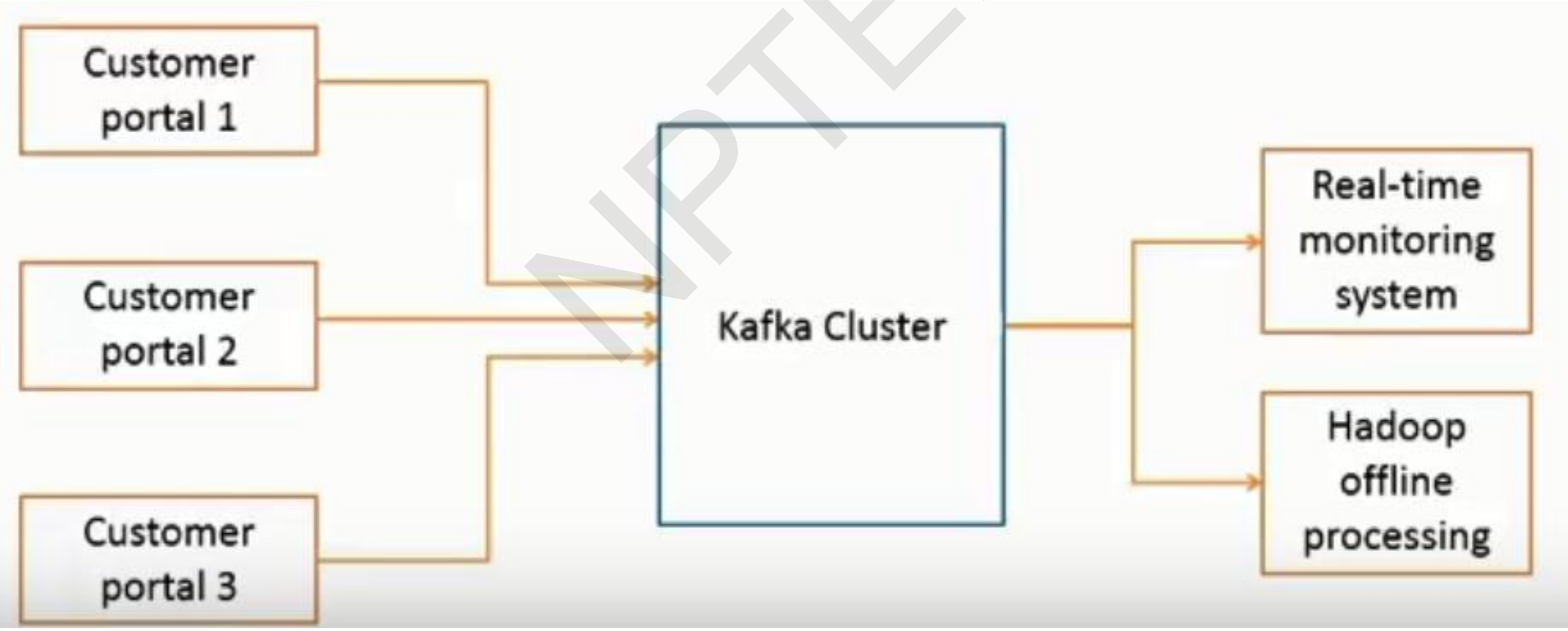


Enter Kafka

- Adopted at 1000s of companies worldwide

Aggregating User Activity Using Kafka-Example

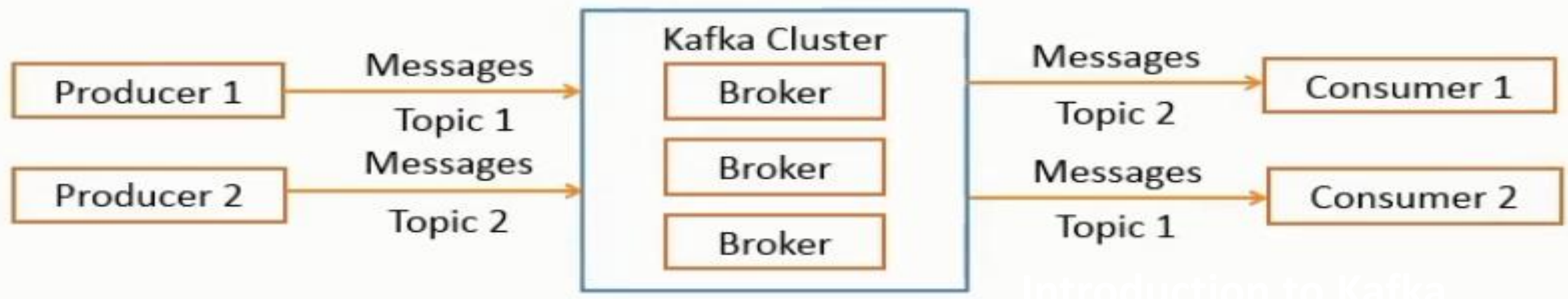
- Kafka can be used to aggregate user activity data such as clicks, navigation, and searches from different websites of an organization; such user activities can be sent to a real-time monitoring system and hadoop system for offline processing.



Kafka Data Model

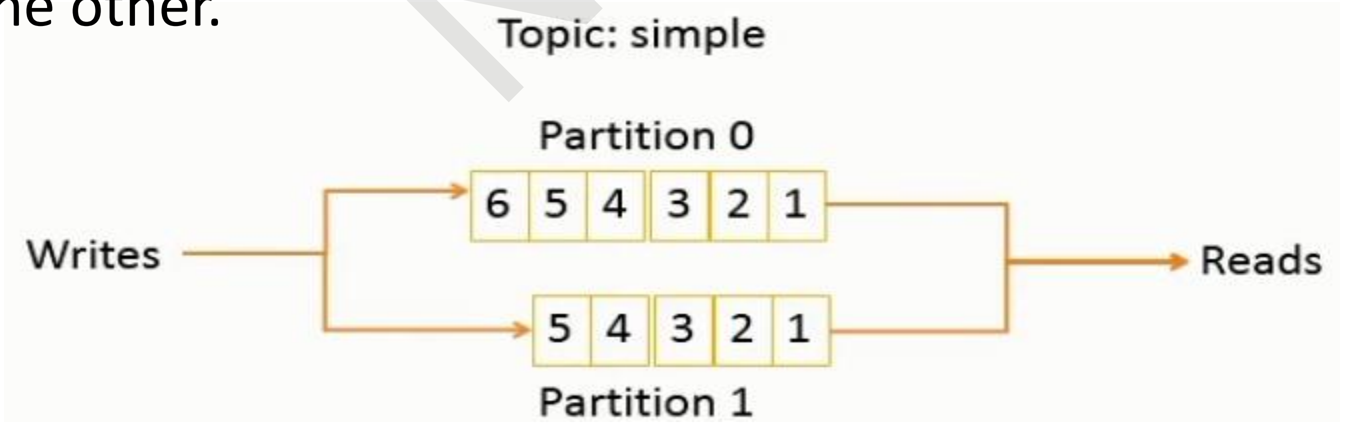
The Kafka data model consists of messages and topics.

- **Messages represent information such as, lines in a log file, a row of stock market data, or an error message from a system.**
- **Messages are grouped into categories called topics.**
  Example: LogMessage and Stock Message.
- **The processes that publish messages into a topic in Kafka are known as producers.**
- **The processes that receive the messages from a topic in Kafka are known as consumers.**
- **The processes or servers within Kafka that process the messages are known as brokers.**
- **A Kafka cluster consists of a set of brokers that process the messages.**



Topics

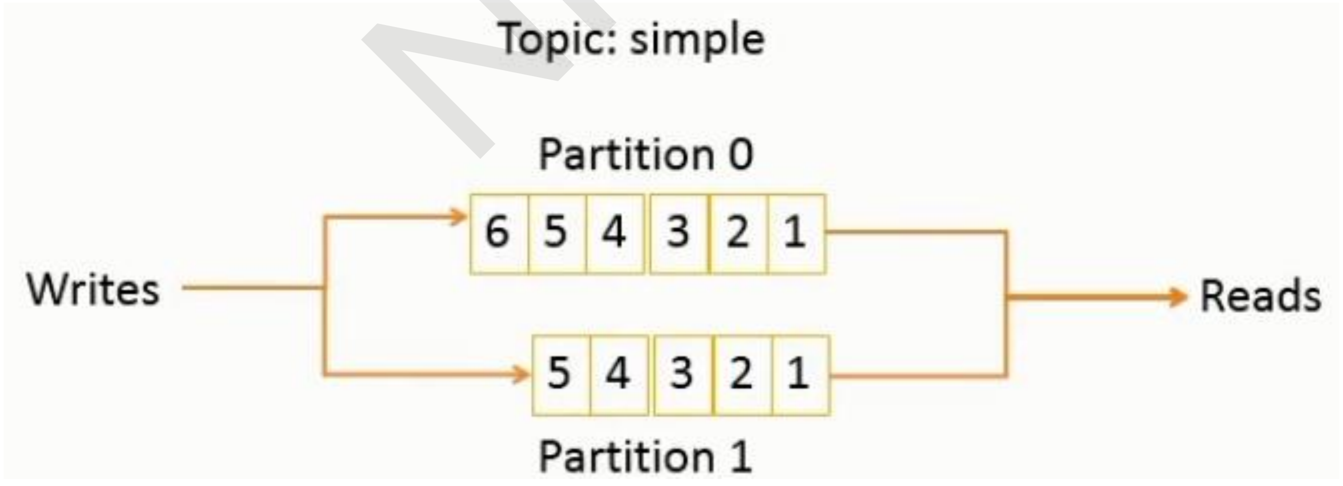
- A topic is a category of messages in Kafka.
- The producers publish the messages into topics.
- The consumers read the messages from topics.
- A topic is divided into one or more partitions.
- A partition is also known as a commit log.
- Each partition contains an ordered set of messages.
- Each message is identified by its offset in the partition.
- Messages are added at one end of the partition and consumed at the other.



Partitions

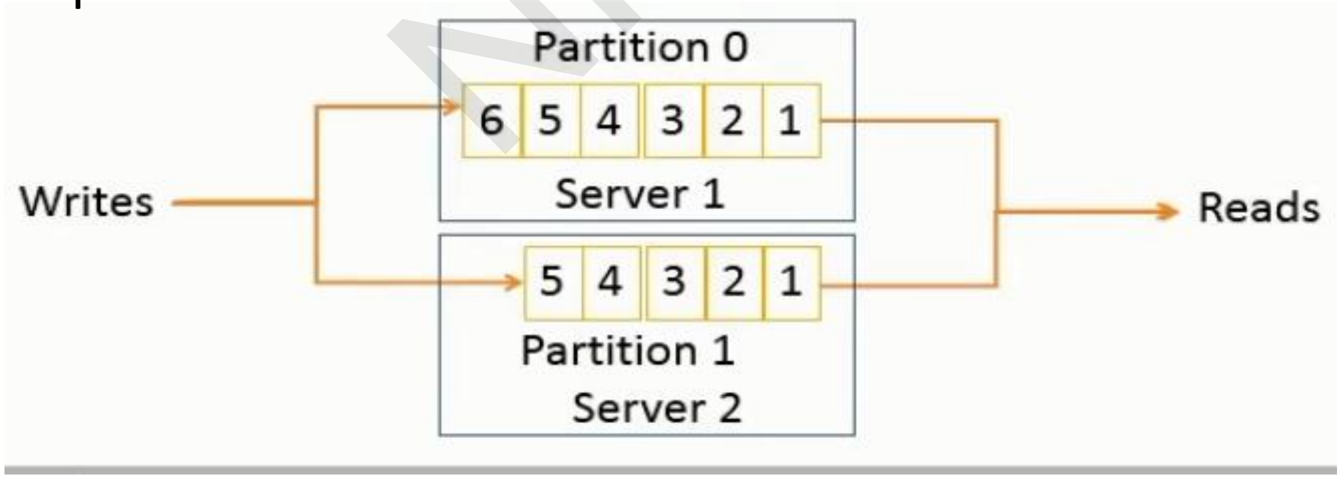
- Topics are divided into partitions, which are the unit of parallelism in Kafka.

  - Partitions allow messages in a topic to be distributed to multiple servers.
  - A topic can have any number of partitions.
  - Each partition should fit in a single Kafka server.
  - The number of partitions decide the parallelism of the topic.



Partition Distribution

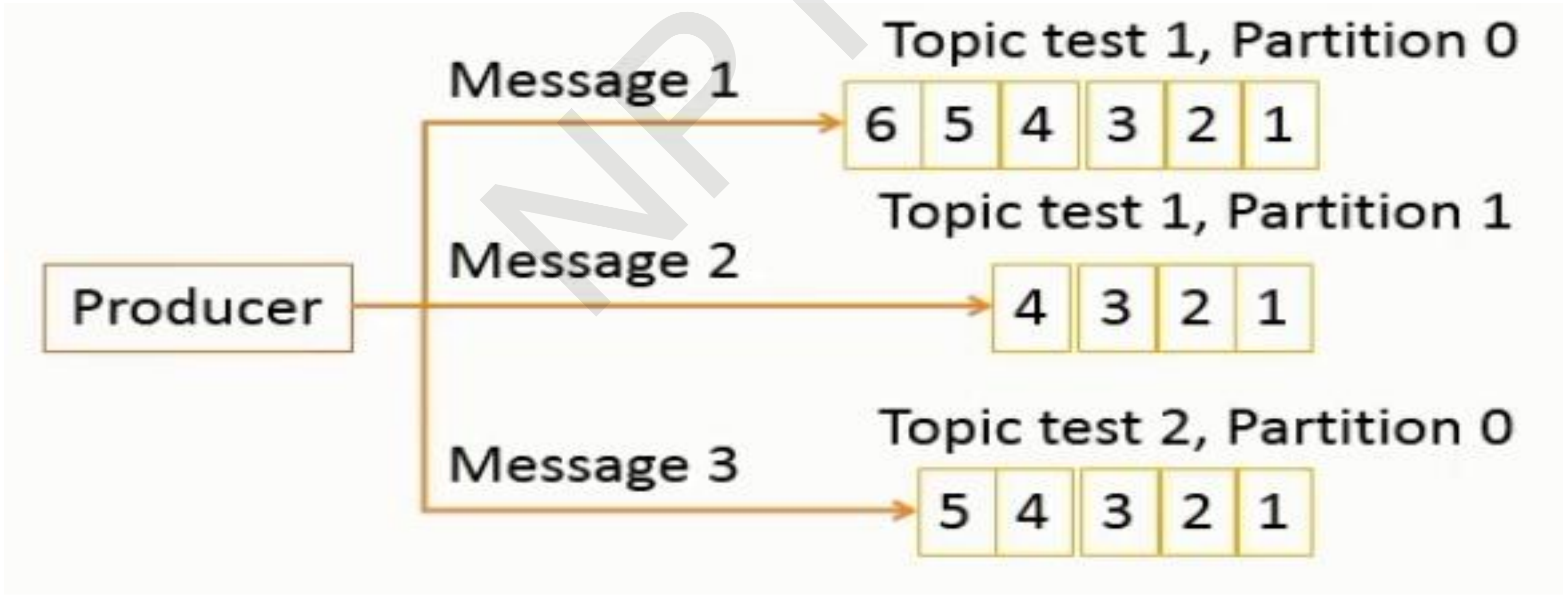
- Partitions can be distributed across the Kafka cluster.
- Each Kafka server may handle one or more partitions.
- A partition can be replicated across several servers fro fault-tolerance.
- One server is marked as a leader for the partition and the others are marked as followers.
- The leader controls the read and write for the partition, whereas, the followers replicate the data.
- If a leader fails, one of the followers automatically become the leader.
- Zookeeper is used for the leader selection.



Producers

The producer is the creator of the message in Kafka.

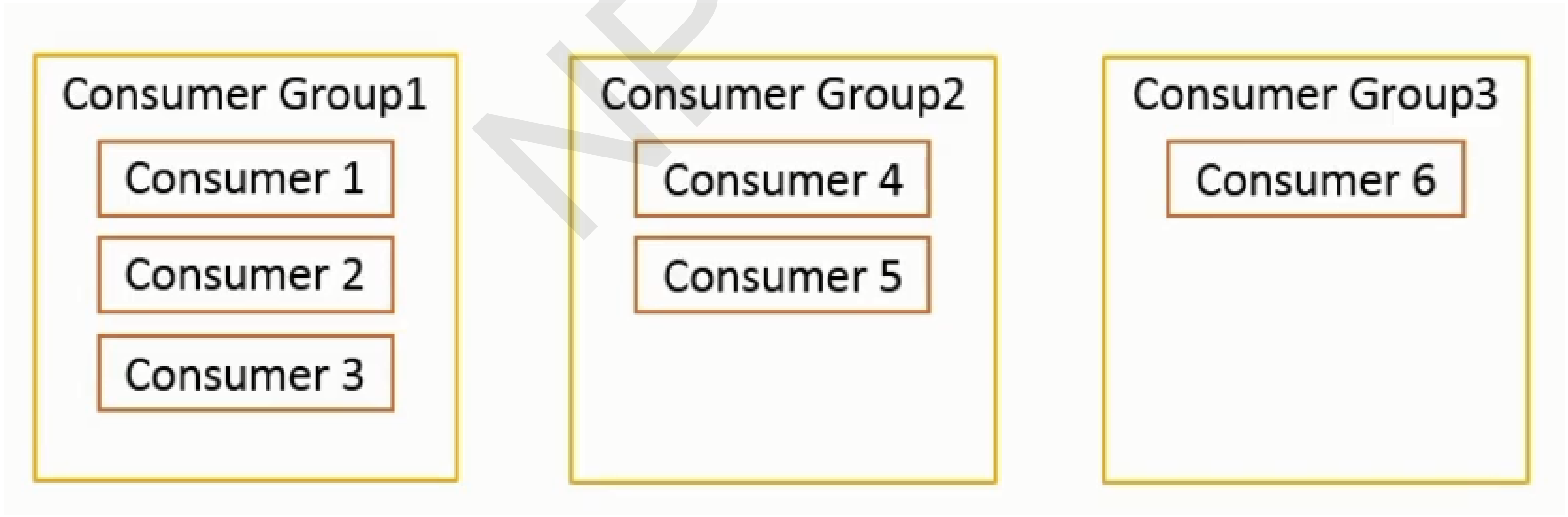
- The producers place the message to a particular topic.
- The producers also decide which partition to place the message into.
- Topics should already exist before a message is placed by the producer.
- Messages are added at one end of the partition.


Producer
Message 1
Message 2
Message 3
Topic test 1, Partition 0
6 5 4 3 2 1
Topic test 1, Partition 1
4 3 2 1
Topic test 2, Partition 0
5 4 3 2 1

Consumers

The consumer is the receiver of the message in Kafka.

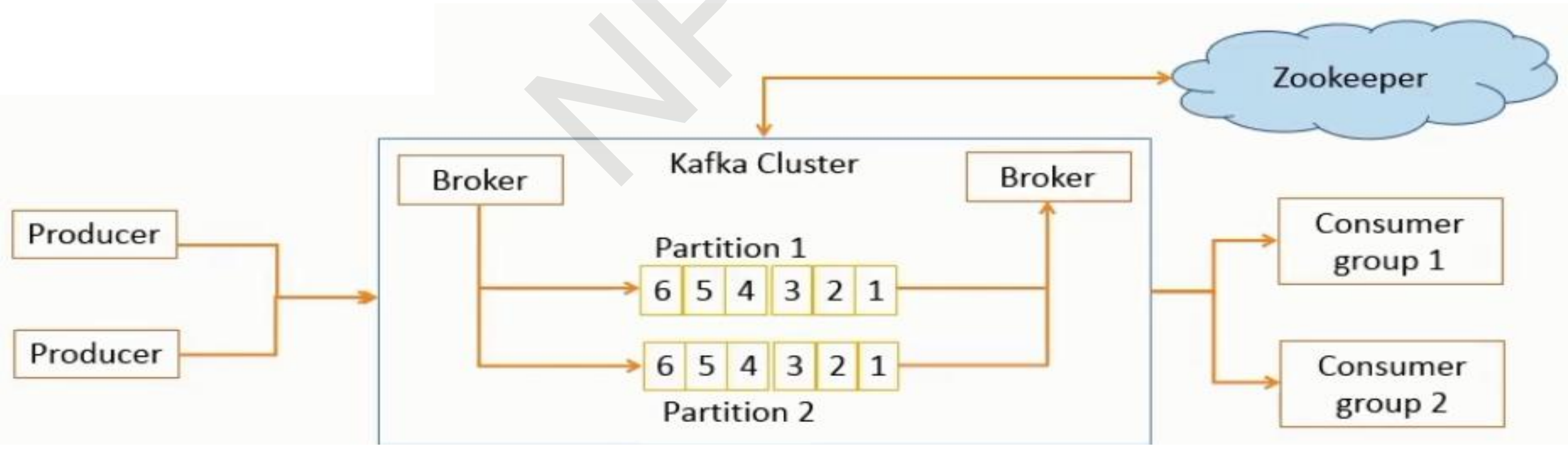
- Each consumer belongs to a consumer group.
- A consumer group may have one or more consumers.
- The consumers specify what topics they want to listen to.
- A message is sent to all the consumers in a consumer group.
- The consumer groups are used to control the messaging system.



Kafka Architecture

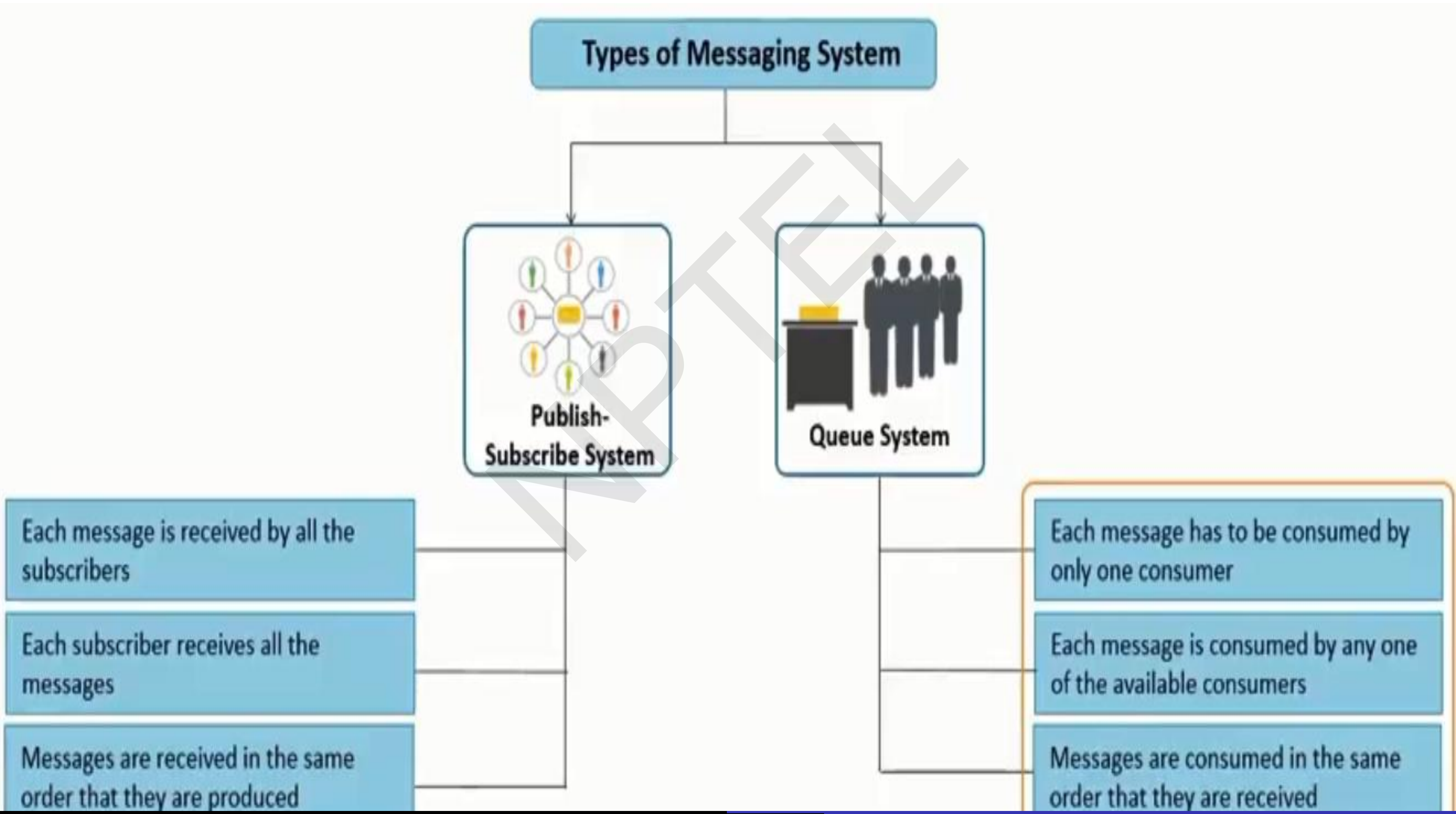
Kafka architecture consists of brokers that take messages from the producers and add to a partition of a topic. Brokers provide the messages to the consumers from the partitions.

- A topic is divided into multiple partitions.
- The messages are added to the partitions at one end and consumed in the same order.
- Each partition acts as a message queue.
- Consumers are divided into consumer groups.

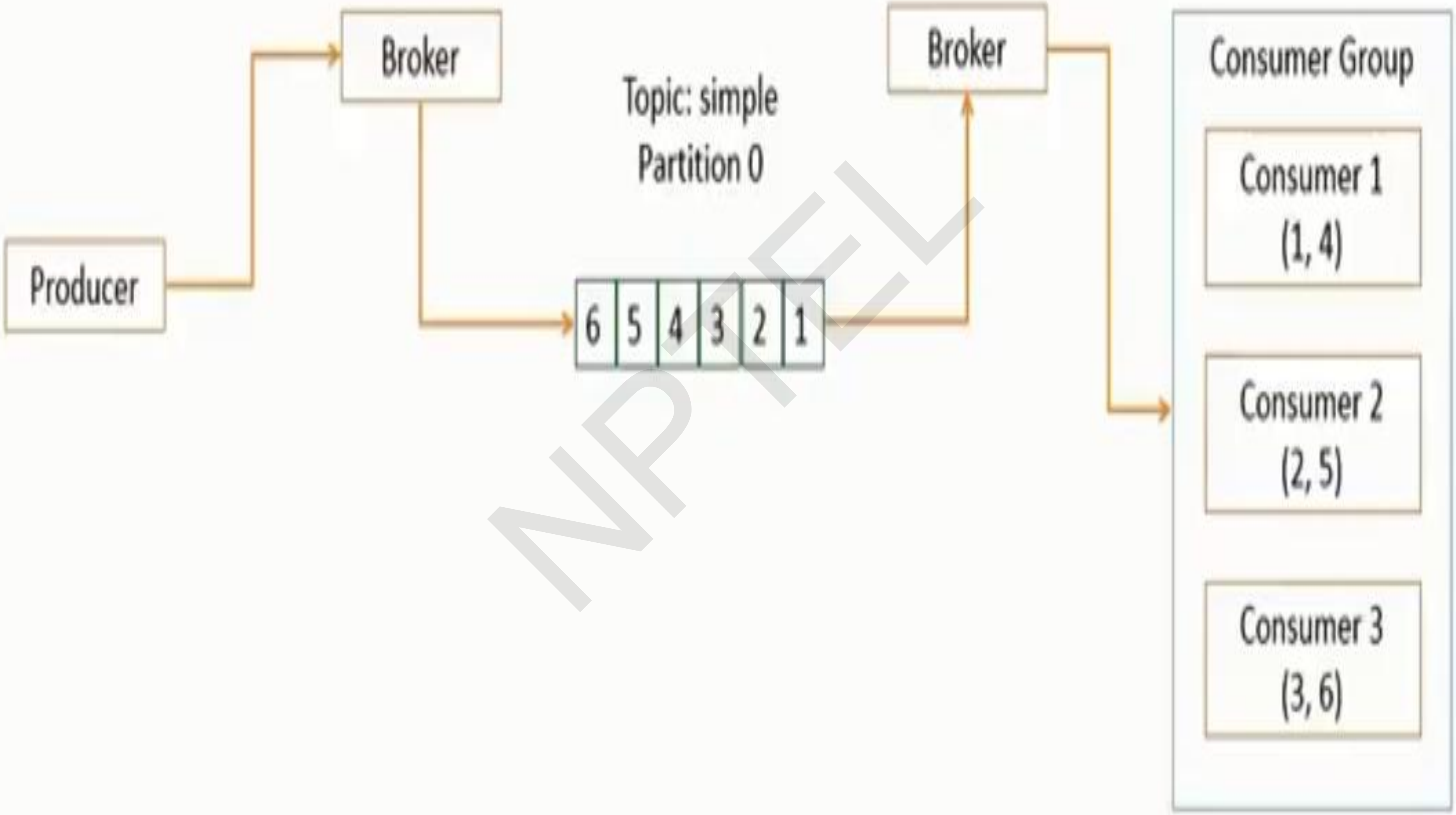


Types of Messaging Systems

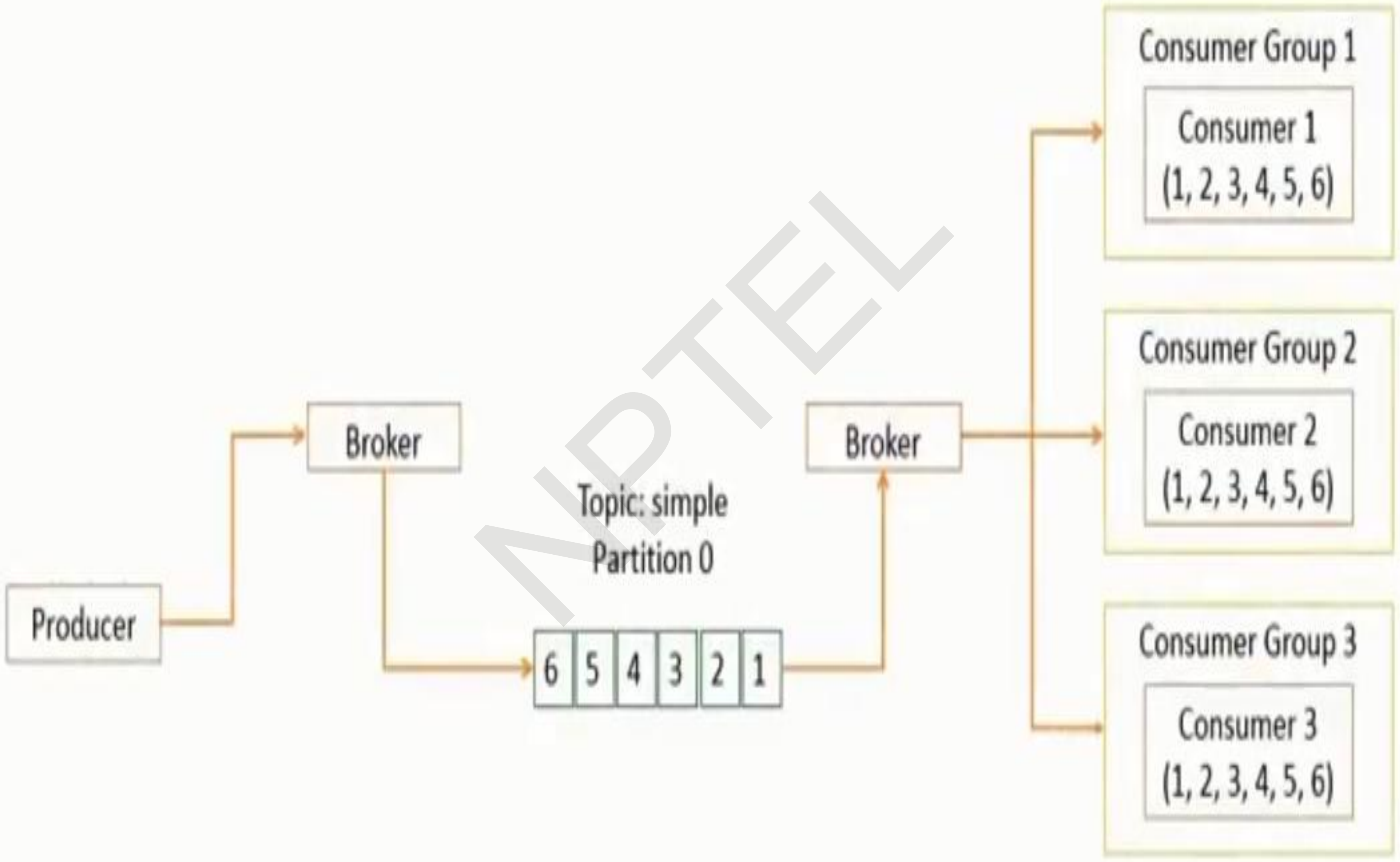
Kafka architecture supports the publish-subscribe and queue system.


Types of Messaging System
Publish-Subscribe System
Queue System
Each message is received by all the subscribers
Each subscriber receives all the messages
Messages are received in the same order that they are produced
Each message has to be consumed by only one consumer
Each message is consumed by any one of the available consumers
Messages are consumed in the same order that they are received

Example: Queue System



Example: Publish-Subscribe System



Brokers

Brokers are the Kafka processes that process the messages in Kafka.

- Each machine in the cluster can run one broker.
- They coordinate among each other using Zookeeper.
- One broker acts as a leader for a partition and handles the delivery and persistence, where as, the others act as followers.

Kafka Guarantees

- Kafka guarantees the following:

1. **Messages sent by a producer to a topic and a partition are appended in the same order**

2. **A consumer instance gets the messages in the same order as they are produced.**

3. **A topic with replication factor N, tolerates upto N-1 server failures.**

Replication in Kafka

Kafka uses the primary-backup method of replication.

- One machine (one replica) is called a leader and is chosen as the primary; the remaining machines (replicas) are chosen as the followers and act as backups.
- The leader propagates the writes to the followers.
- The leader waits until the writes are completed on all the replicas.
- If a replica is down, it is skipped for the write until it comes back.
- If the leader fails, one of the followers will be chosen as the new leader; this mechanism can tolerate n-1 failures if the replication factor is ‘n’

Persistence in Kafka

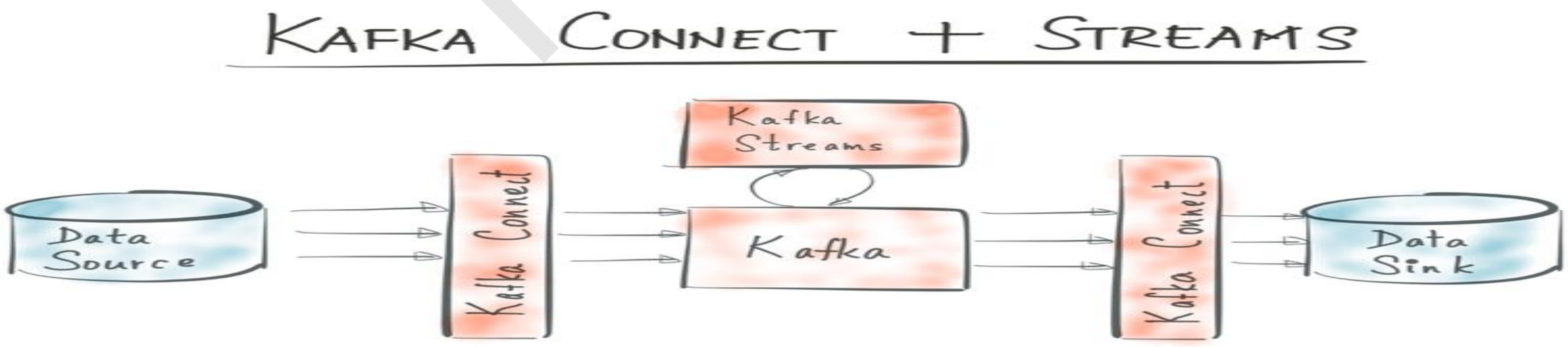
Kafka uses the Linux file system for persistence of messages

- Persistence ensures no messages are lost.
- Kafka relies on the file system page cache for fast reads and writes.
- All the data is immediately written to a file in file system.
- Messages are grouped as message sets for more efficient writes.
- Message sets can be compressed to reduce network bandwidth.
- A standardized binary message format is used among producers, brokers, and consumers to minimize data modification.

Apache Kafka: a Streaming Data Platform

- **Apache Kafka** is an open source streaming data platform (a new category of software!) with **3 major components**:

  1. **Kafka Core**: A **central hub** to **transport** and **store** event streams in real-time.
  2. **Kafka Connect**: A **framework** to **import** event streams from other source data systems into Kafka and **export** event streams from **Kafka** to destination data systems.
  3. **Kafka Streams**: A **Java library** to **process** event streams live as they occur.



Further Learning

- **Kafka Streams code examples**
  - Apache Kafka https://github.com/apache/kafka/tree/trunk/streams/examples/src/main/java/org/apache/kafka/streams/examples
  - Confluent https://github.com/confluentinc/examples/tree/master/kafka-streams
- **Source Code** https://github.com/apache/kafka/tree/trunk/streams
- **Kafka Streams Java docs** http://docs.confluent.io/current/streams/javadocs/index.html
- **First book on Kafka Streams (MEAP)**
  - Kafka Streams in Action https://www.manning.com/books/kafka-streams-in-action
- **Kafka Streams download**
  - Apache Kafka https://kafka.apache.org/downloads
  - Confluent Platform http://www.confluent.io/download

Conclusion

- Kafka is a high-performance, real-time messaging system.
- Kafka can be used as an external commit log for distributed systems.
- Kafka data model consists of messages and topics.
- Kafka architecture consists of brokers that take messages from the producers and add to a partition of a topics.
- Kafka architecture supports two types of messaging system called publish-subscribe and queue system.
- Brokers are the Kafka processes that process the messages in Kafka.