Design of HBase

Dr. Rajiv Misra
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:
  - **What is HBase?**
  - **HBase Architecture**
  - **HBase Components**
  - **Data model**
  - **HBase Storage Hierarchy**
  - **Cross-Datacenter Replication**
  - **Auto Sharding and Distribution**
  - **Bloom Filter and Fold, Store, and Shift**

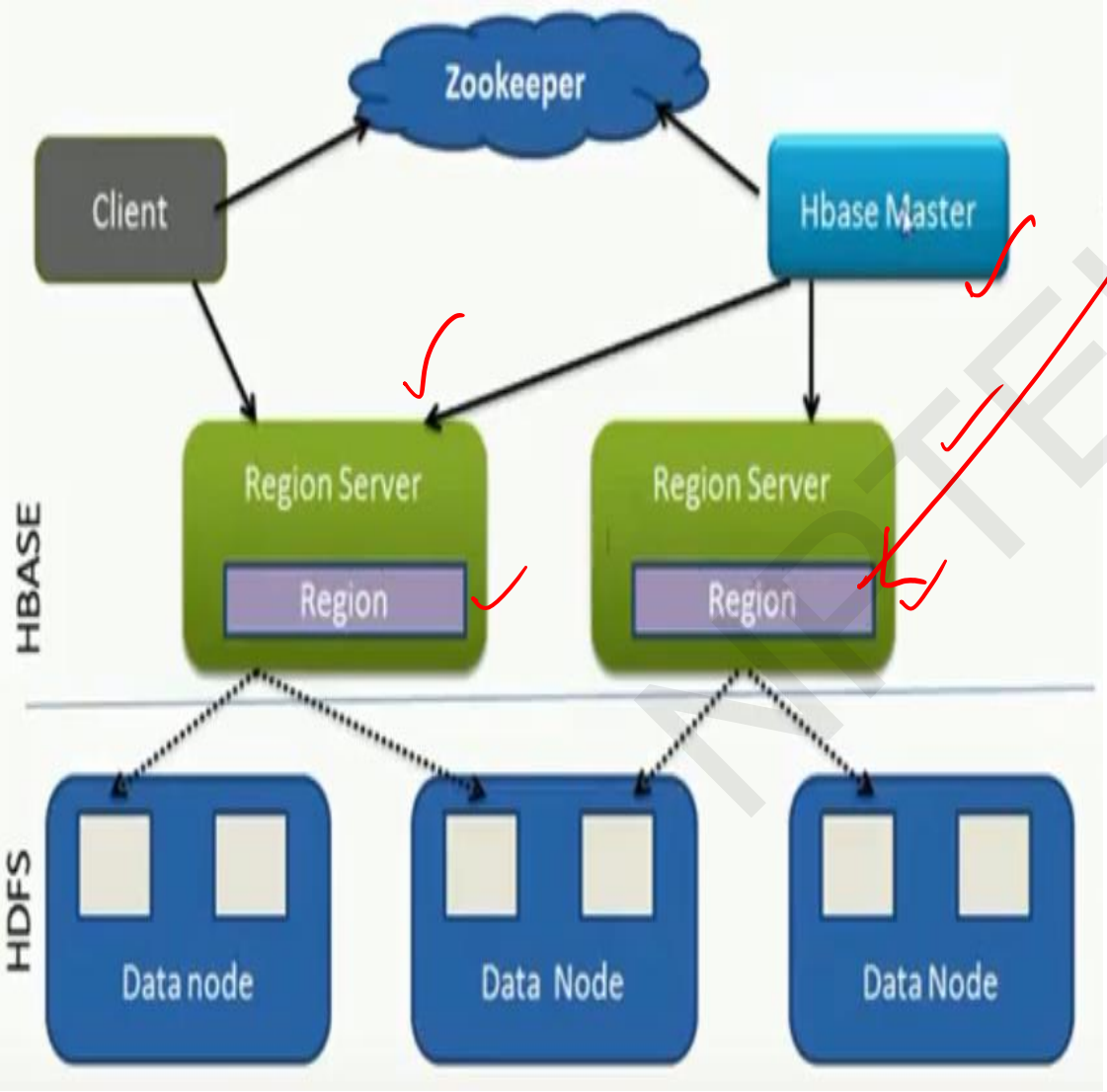
HBase is:

- **An opensource NOSQL database.**
- A distributed column-oriented data store that can scale out horizontally to 1,000s of commodity servers and petabytes of indexed storage.
- Designed to operate on top of the Hadoop distributed file system (HDFS) for scalability, fault tolerance, and high availability.
- Hbase is actually an implementation of the **BigTable** storage architecture, which is a distributed storage system **developed by Google**.
- Works with structured, unstructured and semi-structured data.

HBase

- **Google's BigTable** was first "blob-based" storage system
- **Yahoo!** Open-sourced it → HBase
- Major Apache project today
- **Facebook** uses HBase internally
- **API functions**
  - Get/Put(row)
  - Scan(row range, filter) – range queries
  - MultiPut
- Unlike Cassandra, HBase prefers consistency (over availability)

HBase Architecture



- Table Split into **regions** and served by region servers.
- Regions vertically divided by column families into "**stores**".
- Stores saved as files on HDFS.
- Hbase utilizes zookeeper for distributed coordination.

HBase Components

- **Client:**

  Finds RegionServers that are serving particular row range of interest

- **HMaster:**

  Monitoring all RegionServer instances in the cluster

- **Regions:**

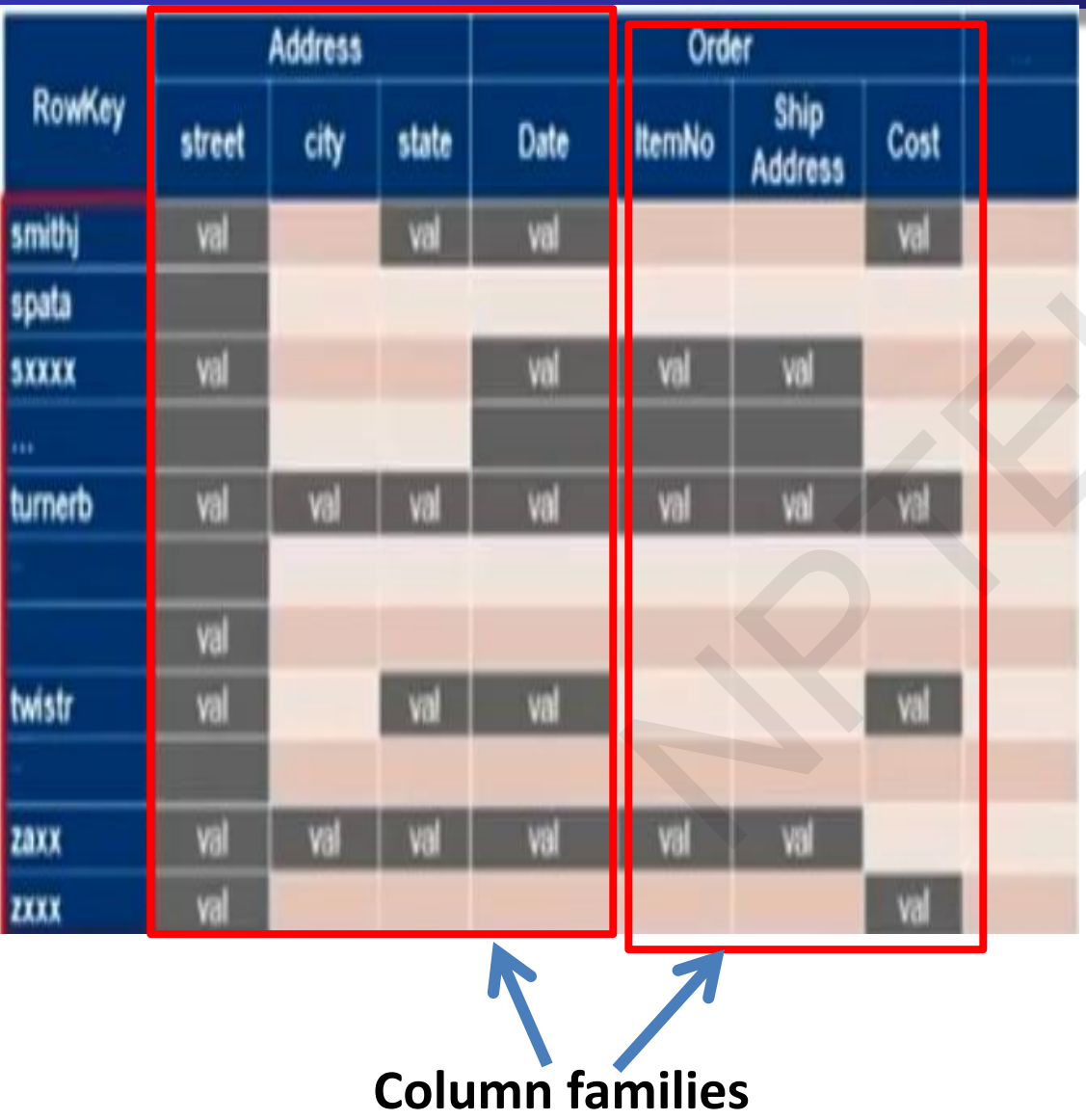
  Basic element of availability and distribution for tables

- **RegionServer:**

  Serving and managing regions

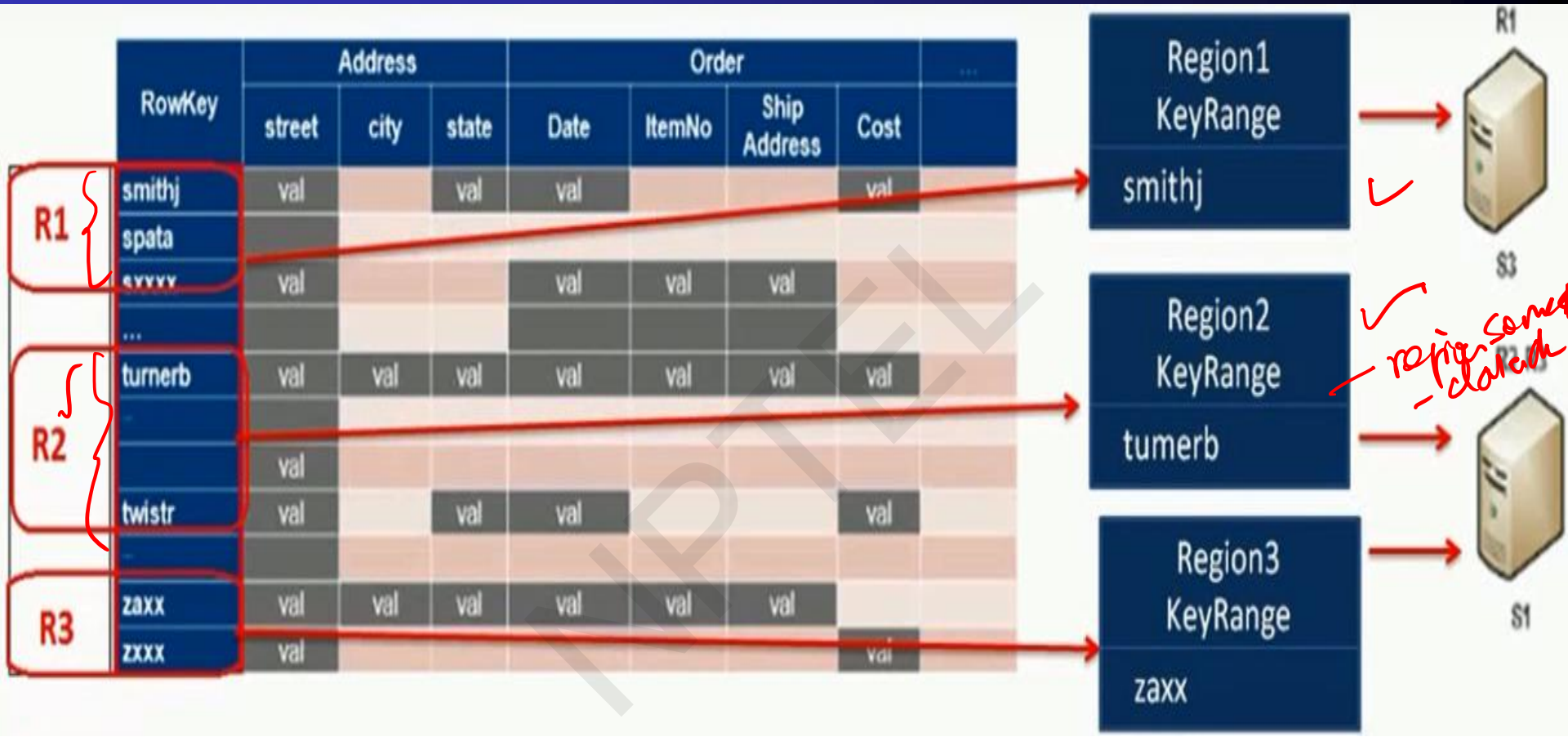
  In a distributed cluster, a RegionServer runs on a DataNode

Data Model



- Data stored in Hbase is located by its "**rowkey**"
- RowKey is like a primary key from a rational database.
- Records in Hbase are stored in sorted order, according to rowkey.
- Data in a row are grouped together as Column Families. Each Column Family has one or more Columns
- These Columns in a family are stored together in a low level storage file known as **HFile**

HBase Components



- Tables are divided into sequences of rows, by key range, called **regions.**
- These regions are then assigned to the data nodes in the cluster called **"RegionServers."**

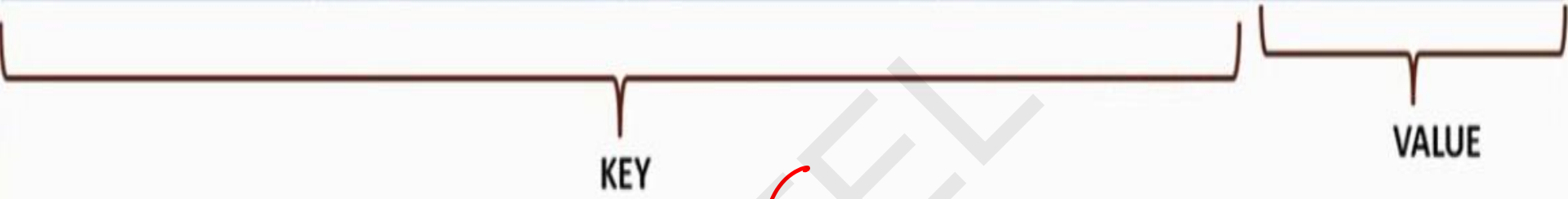
Column Family

Row Key	Personal Data		ProfessionalData	
Emp Id	Name	City	Designation	Salary
101	John	Mumbai	Manager	10L
102	Geetha	New Delhi	Sr.Software Engineer	8L
103	Smita	Pune	Programmer Analyst	4L
104	Ankit	Bangalore	Data Analyst	12L

- A column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon.ex-personaldata:Name
- Column families are mapped to storage files and are stored in separate files, which can also be accesses separately.

Cell in HBase Table

Row Key	Column Family	Column Qualifier	Timestamp	Value
John	PersonalData	City	123456790123	Mumbai



- Data is stored in HBASE tables Cells.
- **Cell is a combination of row, column family, column qualifier and contains a value and a timestamp**
- The key consists of the row key, column name, and timestamp.
- The entire cell, with the added structural information, is called **Key Value.**

HBase Data Model

- **Table:** Hbase organizes data into tables. Table names are Strings and composed of characters that are safe for use in a file system path.

- **Row:** Within a table, data is stored according to its row. Rows are identified uniquely by their row key. Row keys do not have a data type and are always treated as a byte[ ] (byte array).

- **Column Family:** Data within a row is grouped by column family. Every row in a table has the same column families, although a row need not store data in all its families. Column families are Strings and composed of characters that are safe for use in a file system path.

HBase Data Model

- **Column Qualifier:** Data within a column family is addressed via its column qualifier, or simply , column, Column qualifiers need not be specified in advance. Column qualifiers need not be consistent between rows. Like row keys, column qualifiers do not have a data type and are always treated as a byte[ ].

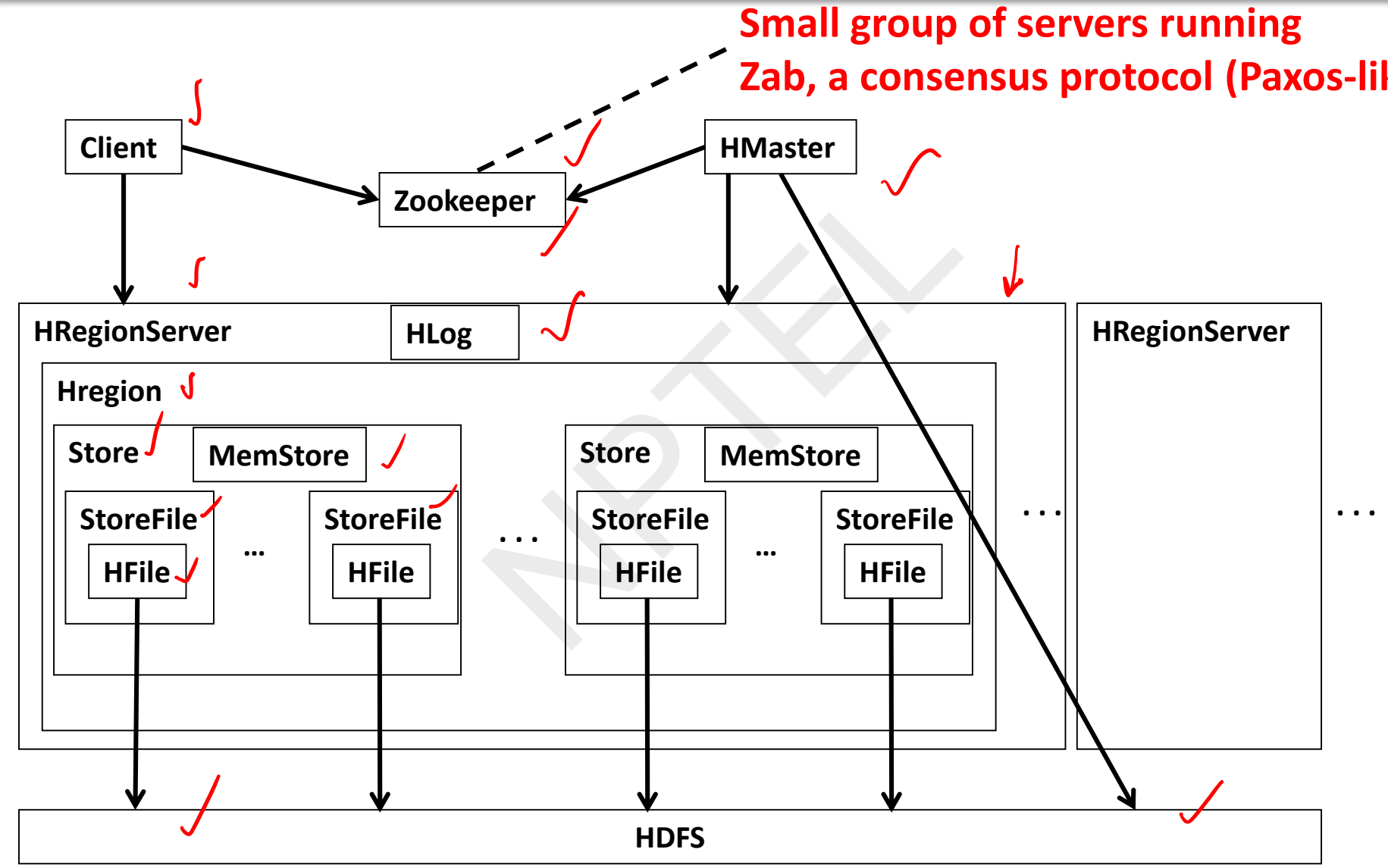
- **Cell:** A combination of row key, column family, and column qualifier uniquely identifies a cell. The data stored in a cell is referred to as that cell's value.

HBase Data Model

- **Timestamp:** Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp is used.

- If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by Hbase is configured for each column family. The default number of cell versions is three.

HBase Architecture

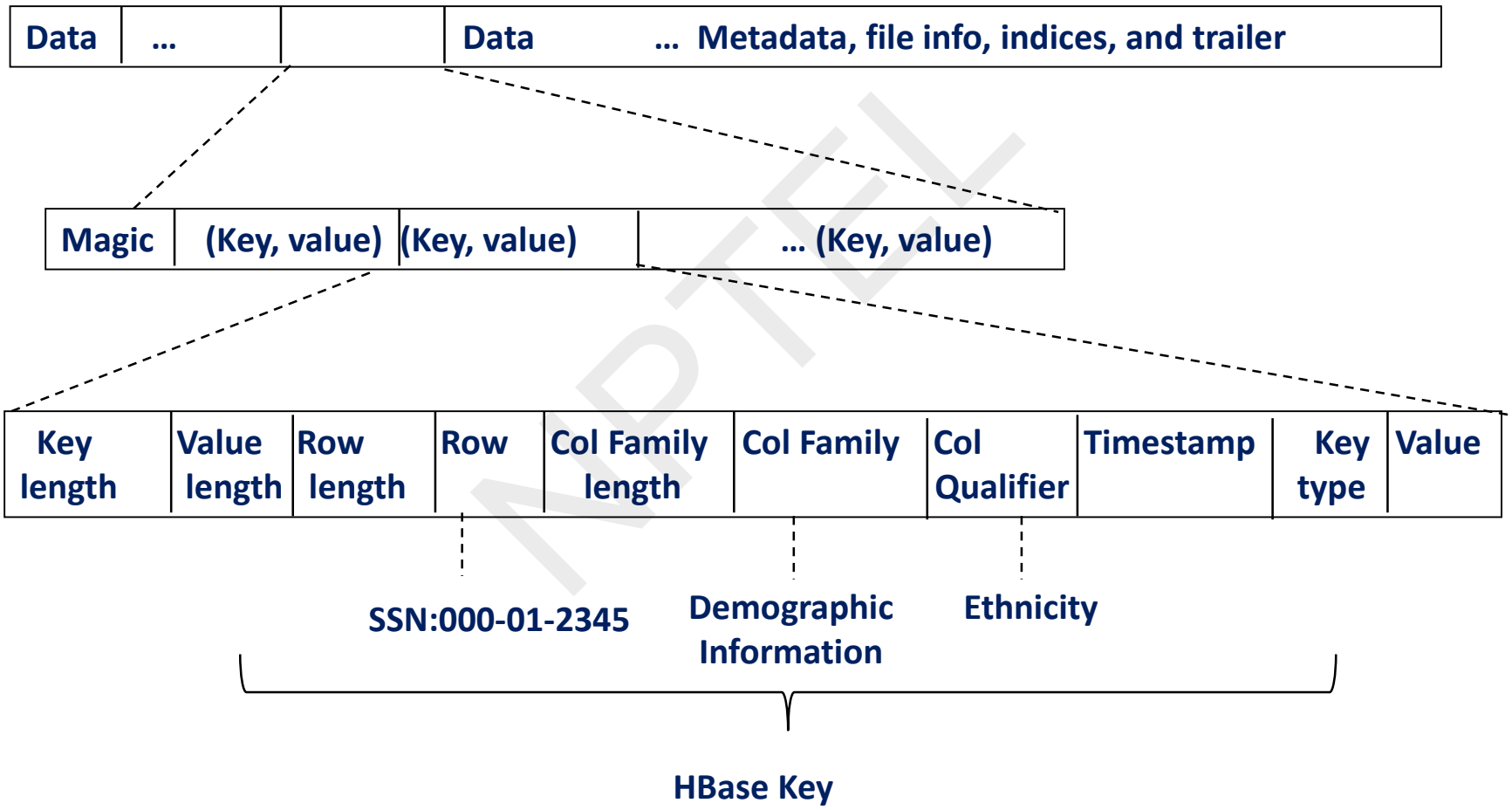


HBase Storage Hierarchy

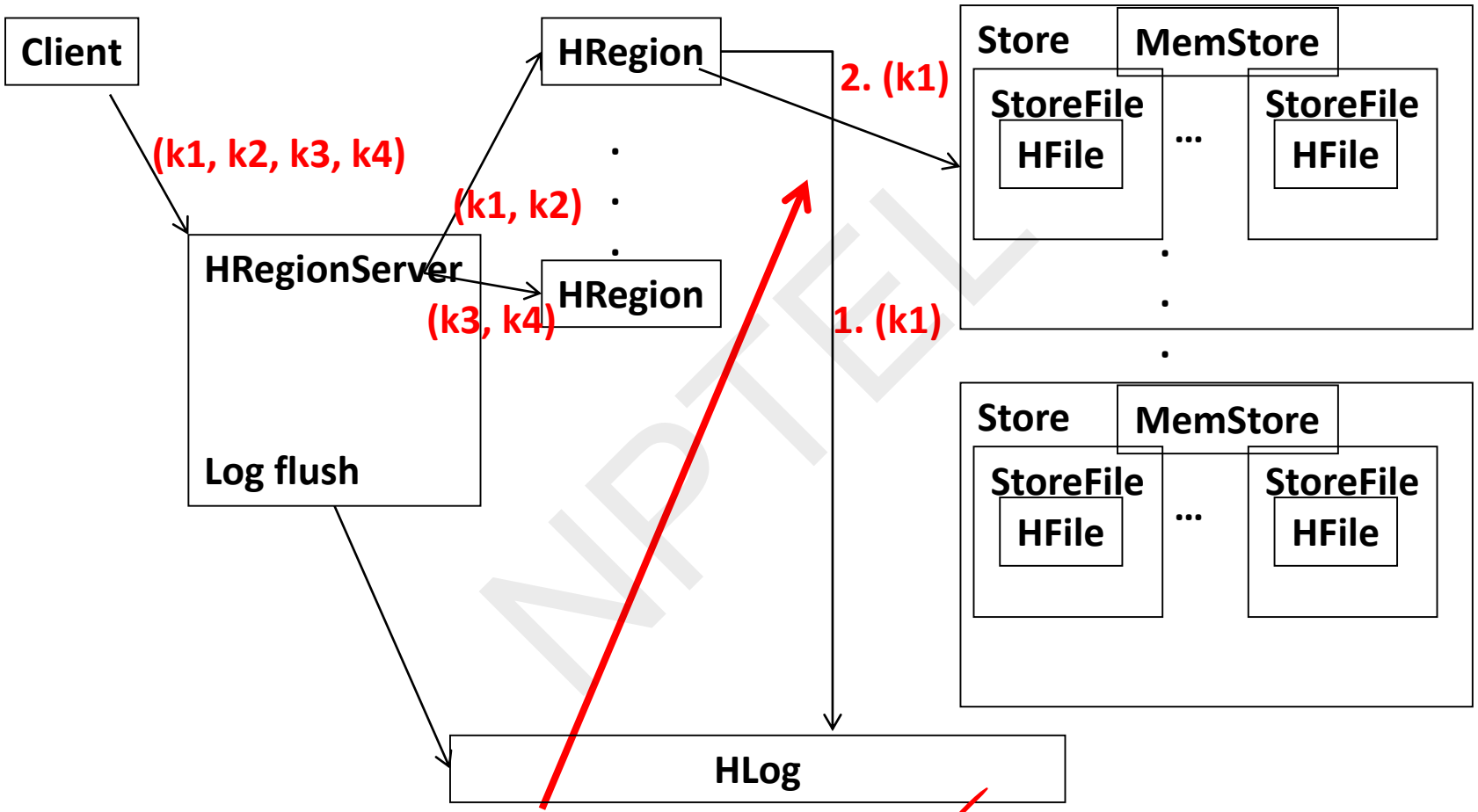
- **HBase Table**
  - Split it into multiple **regions**: replicated across servers
    - ColumnFamily = subset of columns with similar query patterns
    - One **Store** per combination of ColumnFamily + region
      - **Memstore** for each Store: in-memory updates to Store; flushed to disk when full
        - » **StoreFiles** for each store for each region: where the data lives
          - **Hfile**

- **HFile**
  - SSTable from Google's BigTable

HFile


Data
...
Data
... Metadata, file info, indices, and trailer
Magic
(Key, value)
(Key, value)
... (Key, value)
Key length
Value length
Row length
Row
Col Family length
Col Family
Col Qualifier
Timestamp
Key type
Value
SSN:000-01-2345
Demographic Information
Ethnicity
HBase Key

Strong Consistency: HBase Write-Ahead Log



Write to HLog before writing to MemStore ✓

Helps recover from failure by replaying Hlog.

Log Replay

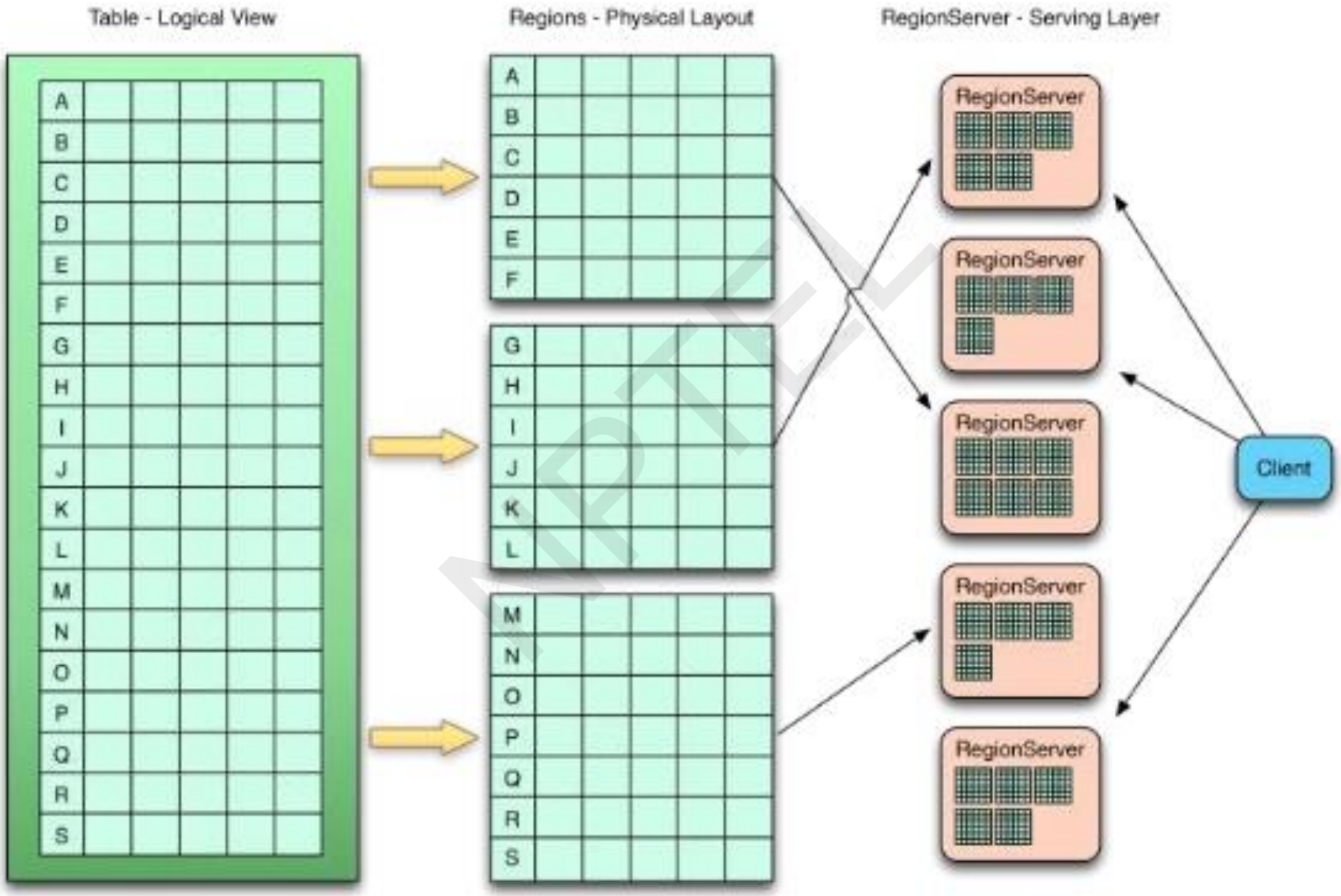
- **After recovery from failure, or upon bootup (HRegionServer/HMaster)**
  - Replay any stale logs (use timestamps to find out where the database is with respect to the logs)
  - Replay: add edits to the MemStore

Cross-Datacenter Replication

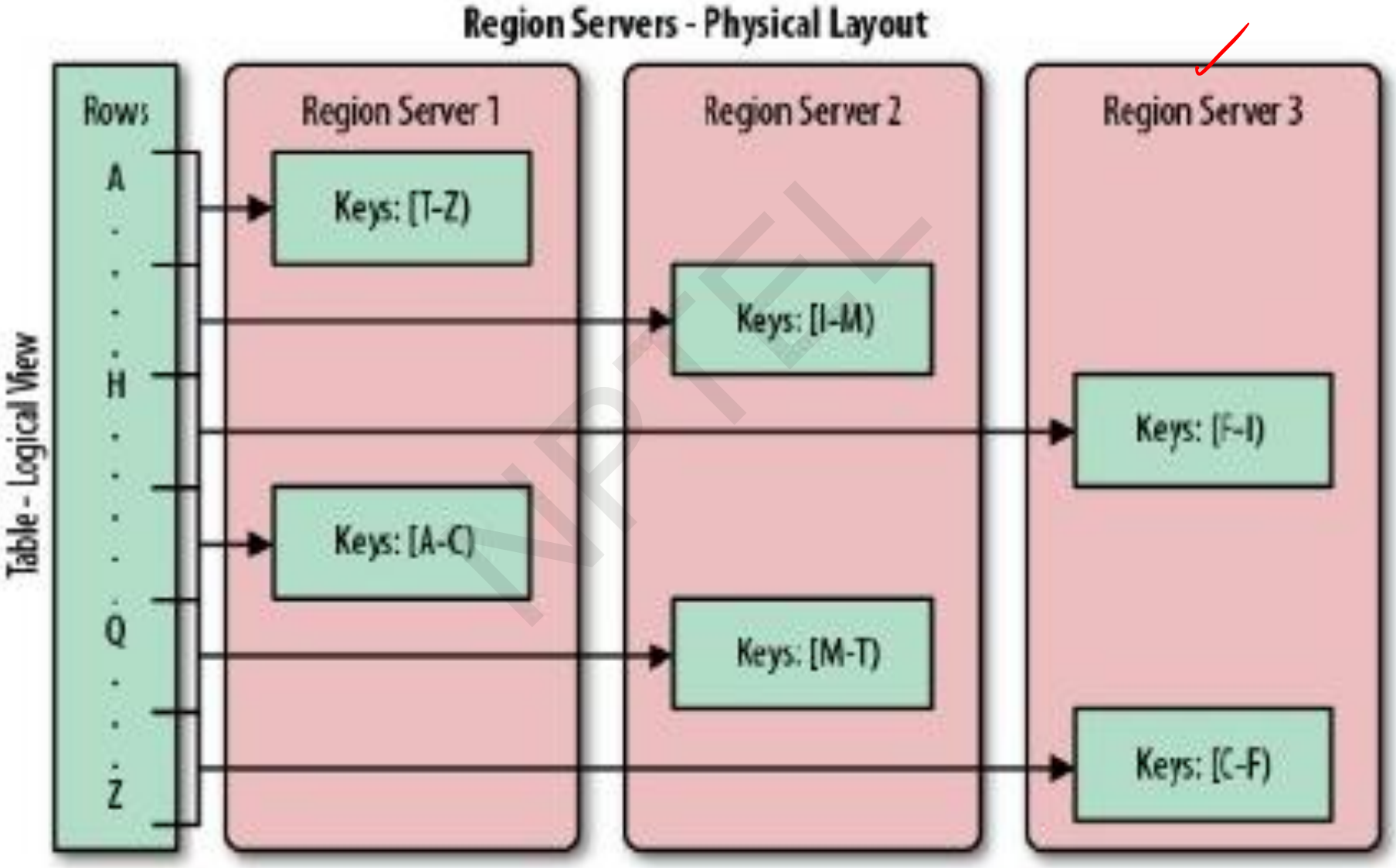
- Single "**Master**" cluster
- Other "**Slave**" clusters replicate the same tables
- Master cluster synchronously sends HLogs over to slave clusters
- Coordination among clusters is via Zookeeper
- Zookeeper can be used like a file system to store control information

1. */hbase/replication/state*
2. */hbase/replication/peers/<peer cluster number>*
3. */hbase/replication/rs/<hlog>*

Auto Sharding



Distribution



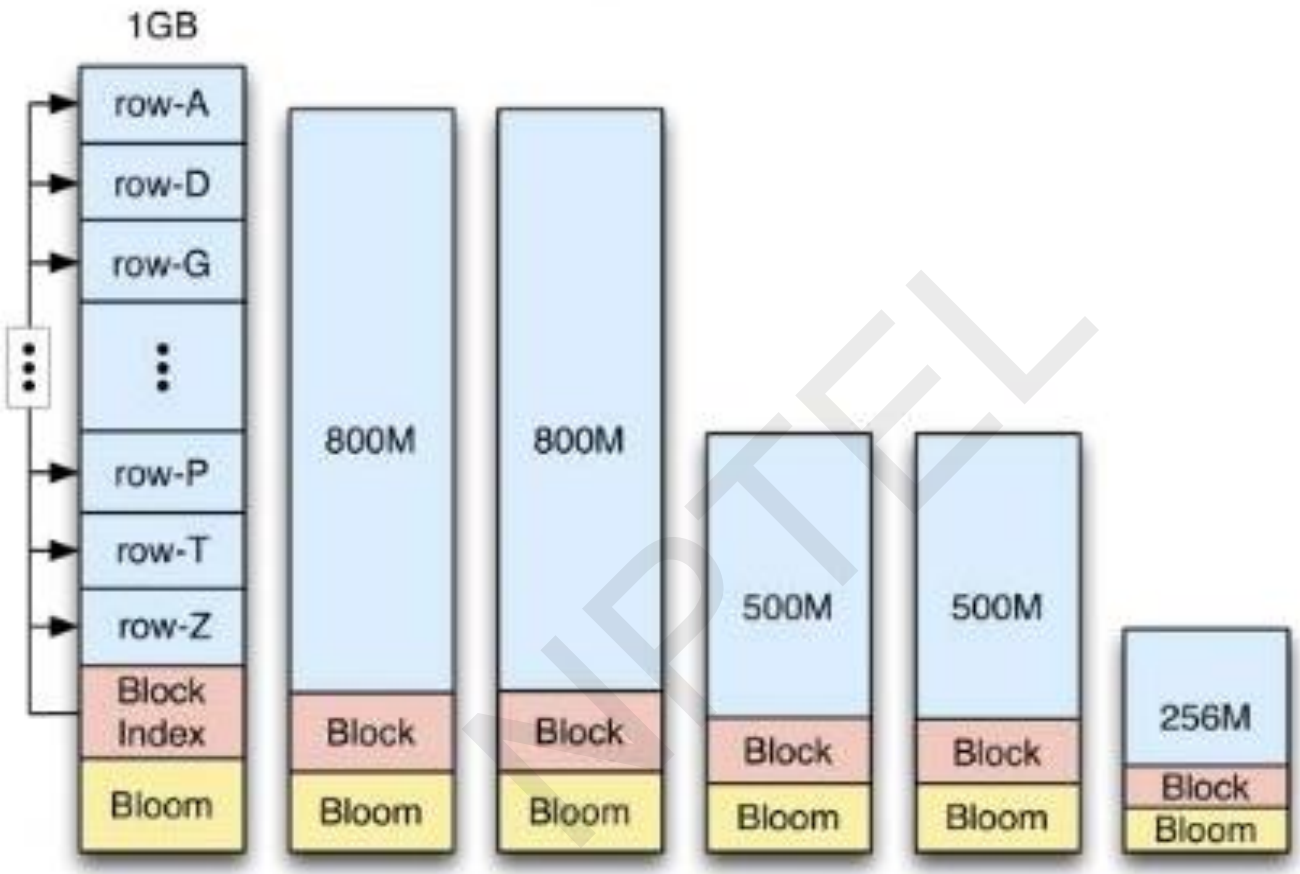
Auto Sharding and Distribution

- Unit of scalability in HBase is the Region
- Sorted, contiguous range of rows
- Spread “randomly” across RegionServer
- Moved around for load balancing and failover
- Split automatically or manually to scale with growing data
- Capacity is solely a factor of cluster nodes vs. Regions per node

Bloom Filter

- Bloom Filters are generated when HFile is persisted
  - Stored at the end of each HFile
  - Loaded into memory
- Allows check on row + column level
- Can filter entire store files from reads
  - Useful when data is grouped
- Also useful when many misses are expected during reads (non existing keys)

Bloom Filter

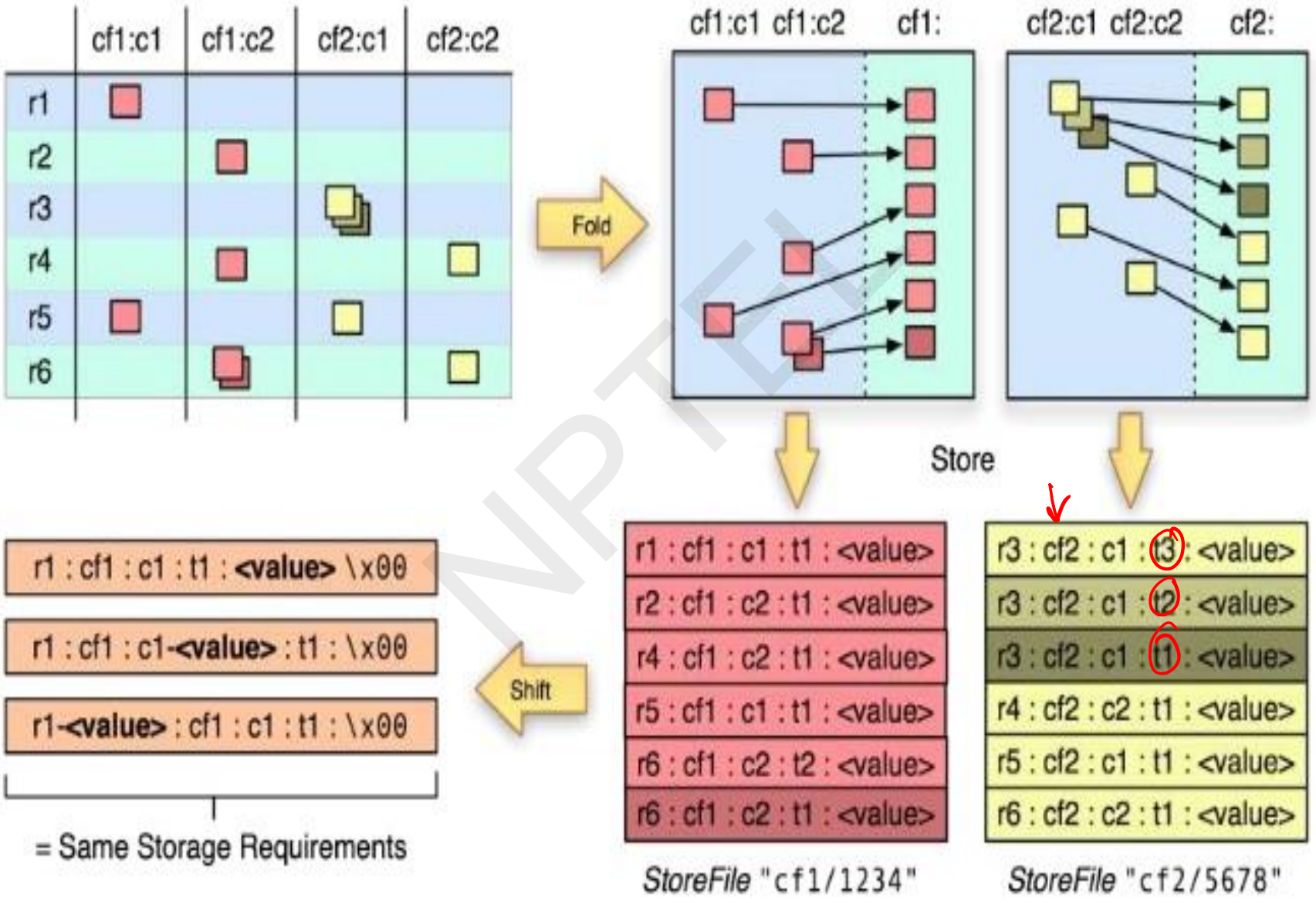


bottom of HFile

Block Index	Yes	Yes	Yes	Yes	Yes	Yes	→ 6 seeks/blocks loaded
Bloom Filter	No	No	Maybe	No	No	Maybe	→ 2 seeks/blocks loaded

Question: which file has (possibly) "row-R"?

Fold, Store, and Shift



Fold, Store, and Shift

- Logical layout does not match physical one
- All values are stored with the full coordinates, including: Row Key, Column Family, Column Qualifier, and Timestamp
- Folds columns into “row per column”
- NULLs are cost free as nothing is stored
- Versions are multiple “rows” in folded table

Conclusion

- Traditional Databases (RDBMSs) work with strong consistency, and offer ACID
- Modern workloads don't need such strong guarantees, but do need fast response times (availability)
- Unfortunately, CAP theorem
- **Key-value/NoSQL systems offer BASE**
  - Eventual consistency, and a variety of other consistency models striving towards strong consistency
- **In this lecture, we have discussed:**
    - HBase Architecture, HBase Components, Data model, HBase Storage Hierarchy, Cross-Datacenter Replication, Auto Sharding and Distribution, Bloom Filter and Fold, Store, and Shift

Spark Streaming and Sliding Window Analytics

Dr. Rajiv Misra
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 Real-time big data processing with Spark Streaming and Sliding Window Analytics.
- We will also discuss a case study based on Twitter Sentiment Analysis with using Streaming.

Big Streaming Data Processing

Fraud detection in bank transactions

Anomalies in sensor data

Cat videos in tweets

How to Process Big Streaming Data

- Scales to hundreds of nodes
- Achieves low latency
- Efficiently recover from failures
- Integrates with batch and interactive processing

What people have been doing?

- Build two stacks – one for batch, one for streaming
  - Often both process same data

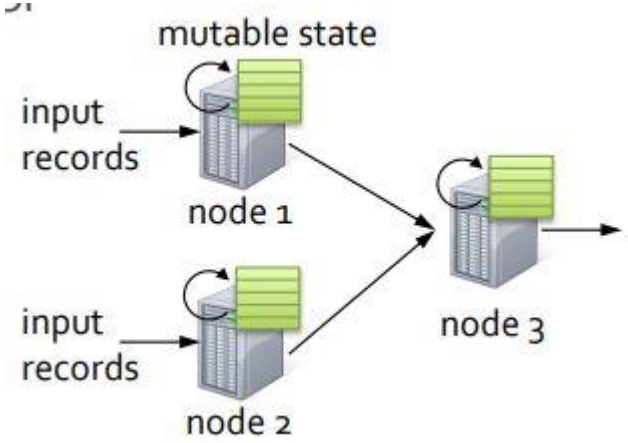
- Existing frameworks cannot do both
  - Either, stream processing of 100s of MB/s with low latency
  - Or, batch processing of TBs of data with high latency

What people have been doing?

- Extremely painful to maintain two different stacks
  - Different programming models
  - Doubles implementation effort
  - Doubles operational effort

Fault-tolerant Stream Processing

- Traditional processing model
  - Pipeline of nodes
  - Each node maintains mutable state
  - Each input record updates the state and new records are sent out



- Mutable state is lost if node fails

failure of nodes is norm rather exception in commodity hw.

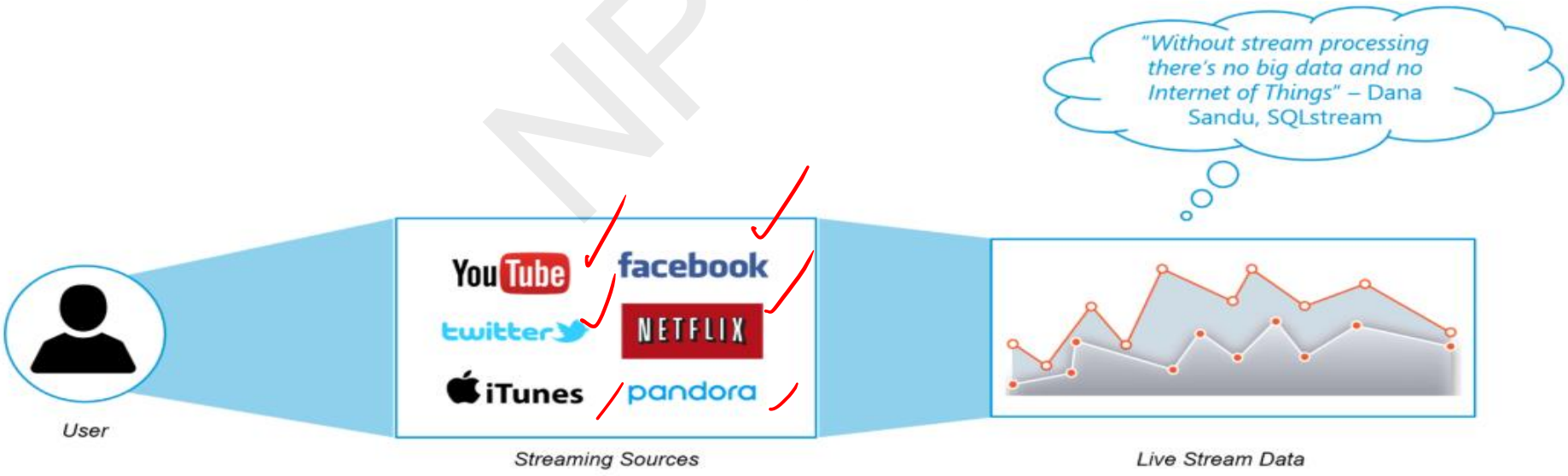
- Making stateful stream processing fault-tolerant is challenging!

Spark Streaming →

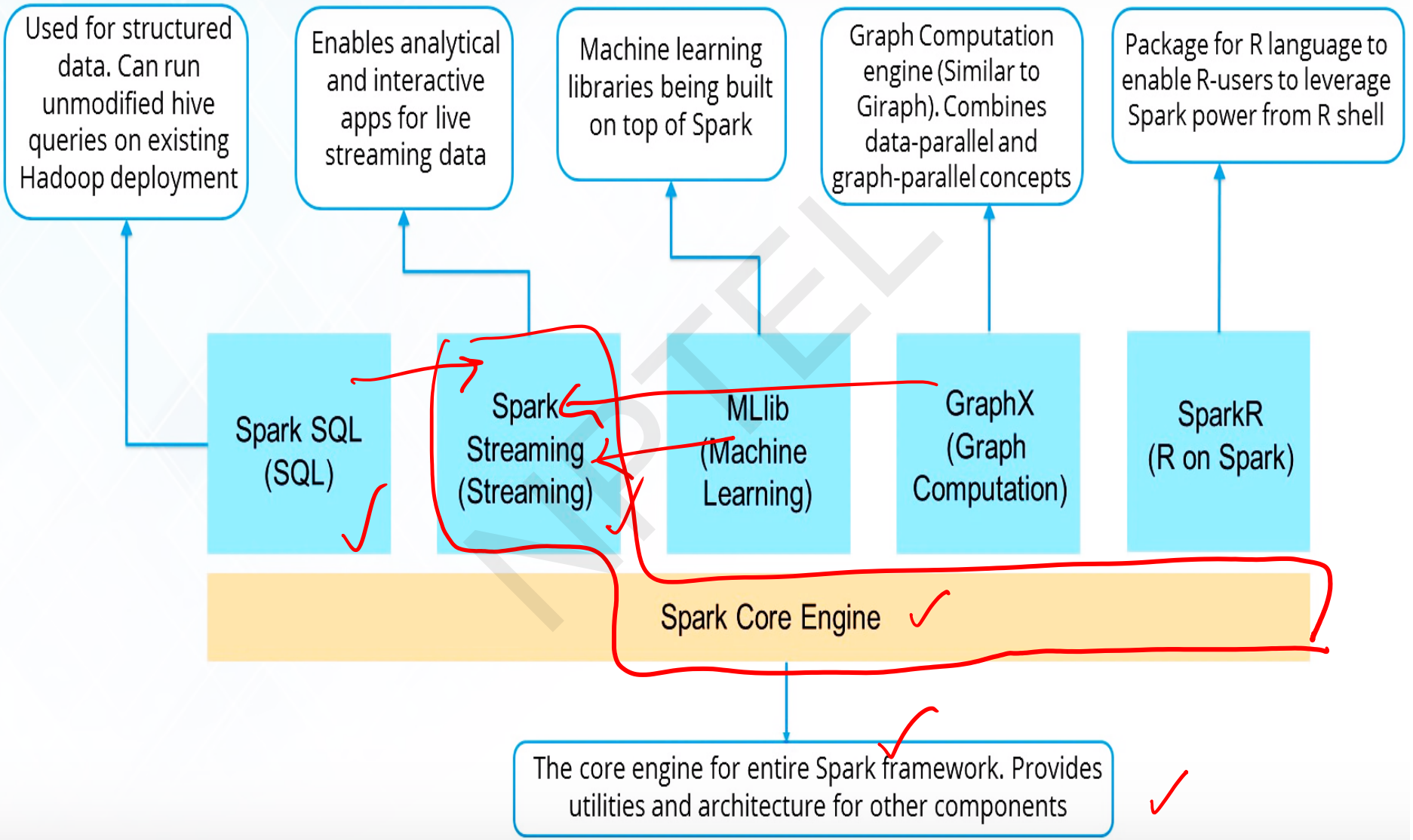
What is Streaming?

- Data Streaming is a technique for transferring data so that it can be processed as a steady and continuous stream.

- Streaming technologies are becoming increasingly important with the growth of the Internet.

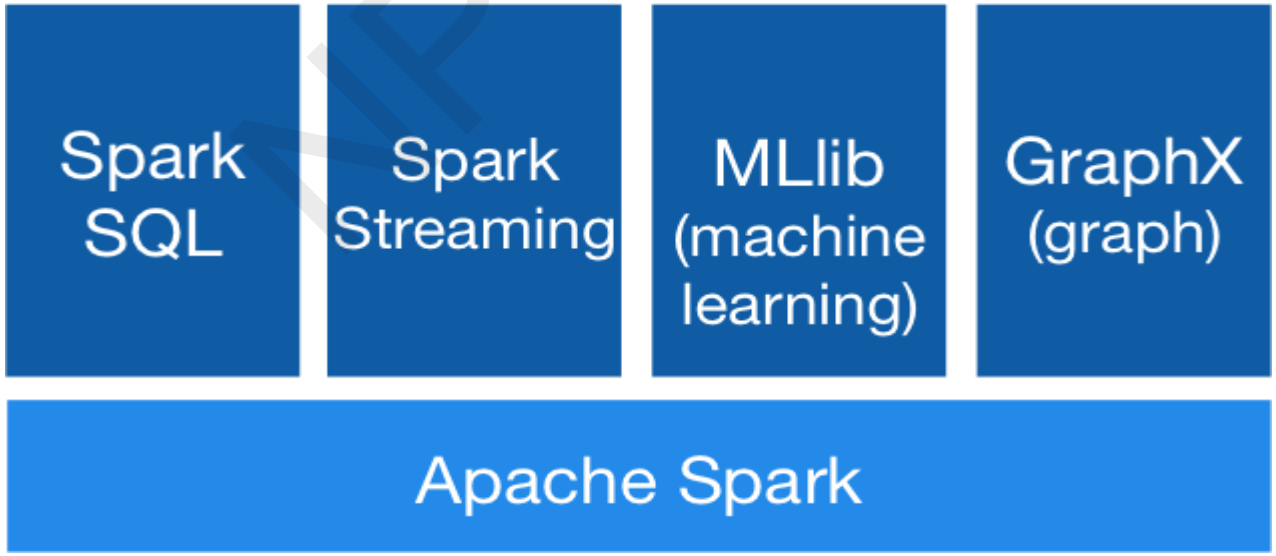


Spark Ecosystem



What is Spark Streaming?

- Extends Spark for doing big data stream processing
- Project started in early 2012, alpha released in Spring 2017 with Spark 0.7
- Moving out of alpha in Spark 0.9
- Spark Streaming has support built-in to consume from Kafka, Flume, Twitter, ZeroMQ, Kinesis, and TCP/IP sockets.
- In Spark 2.x, a separate technology based on Datasets, called Structured Streaming, that has a higher-level interface is also provided to support streaming.



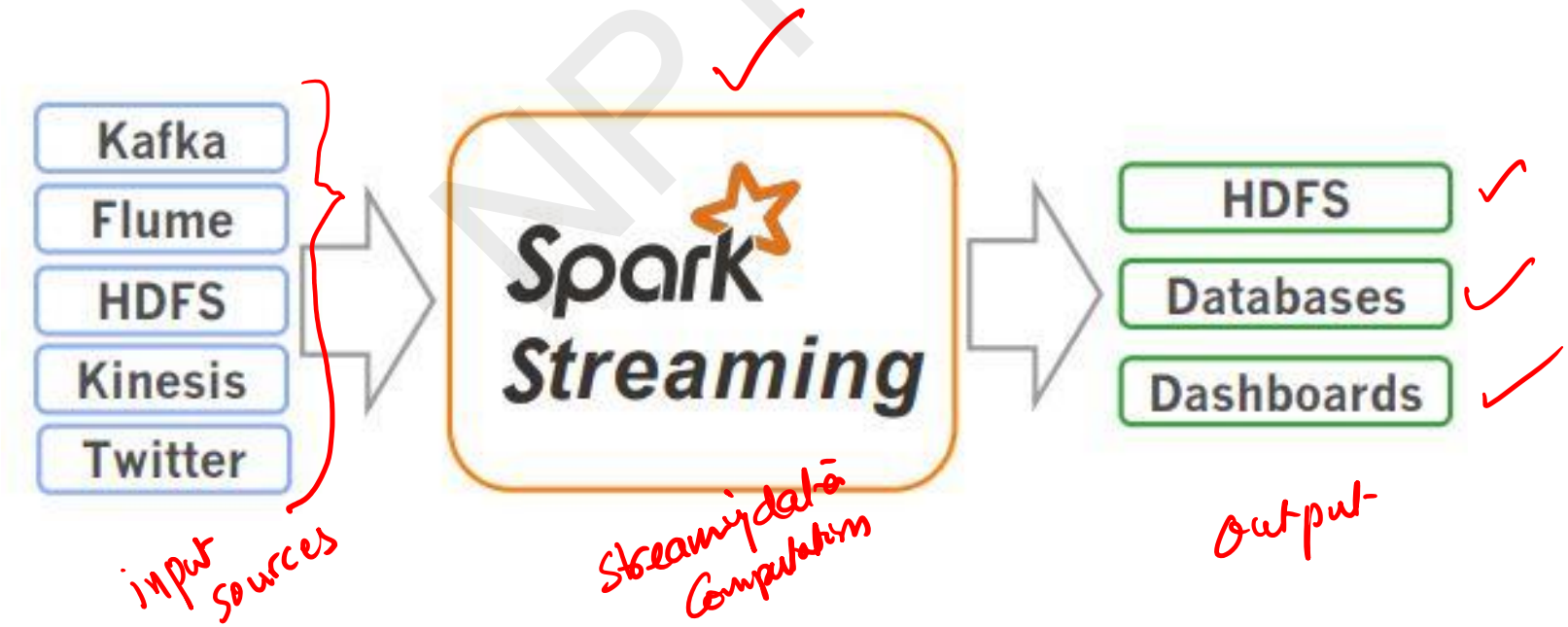
What is Spark Streaming?

- Framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve 1 second scale latencies
  - Integrates with Spark's batch and interactive processing
  - Provides a simple batch-like API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.

What is Spark Streaming?

- Receive data streams from input sources, process them in a cluster, push out to databases/ dashboards

- Scalable, fault-tolerant, "second-scale" latencies



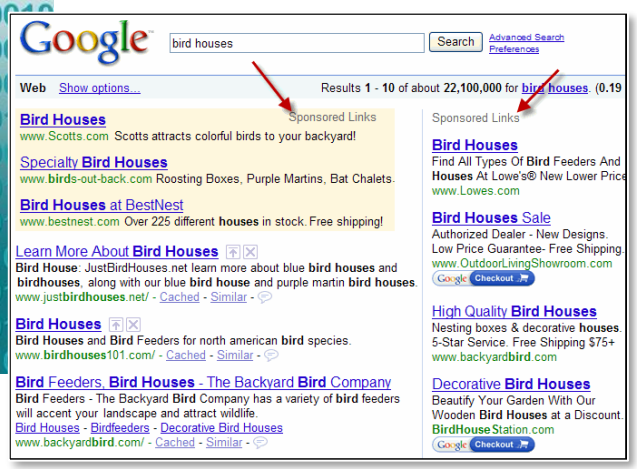
Why Spark Streaming ?

- Many big-data applications need to process large data streams in realtime

Website monitoring

Fraud detection

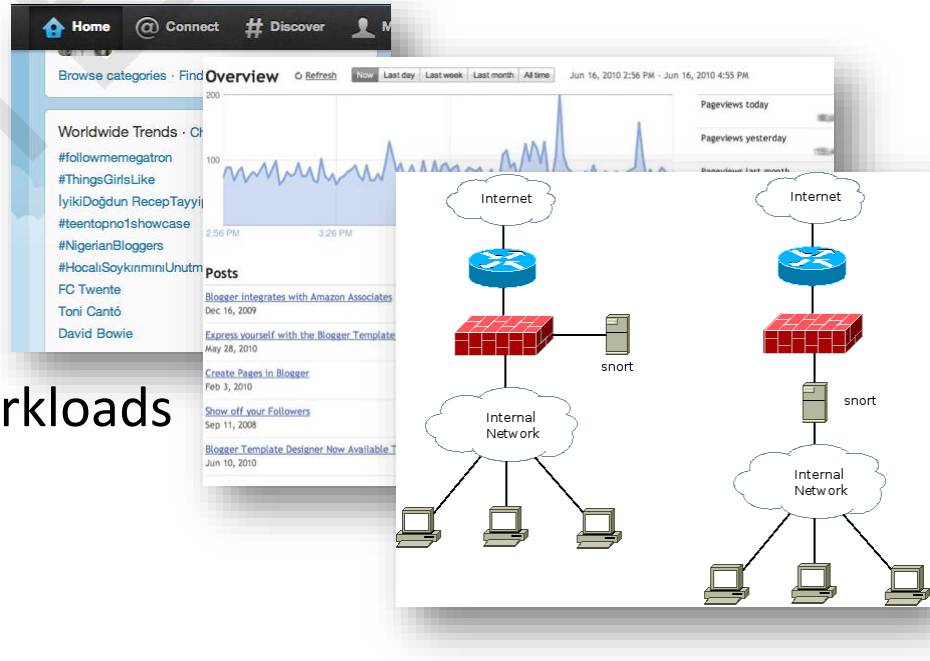
Ad monetization



Why Spark Streaming ?

- Many important applications must process large streams of live data and provide results in near-real-time

  - Social network trends
  - Website statistics
  - Intrustion detection systems
  - etc.



- Require large clusters to handle workloads

- Require latencies of few seconds

Why Spark Streaming ?

- We can use Spark Streaming to stream real-time data from various sources like Twitter, Stock Market and Geographical Systems and perform powerful analytics to help businesses.

Spark Streaming is used to stream real-time data from various sources like Twitter, Stock Market and Geographical Systems and perform powerful analytics to help businesses.

Why Spark Streaming?

Need a framework for big data stream processing that

Scales to hundreds of nodes

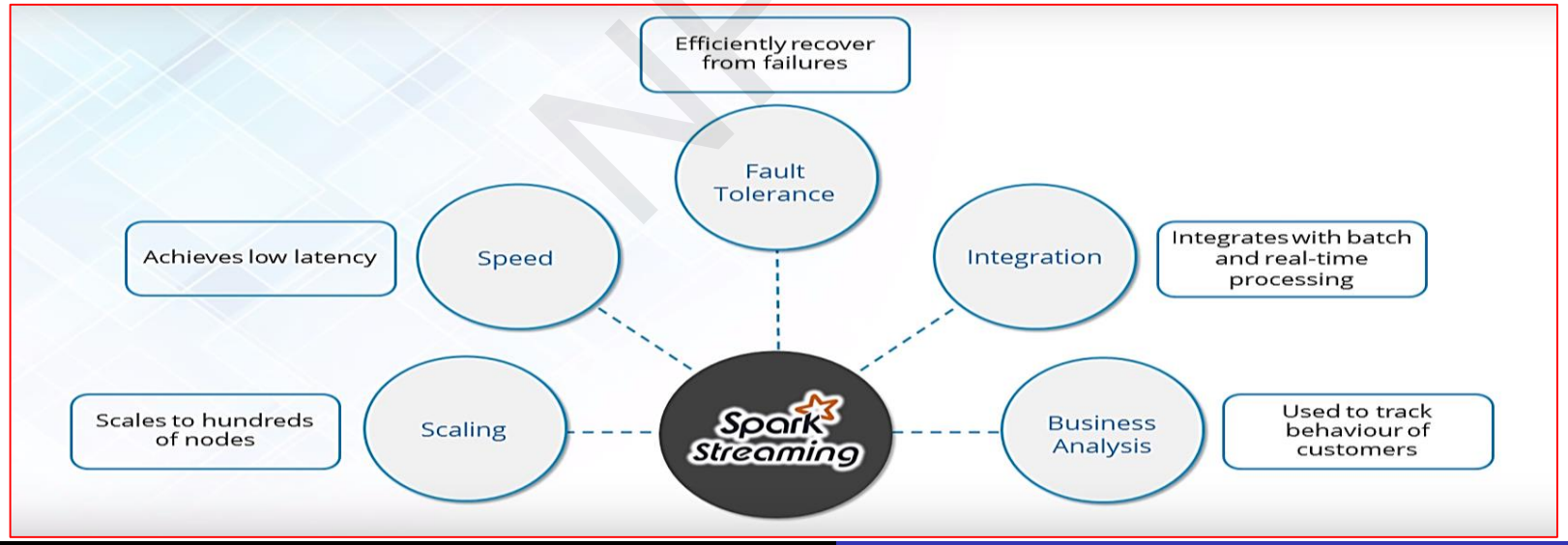
Achieves second-scale latencies

Efficiently recover from failures

Integrates with batch and interactive processing

Spark Streaming Features

- **Scaling:** Spark Streaming can easily scale to hundreds of nodes.
- **Speed:** It achieves low latency.
- **Fault Tolerance:** Spark has the ability to efficiently recover from failures.
- **Integration:** Spark integrates with batch and real-time processing.
- **Business Analysis:** Spark Streaming is used to track the behavior of customers which can be used in business analysis



Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing
- **Efficient fault-tolerance** in stateful computations

Batch vs Stream Processing

Batch Processing

- Ability to process and analyze data at-rest (stored data)
- Request-based, bulk evaluation and short-lived processing
- Enabler for **Retrospective, Reactive and On-demand Analytics**

Stream Processing

- Ability to ingest, process and analyze data in-motion in real- or near-real-time
- Event or micro-batch driven, continuous evaluation and long-lived processing
- Enabler for **real-time Prospective, Proactive and Predictive Analytics** for Next Best Action

Stream Processing + Batch Processing = All Data Analytics

real-time (now) **historical (past)**

Integration with Batch Processing

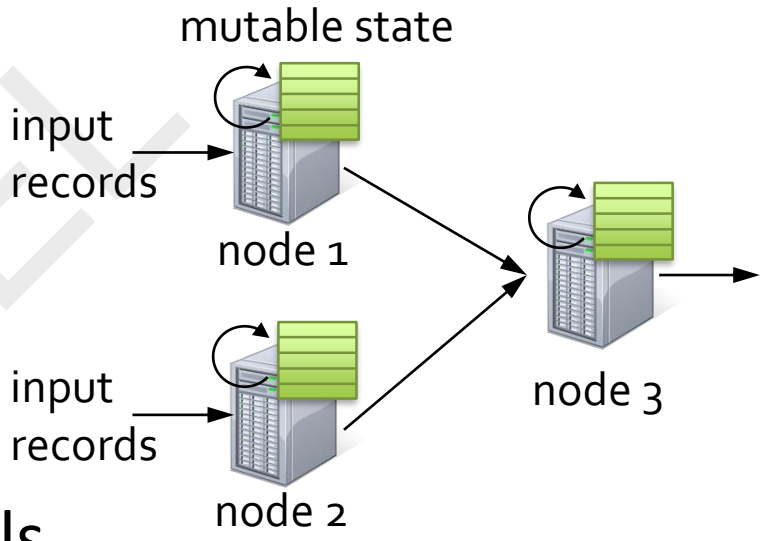
- Many environments require processing same data in live streaming as well as batch post-processing

- Existing frameworks cannot do both
  - Either, stream processing of 100s of MB/s with low latency
  - Or, batch processing of TBs of data with high latency

- Extremely painful to maintain two different
  - Different programming models
  - Double implementation effort

Stateful Stream Processing

- Traditional model
  - Processing pipeline of nodes
  - Each node maintains mutable state
  - Each input record updates the state and new records are sent out



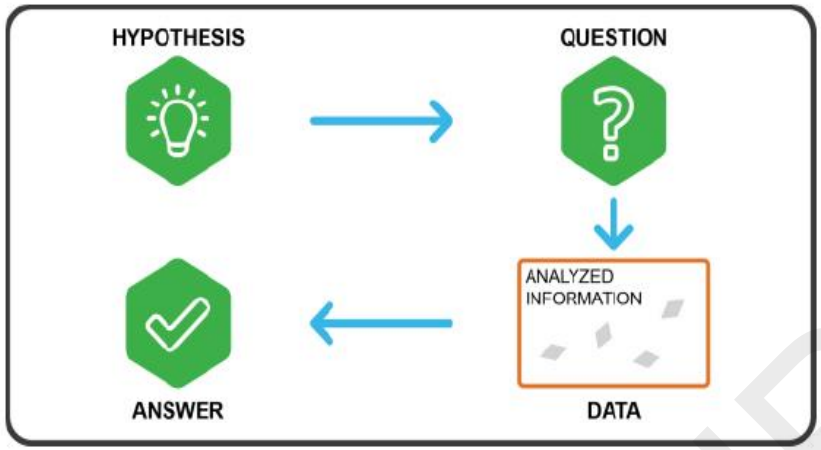
- Mutable state is lost if node fails

- Making stateful stream processing fault tolerant is challenging!

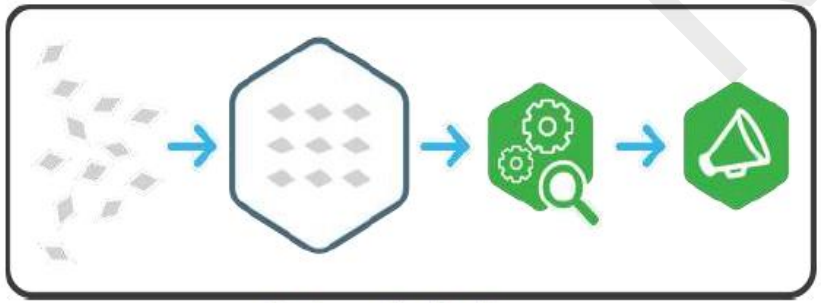
Modern Data Applications approach to Insights

Traditional Analytics

Structured & Repeatable
Structure built to store data



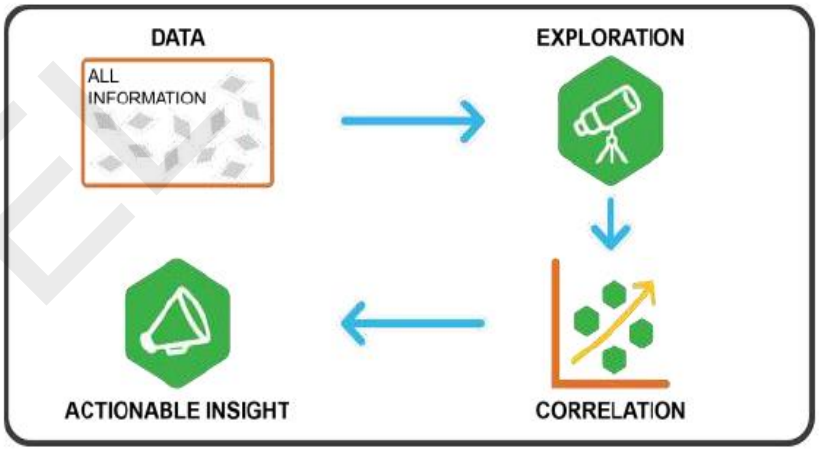
Start with hypothesis
Test against selected data



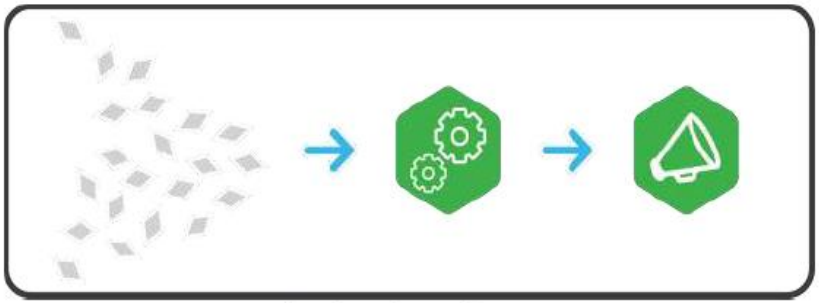
Analyze after landing...

Next Generation Analytics

Iterative & Exploratory
Data is the structure



Data leads the way
Explore *all* data, identify correlations



Analyze in motion...

Existing Streaming Systems

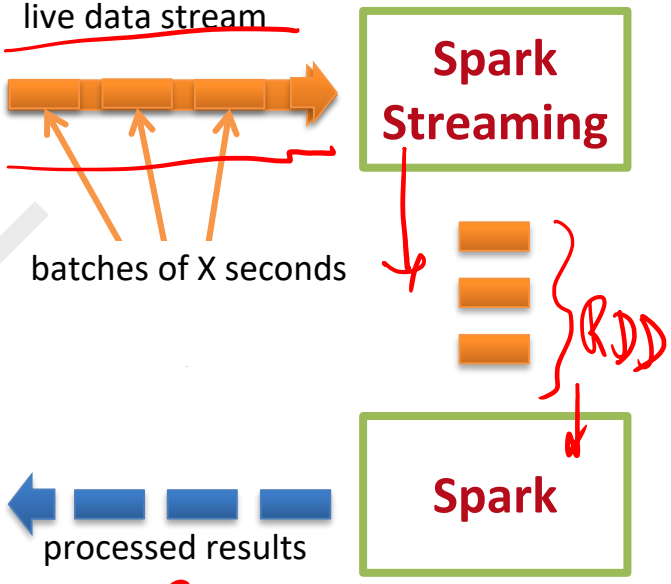
- Storm
  - Replays record if not processed by a node
  - Processes each record *at least once*
  - May update mutable state twice!
  - Mutable state can be lost due to failure!

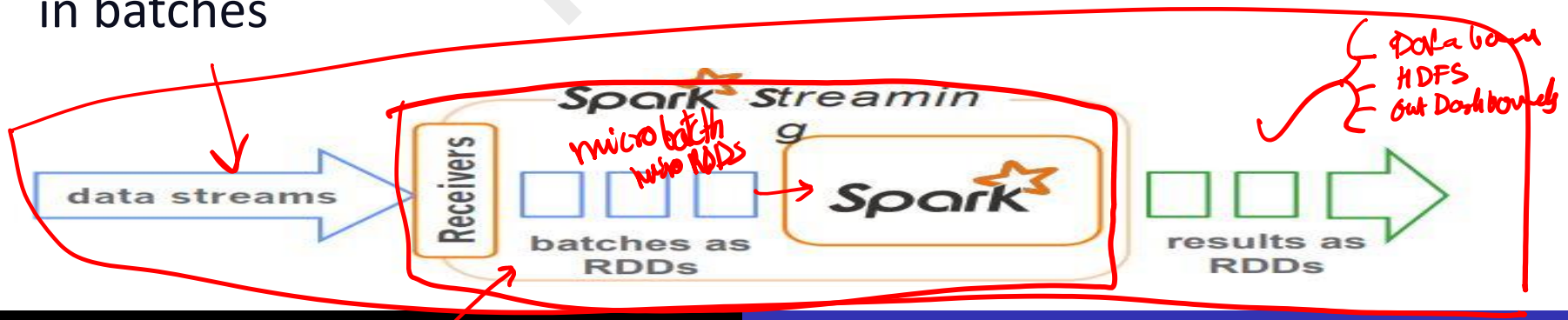
- Trident – Use transactions to update state
  - Processes each record *exactly once*
  - Per-state transaction to external database is slow

How does Spark Streaming work?

Run a streaming computation as a **series of very small, deterministic batch jobs**

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches

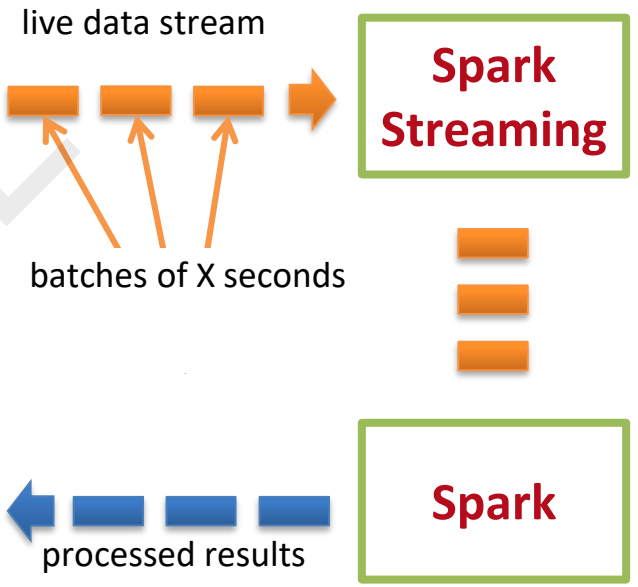




How does Spark Streaming work?

Run a streaming computation as a **series of very small, deterministic batch jobs**

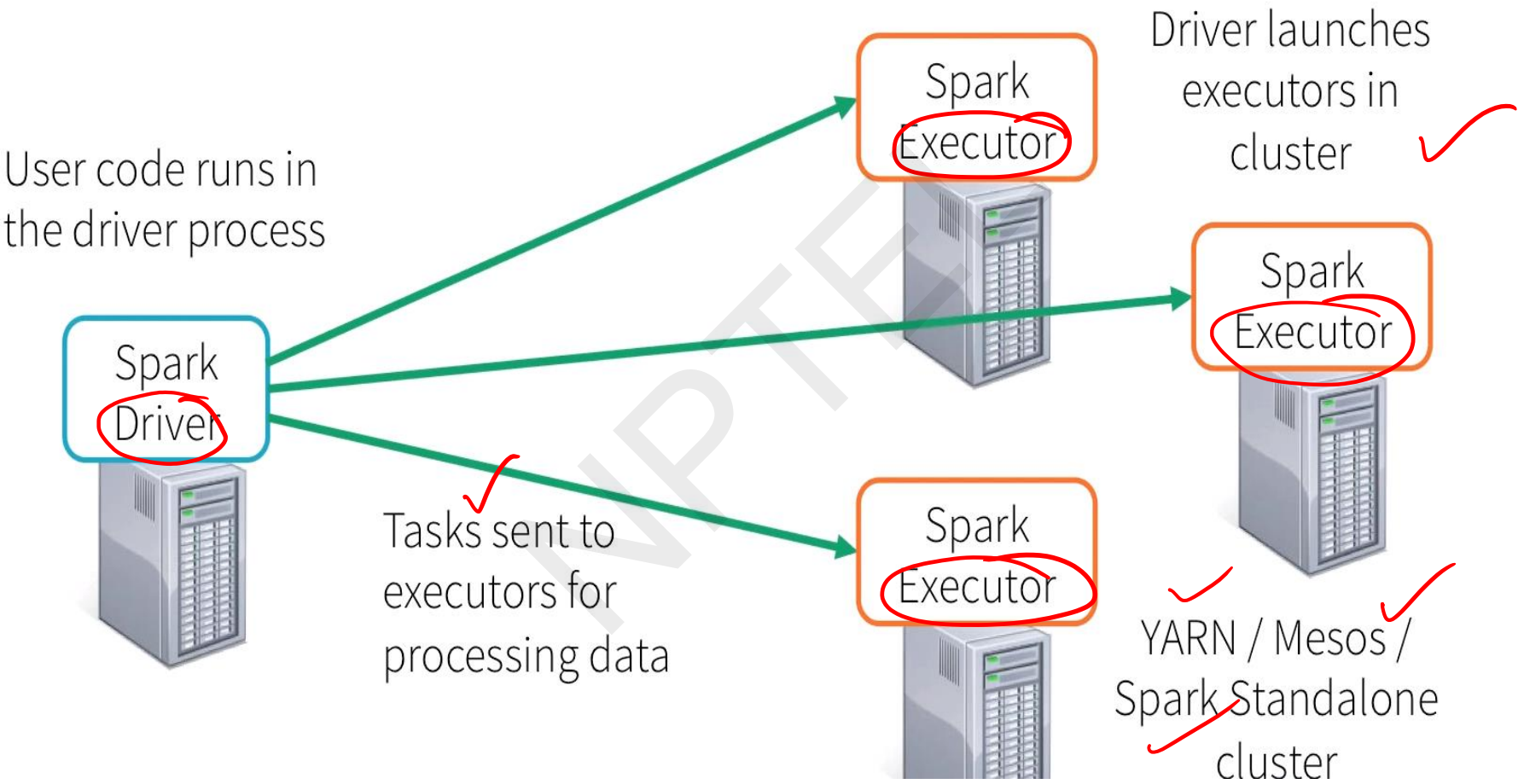
- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system



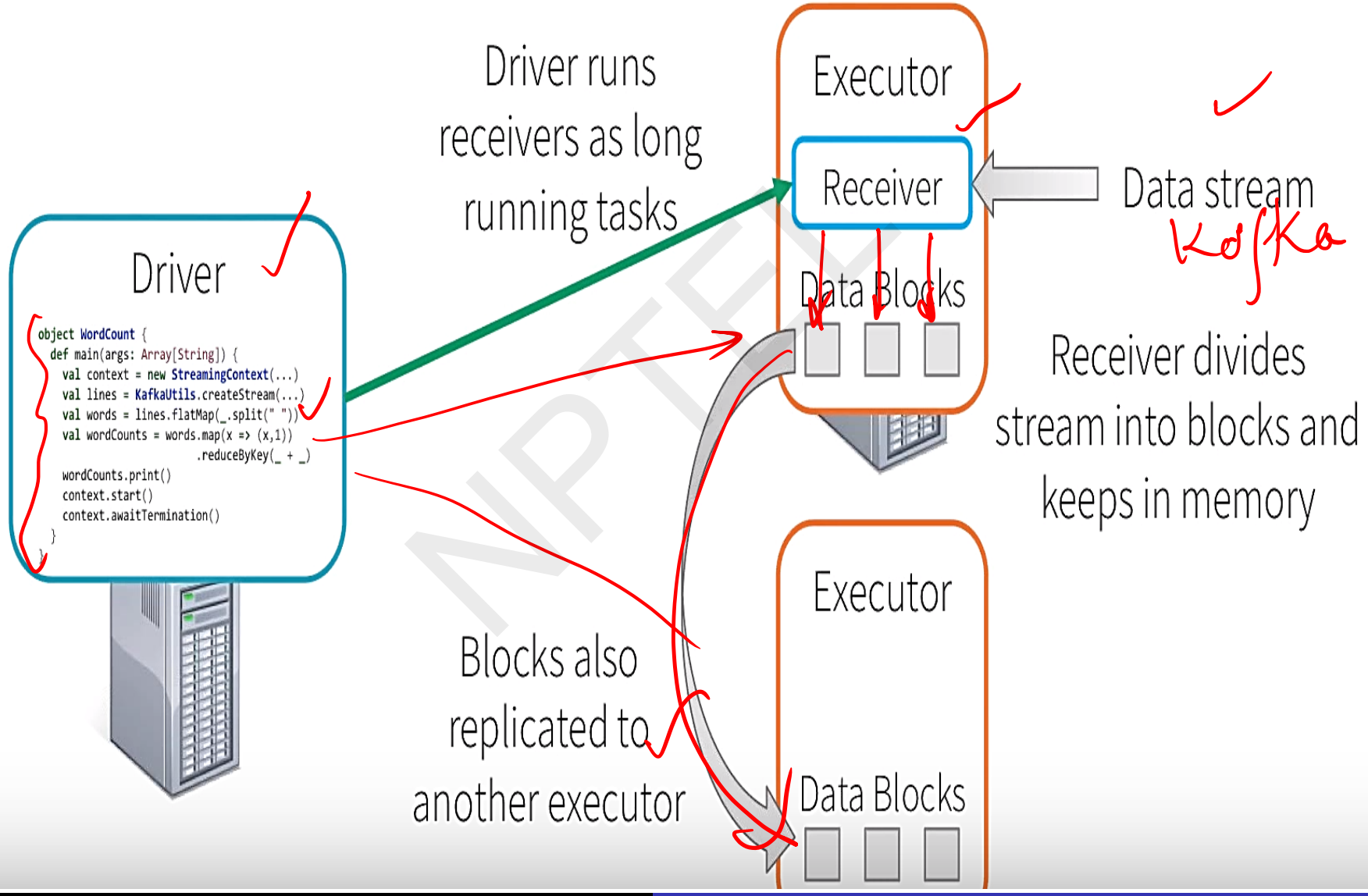
Word Count with Kafka

```
object WordCount {
  def main(args: Array[String]) {
    val context = new StreamingContext(new SparkConf(), Seconds(1))
    val lines = KafkaUtils.createStream(context, ...)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x,1)).reduceByKey(_ + _)
    wordCounts.print()
    context.start()
    context.awaitTermination()
  }
}
```

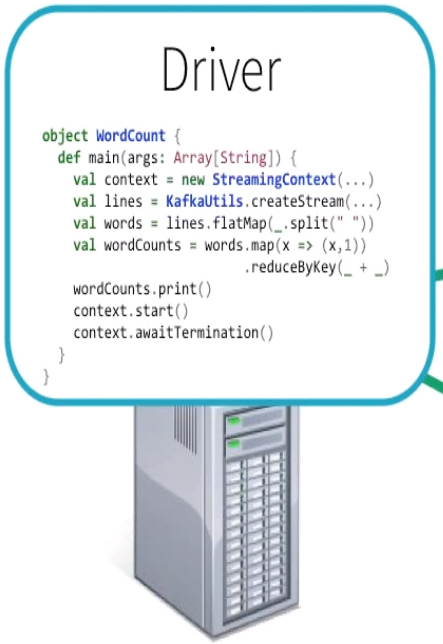
Any Spark Application



Spark Streaming Application: Receive data



Spark Streaming Application: Process data



Every batch interval, driver launches tasks to process the blocks

Executor

Receiver

Data Blocks

Executor

Data Blocks

results

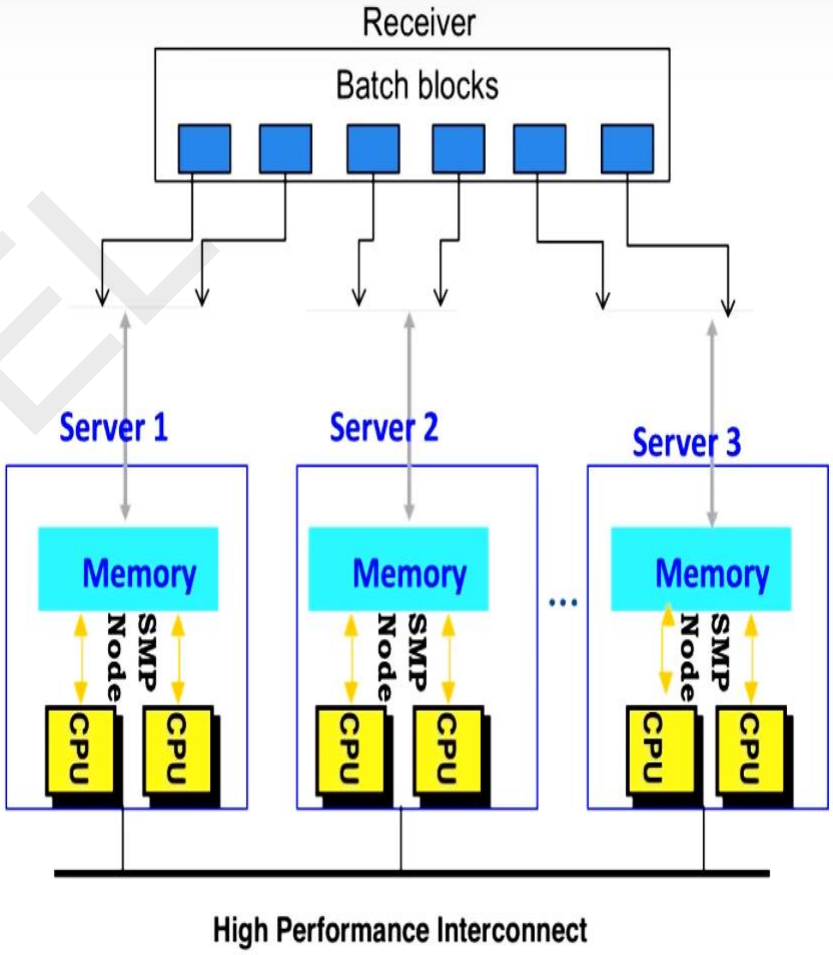
results

Data store

Process data in SP Streaming

Spark Streaming Architecture

- Micro batch architecture.
- Operates on interval of time
- New batches are created at regular time intervals.
- Divides received time batch into blocks for parallelism
- Each batch is a graph that translates into multiple jobs
- Has the ability to create larger size batch window as it processes over time.



Spark Streaming Workflow

- Spark Streaming workflow has four high-level stages. The first is to stream data from various sources. These sources can be streaming data sources like Akka, Kafka, Flume, AWS or Parquet for real-time streaming. The second type of sources includes HBase, MySQL, PostgreSQL, Elastic Search, Mongo DB and Cassandra for static/batch streaming.
- Once this happens, Spark can be used to perform Machine Learning on the data through its MLlib API. Further, Spark SQL is used to perform further operations on this data. Finally, the streaming output can be stored into various data storage systems like HBase, Cassandra, MemSQL, Kafka, Elastic Search, HDFS and local file system.

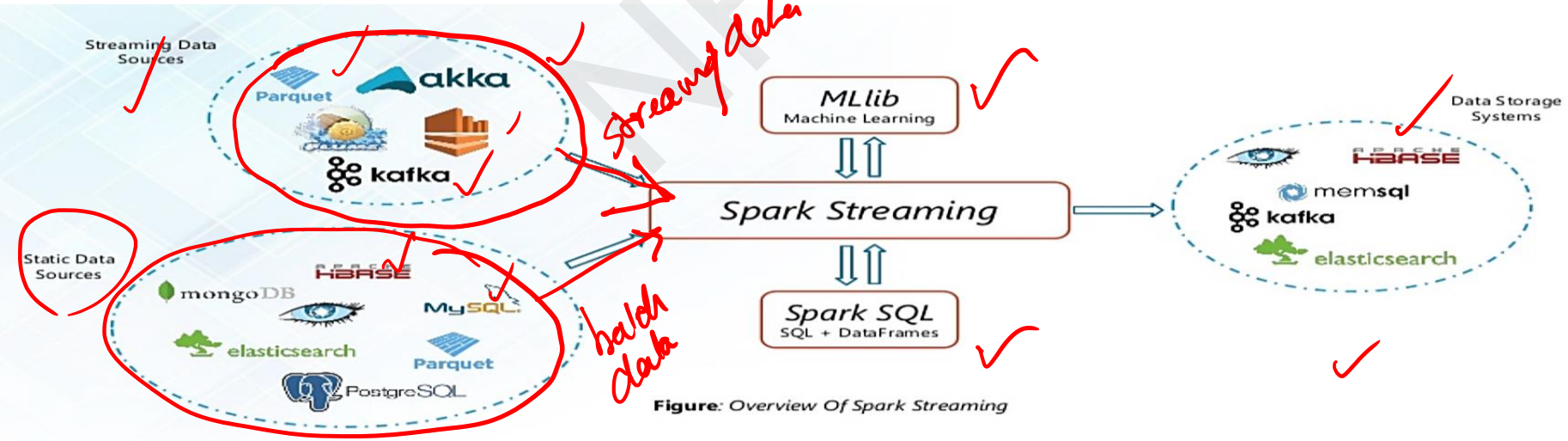


Figure: *Overview Of Spark Streaming*

Spark Streaming Workflow

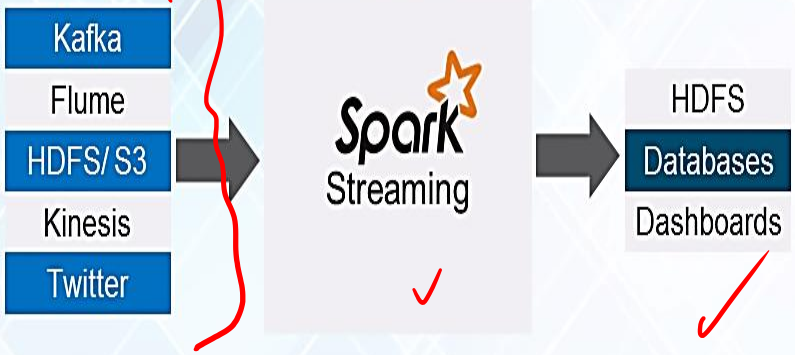


Figure: *Data from a variety of sources to various storage systems*

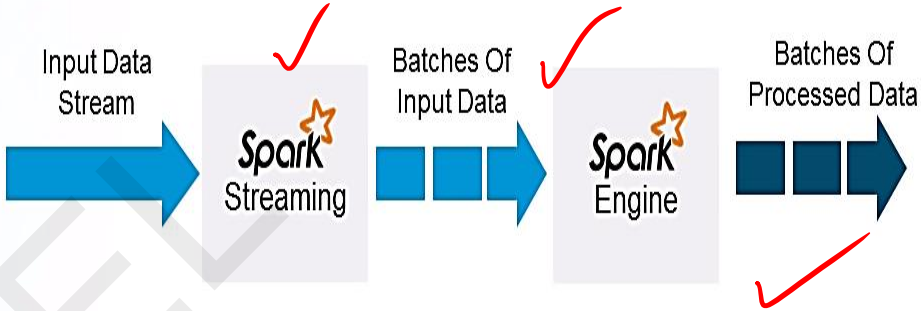


Figure: *Incoming streams of data divided into batches*

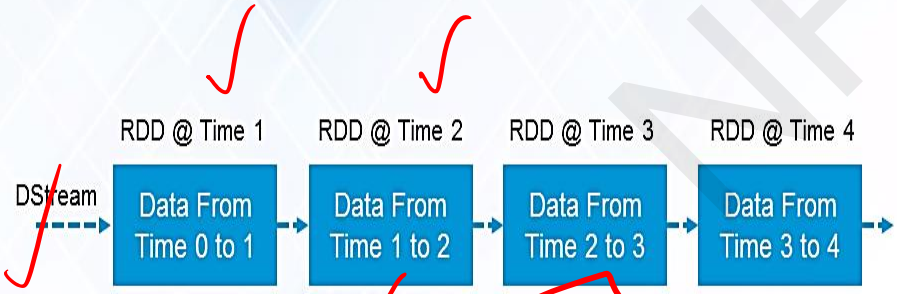


Figure: *Input data stream divided into discrete chunks of data*

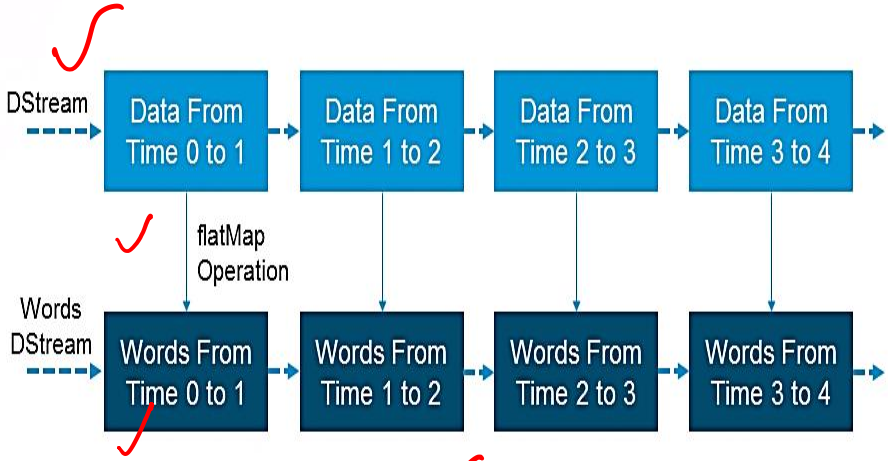
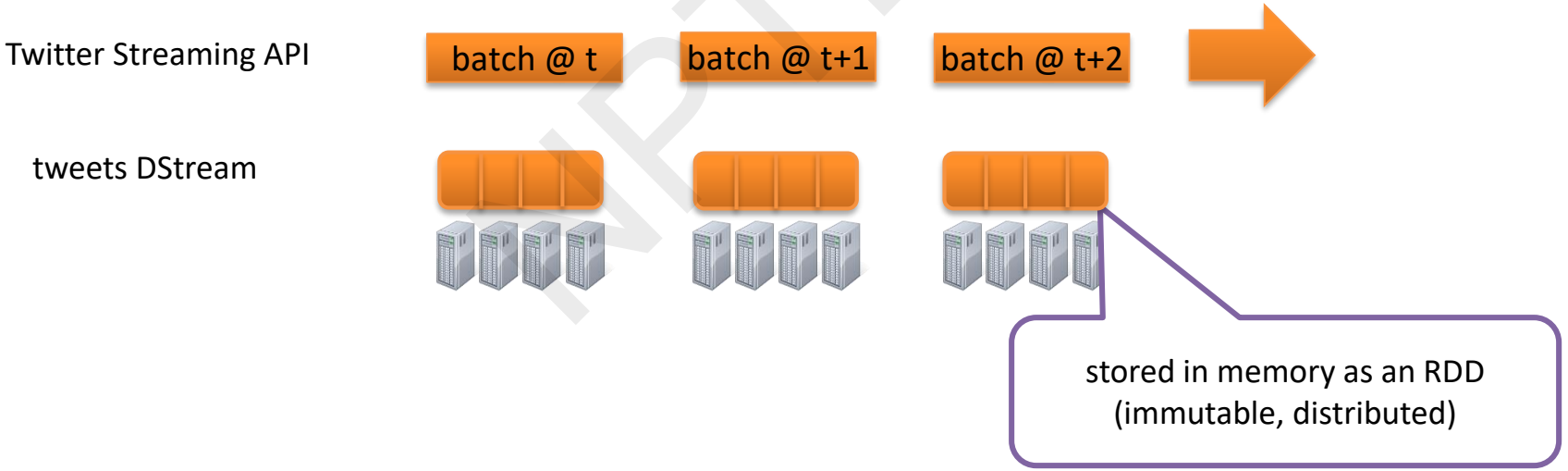


Figure: *Extracting words from an InputStream*

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
```

DStream: a sequence of RDDs representing a stream of data

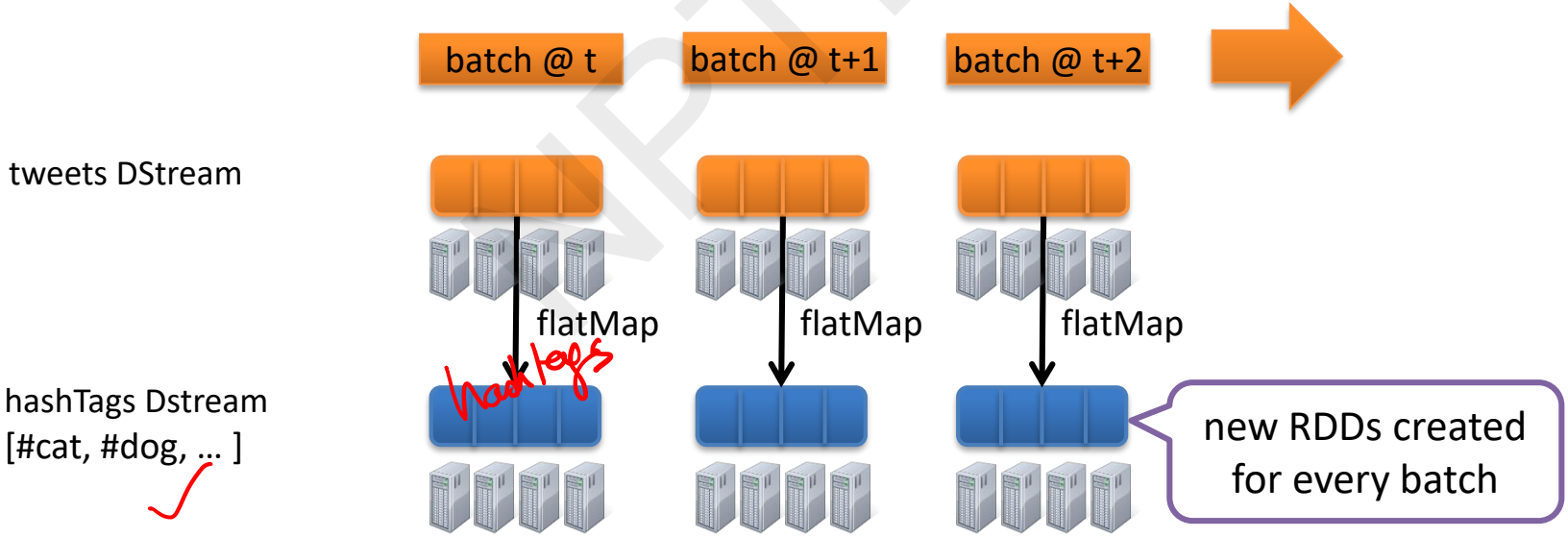


Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
```

new DStream

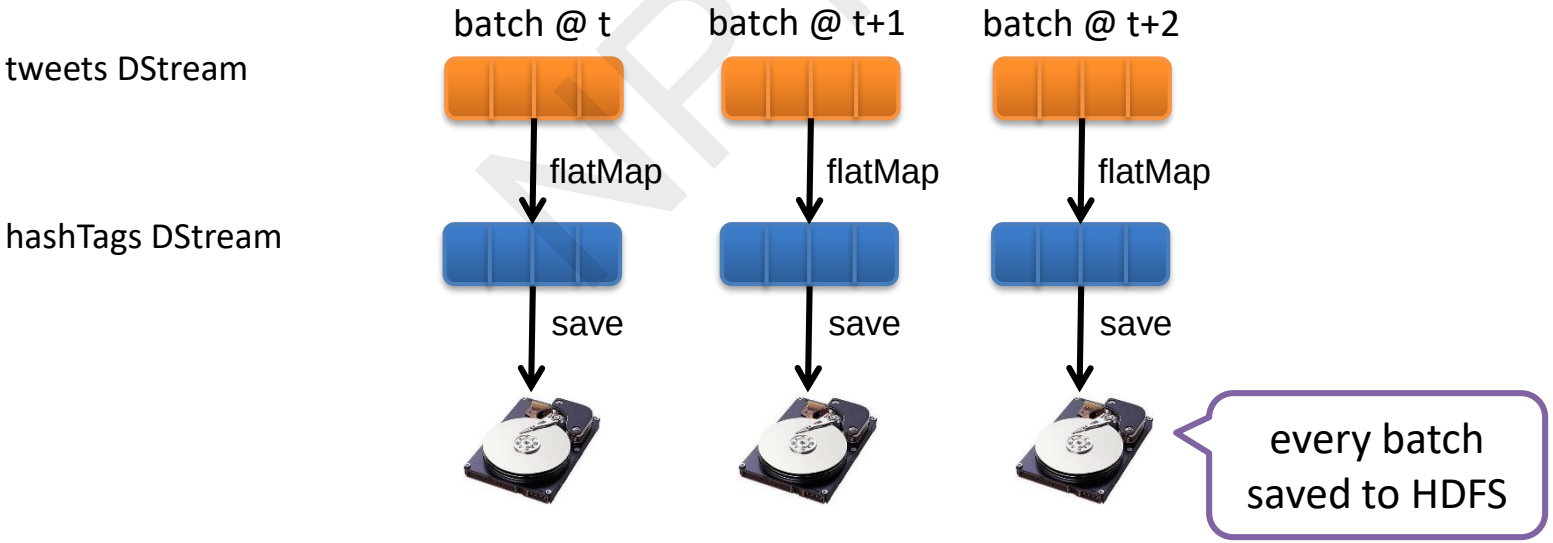
transformation: modify data in one DStream to create another DStream



Example 1– Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

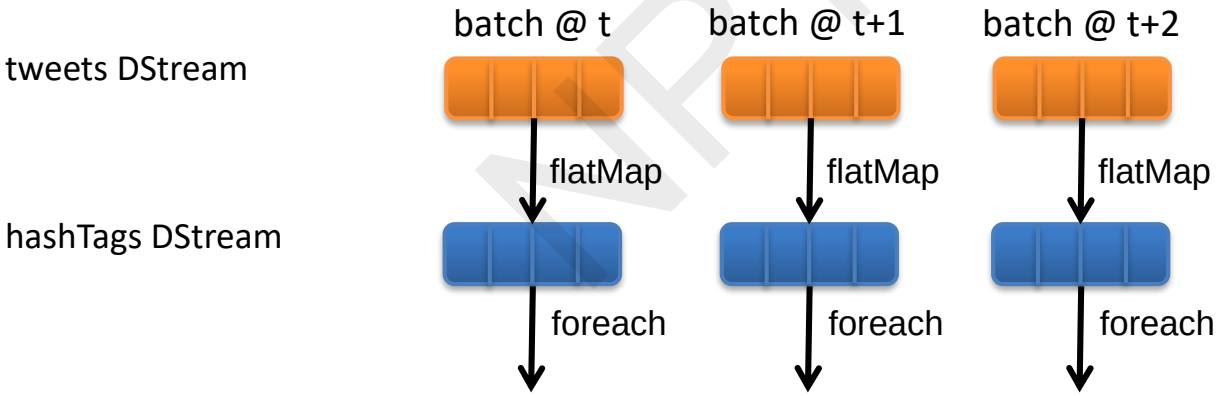
output operation: to push data to external storage



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Write to a database, update analytics UI, do whatever you want

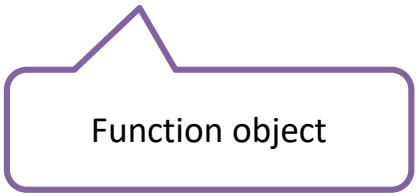
Java Example

Scala

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

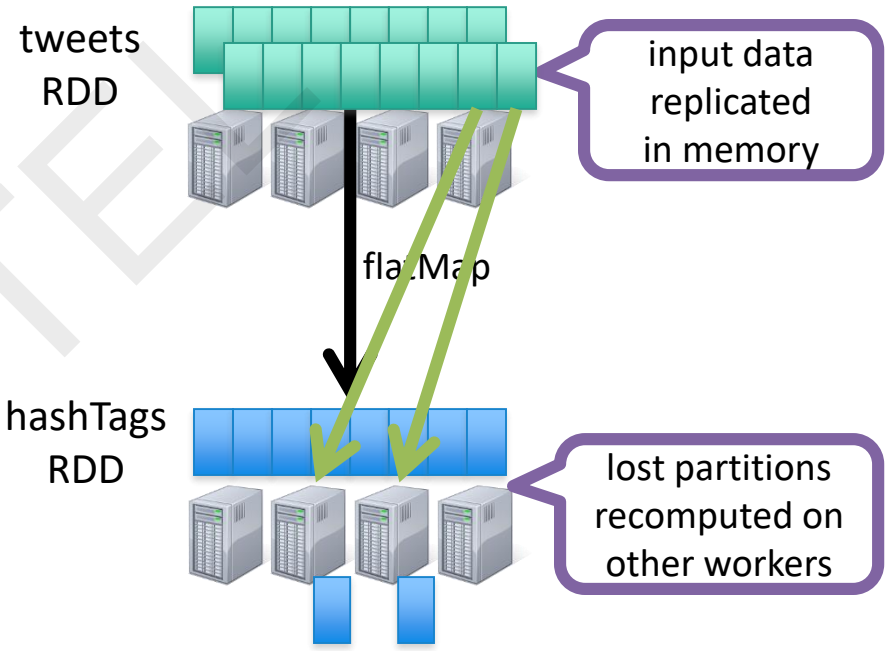
Java

```
JavaDStream<Status> tweets = ssc.twitterStream()
JavaDstream<String> hashTags = tweets.flatMap(new Function<...> {  })
hashTags.saveAsHadoopFiles("hdfs://...")
```



Fault-tolerance

- RDDs are remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data



Key concepts

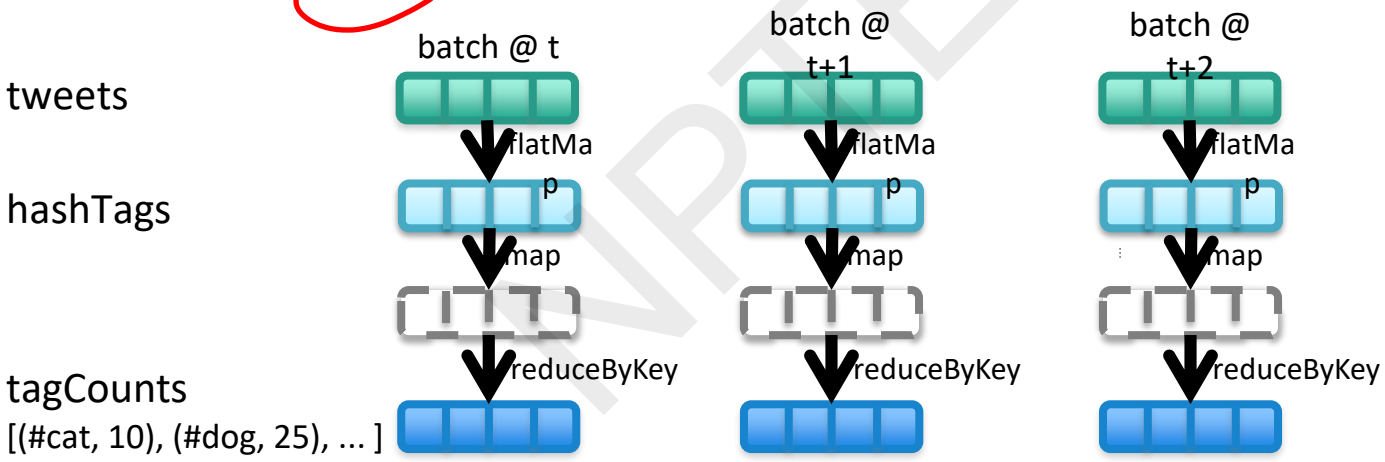
- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets

- **Transformations** – modify data from on DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, …
  - Stateful operations – window, countByValueAndWindow, …

- **Output Operations – send data to external entity**
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

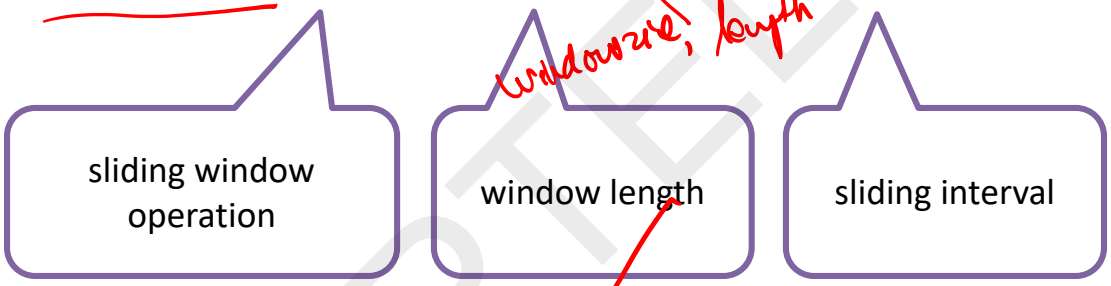
Example 2 – Count the hashtags

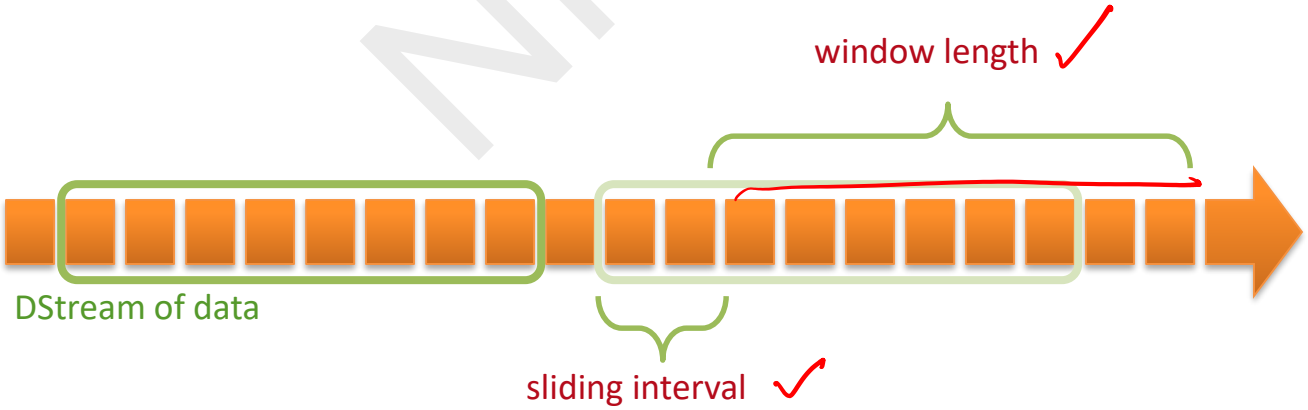
```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```



Example 3 – Count the hashtags over last 10 mins

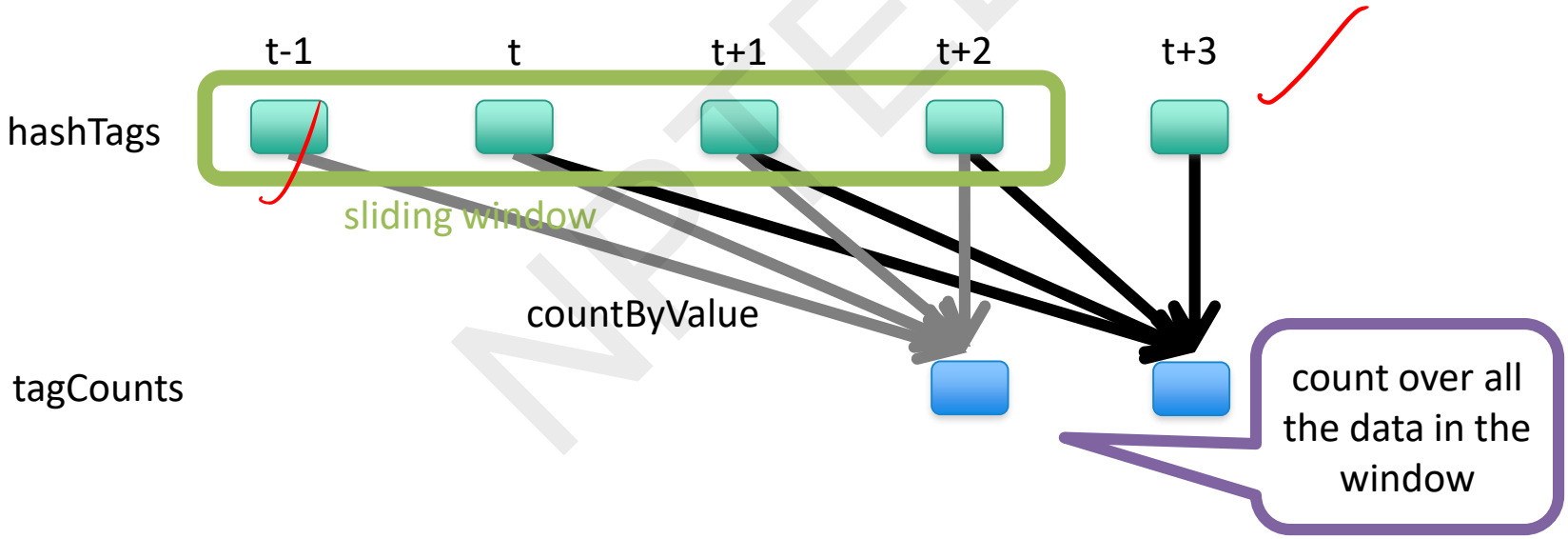
```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
val tagCounts = hashTags.window(Minutes(1),
Seconds(5)).countByValue()
```





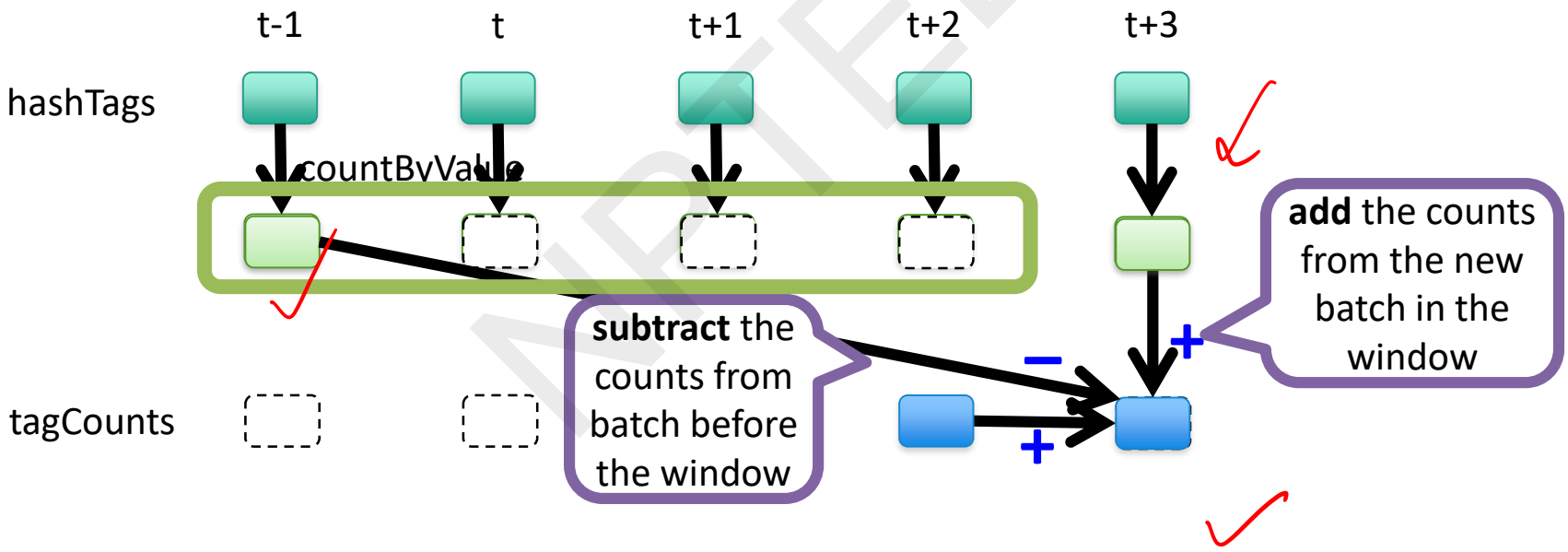
Example 3 – Counting the hashtags over last 10 mins

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



Smart window-based *reduce*

- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to "inverse reduce" ("subtract" for counting)

- Could have implemented counting as:

  hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), …)

Arbitrary Stateful Computations

Specify function to generate new state based on previous state and new data

- Example: Maintain per-user mood as state, and update it with their tweets

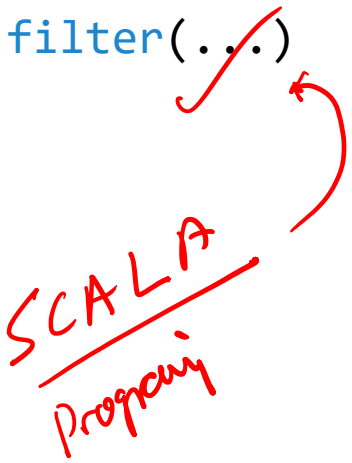
```
def updateMood(newTweets, lastMood) => newMood

moods = tweetsByUser.updateStateByKey(updateMood _)
```

Arbitrary Combinations of Batch and Streaming Computations

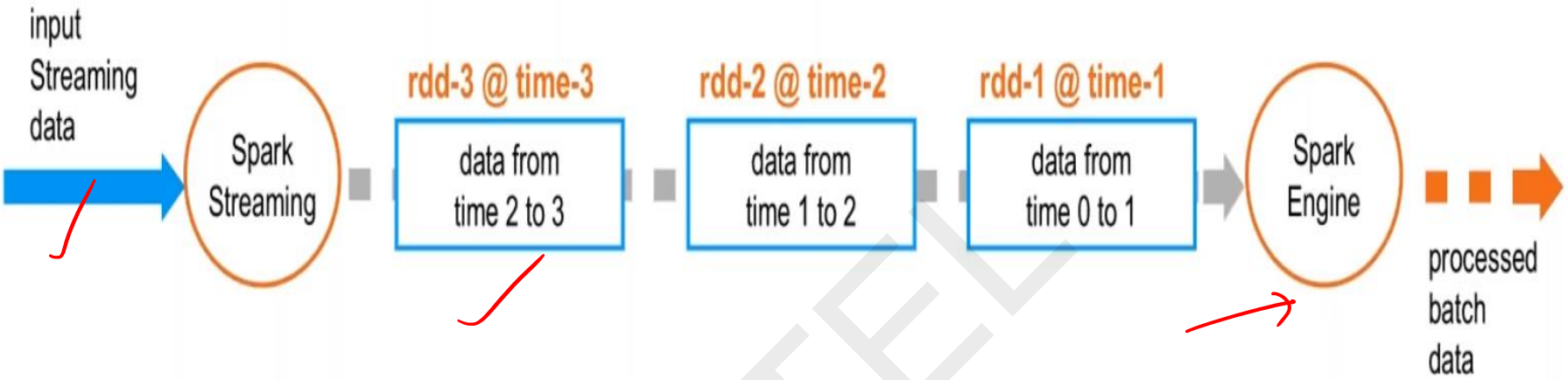
Inter-mix RDD and DStream operations!

- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

```
tweets.transform(tweetsRDD => {
    tweetsRDD.join(spamHDFSFile).filter(...)
})
```



Spark Streaming-Dstreams, Batches and RDDs



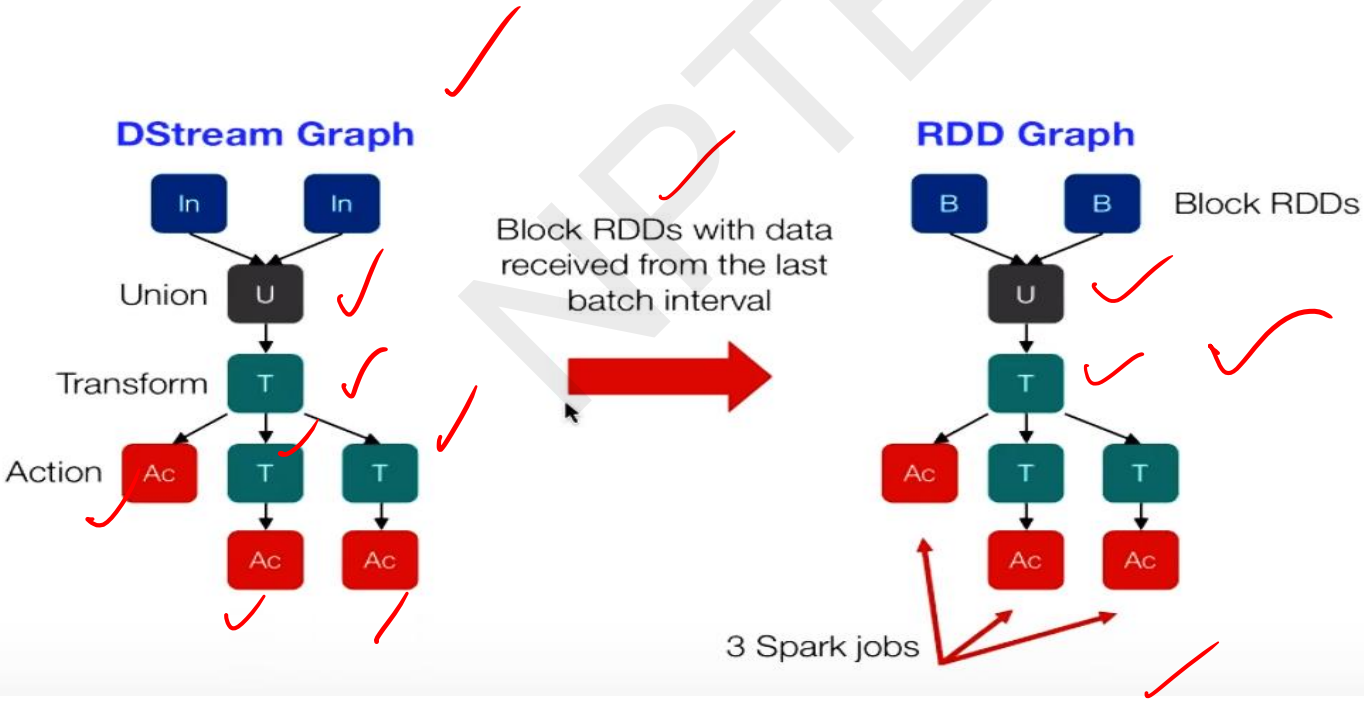
- These steps repeat for each batch.. Continuously
- Because we are dealing with Streaming data. Spark Streaming has the ability to “remember” the previous RDDs…to some extent.

DStreams + RDDs = Power

- Online machine learning
  - Continuously learn and update data models (*updateStateByKey* and *transform*)
- Combine live data streams with historical data
  - Generate historical data models with Spark, etc.
  - Use data models to process live data stream (*transform*)
- CEP-style processing
  - window-based operations (reduceByWindow, etc.)

From DStreams to Spark Jobs

- Every interval, an RDD graph is computed from the DStream graph
- For each output operation, a Spark action is created
- For each action, a Spark job is created to compute it



Input Sources

- Out of the box, we provide
  - Kafka, HDFS, Flume, Akka Actors, Raw TCP sockets, etc.
- Very easy to write a *receiver* for your own data source
- Also, generate your own RDDs from Spark, etc. and push them in as a "stream"

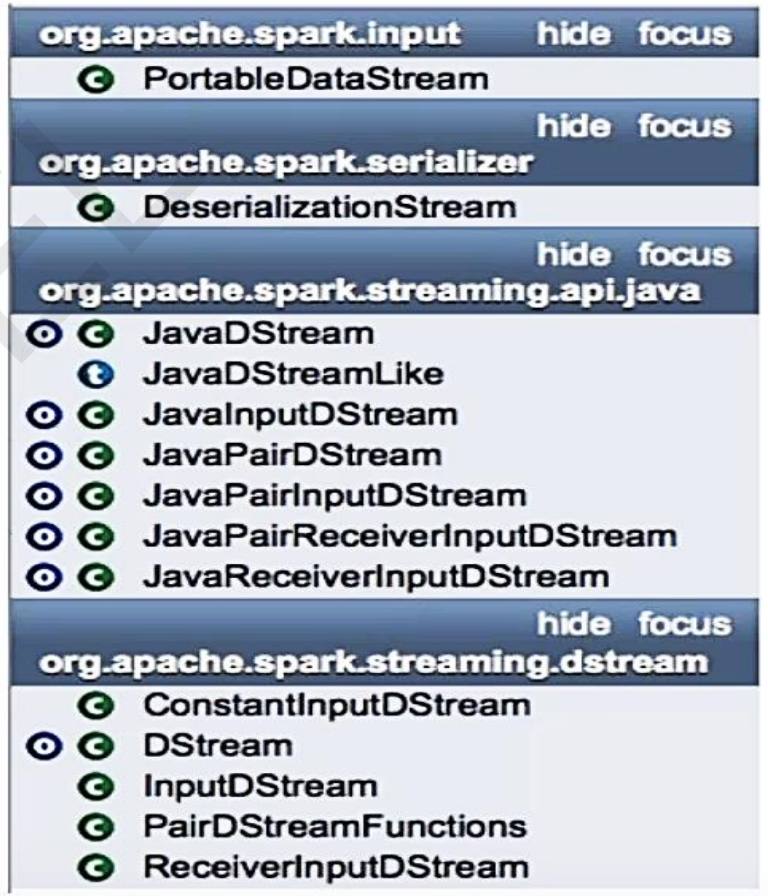
Current Spark Streaming I/O

- Input Sources
  - Kafka, Flume, Twitter, ZeroMQ, MQTT, TCP sockets
  - Basic sources: sockets, files, Akka actors
  - Other sources require receiver threads

- Output operations
  - Print(), saveAsTextFiles(), saveAsObjectFiles(), saveAsHadoopFiles(), foreachRDD()
  - foreachRDD can be used for message queues, DB operations and more

Dstream Classes

- Different classes for different languages (Scala, Java)
- Dstream has 36 value members
- Multiple types of Dstreams
- Separate Python API



Spark Streaming Operations

- All the Spark RDD operations
  - Some available through the transform() operation

map/flatmap	filter	repartition	union
count	reduce	countByValue	reduceByKey
join	cogroup	transform	updateStateByKey

- Spark Streaming window operations

window	countByWindow	reduceByWindow
reduceByKeyAndWindow	countByValueAndWindow	

- Spark Streaming output operations

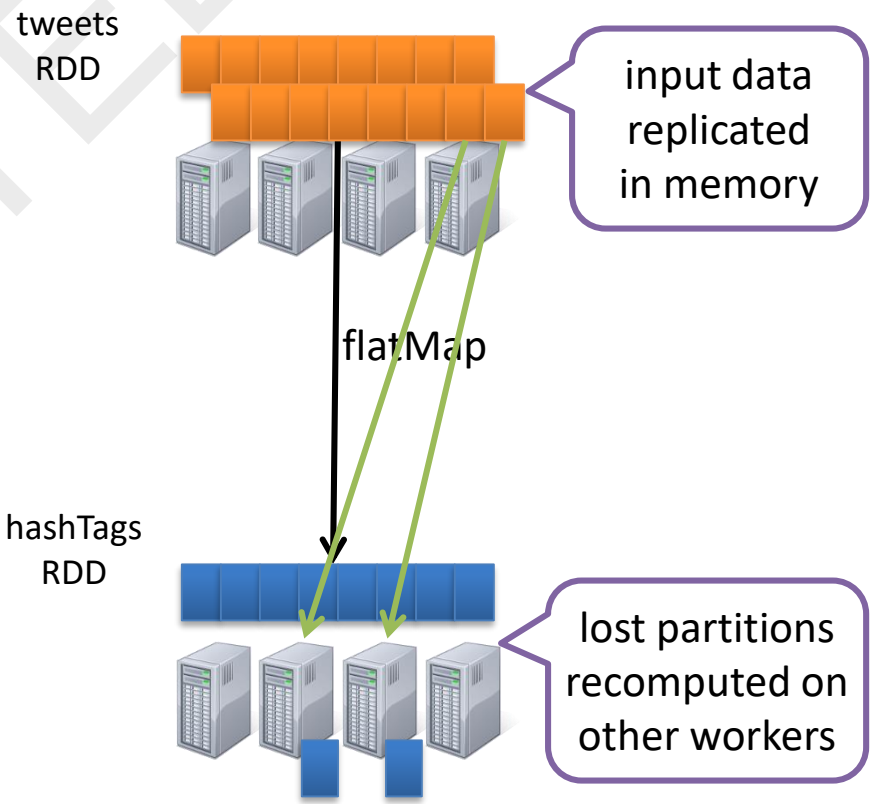
print	saveAsTextFiles	saveAsObjectFiles
saveAsHadoopFiles	foreachRDD	

Fault-tolerance

- Batches of input data are replicated in memory for fault-tolerance

- Data lost due to worker failure, can be recomputed from replicated input data

  - All transformations are fault-tolerant, and *exactly-once* transformations

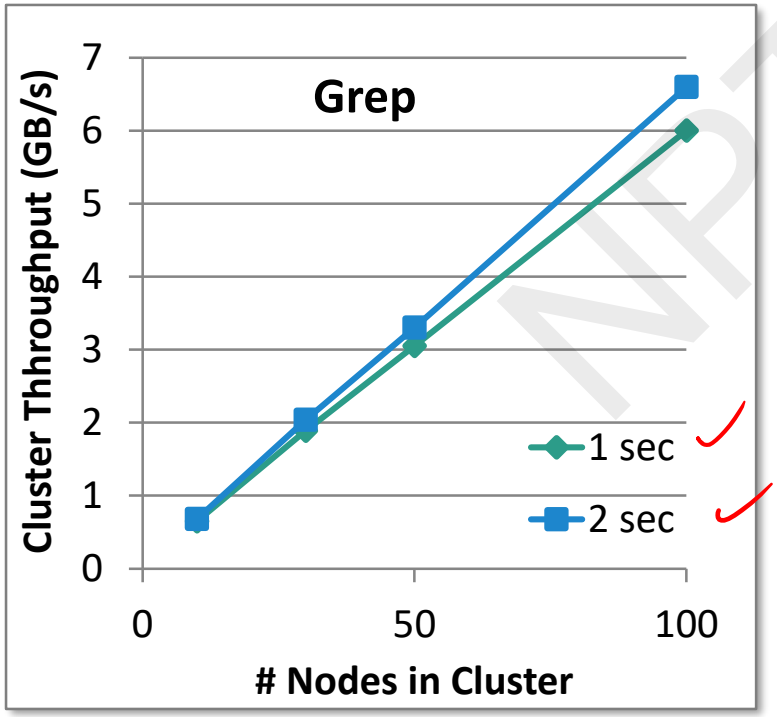


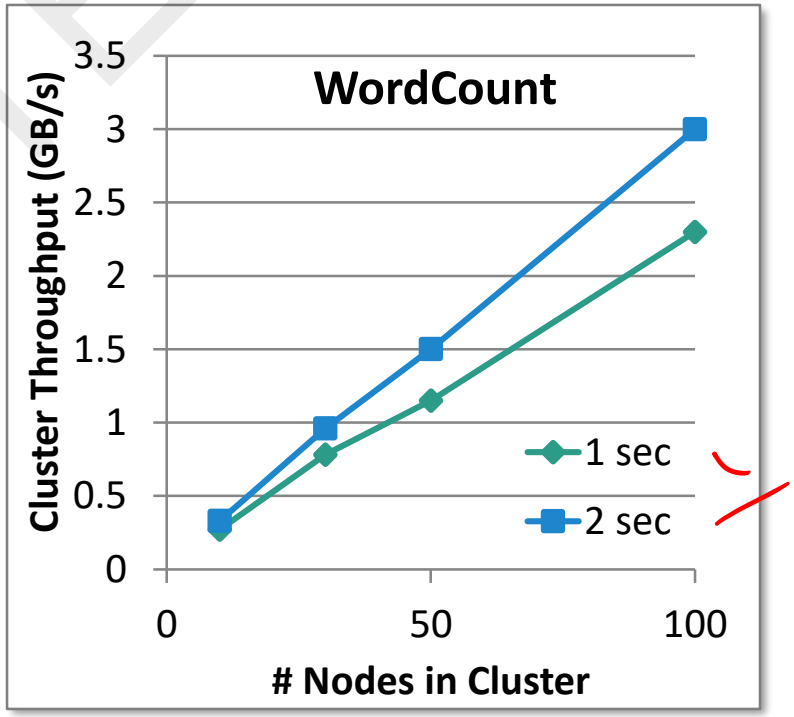
Fault-tolerance

- Received data is replicated among multiple Spark executors
  - Default factor: 2
- Checkpointing
  - Saves state on regular basis, typically every 5-10 batches of data
  - A failure would have to replay the 5-10 previous batched to recreate the appropriate RDDs
  - Checkpoint done to HDFS or equivalent
- Must protect the driver program
  - If the driver node running the Spark Streaming application fails
  - Driver must be restarted on another node.
  - Requires a checkpoint directory in the StreamingContext
- Streaming Backpressure
  - spark.streaming.backpressure.enabled
  - spark.streaming.receiver.maxRate

Performance

Can process **60M records/sec (6 GB/sec)** on **100 nodes** at **sub-second** latency

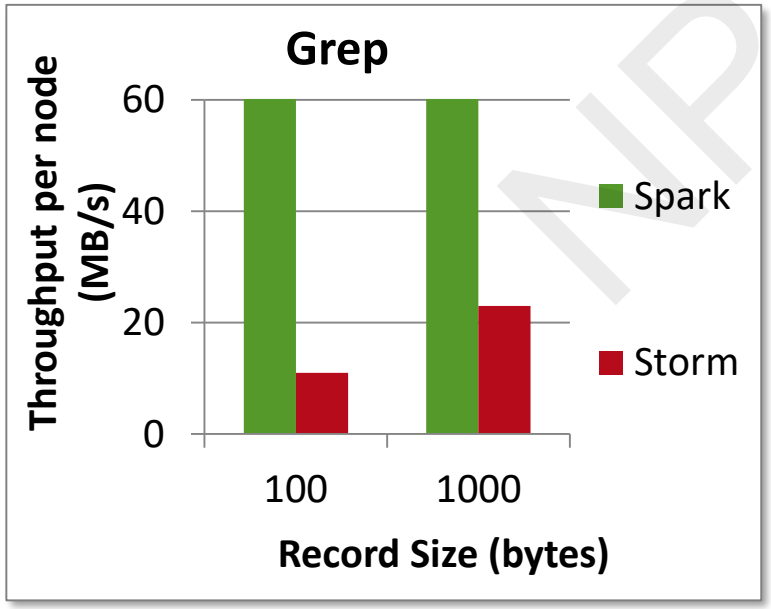


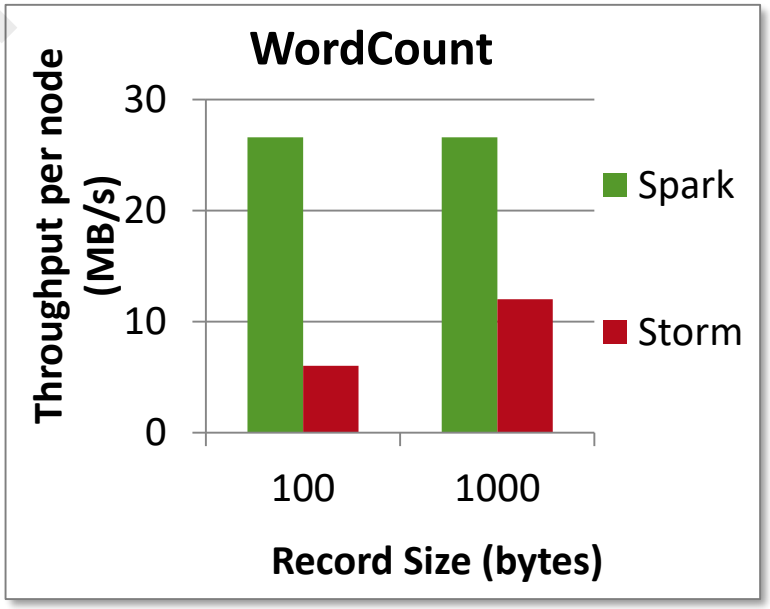


Comparison with other systems

Higher throughput than Storm

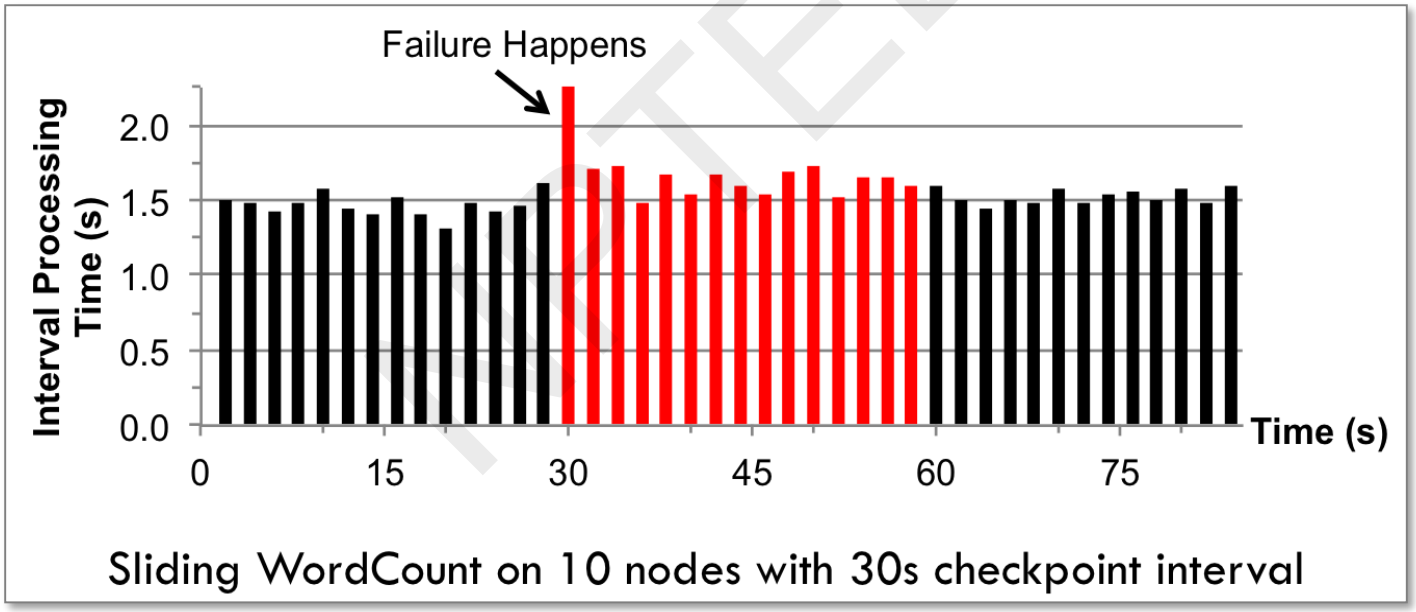
- Spark Streaming: **670k** records/sec/node
- Storm: **115k** records/sec/node
- Commercial systems: **100-500k** records/sec/node





Fast Fault Recovery

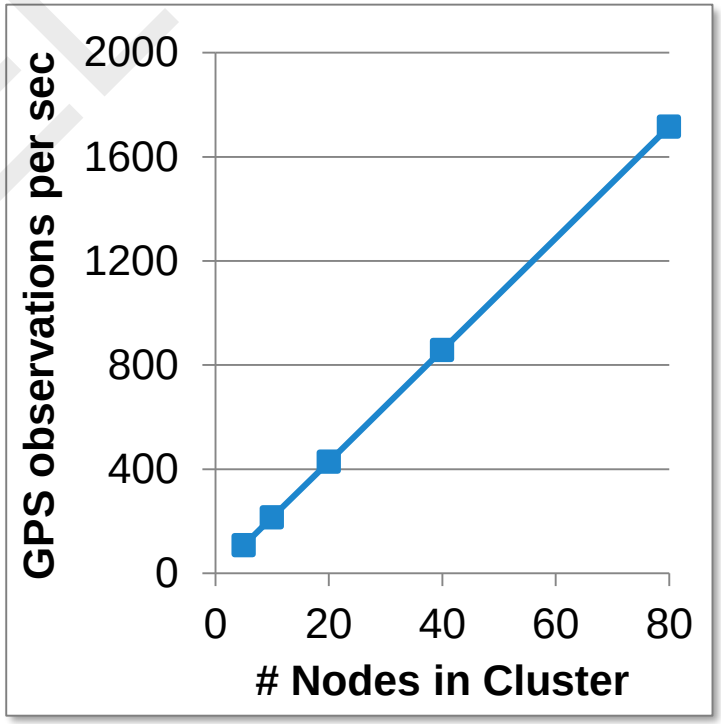
Recovers from faults/stragglers within **1 sec**



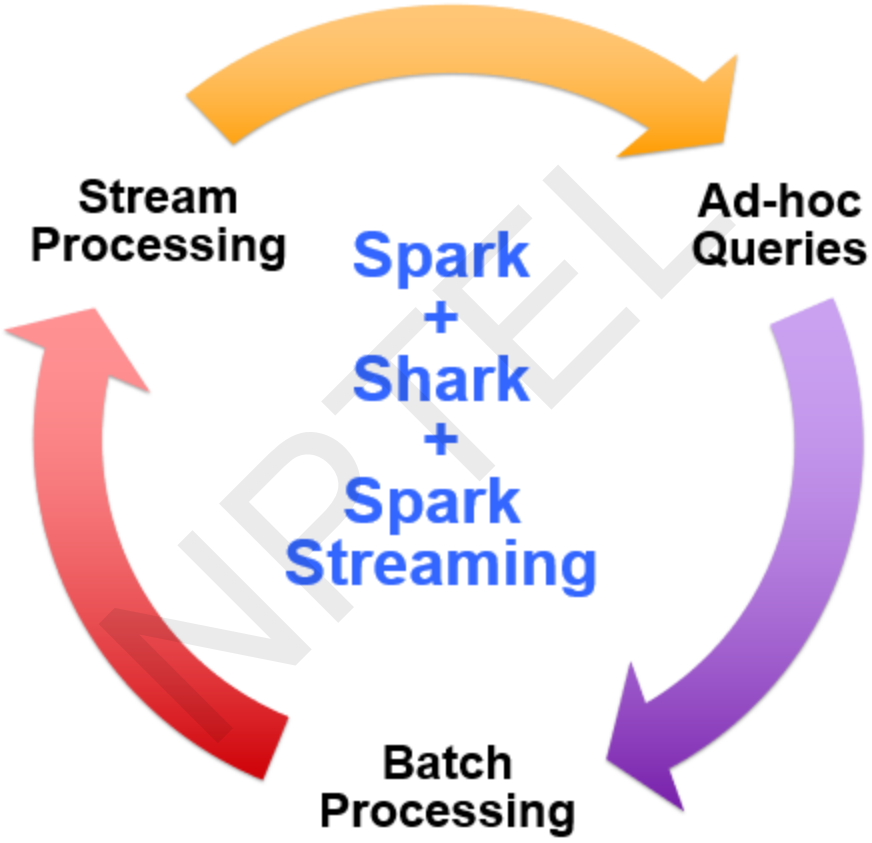
Real time application: Mobile Millennium Project

Traffic transit time estimation using online machine learning on GPS observations

- Markov-chain Monte Carlo simulations on GPS observations
- Very CPU intensive, requires dozens of machines for useful computation
- Scales linearly with cluster size



Vision - *one stack to rule them all*



Spark program vs Spark Streaming program

Spark Streaming program on Twitter stream

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Spark program on Twitter log file

```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

Advantage of an unified stack

- Explore data interactively to identify problems
- Use same code in Spark for processing large logs
- Use similar code in Spark Streaming for realtime processing

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(...)
```

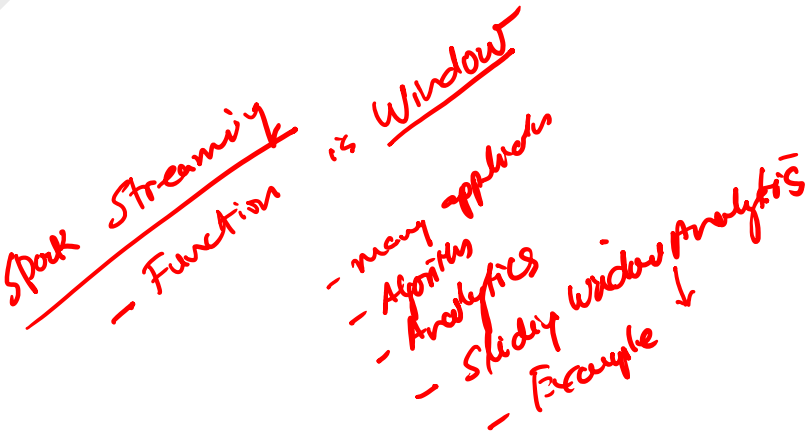
```
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = stream.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

Roadmap

- Spark 0.8.1
  - Marked alpha, but has been quite stable
  - Master fault tolerance – manual recovery
    - Restart computation from a checkpoint file saved to HDFS
- Spark 0.9 in Jan 2014 – out of alpha!
  - Automated master fault recovery
  - Performance optimizations
  - Web UI, and better monitoring capabilities
- Spark v2.4.0 released in November 2, 2018

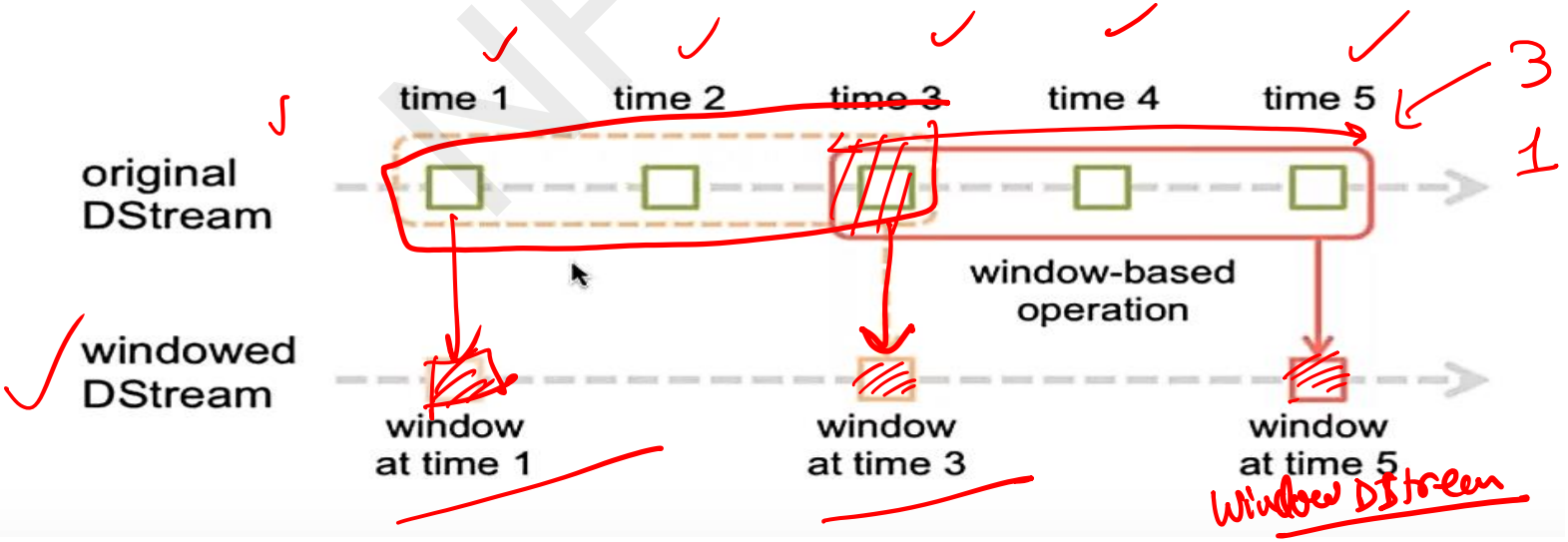
Sliding Window Analytics



Spark Streaming Windowing Capabilities

- **Parameters**
  - **Window length:** duration of the window
  - **Sliding interval:** interval at which the window operation is performed
  - Both the parameters must be a multiple of the batch interval

- A window creates a new DStream with a larger batch size



Spark Window Functions

Spark Window Functions for DataFrames and SQL

Introduced in Spark 1.4, Spark window functions improved the expressiveness of Spark DataFrames and Spark SQL. With window functions, you can easily calculate a moving average or cumulative sum, or reference a value in a previous row of a table. Window functions allow you to do many common calculations with DataFrames, without having to resort to RDD manipulation.

Aggregates, UDFs vs. Window functions

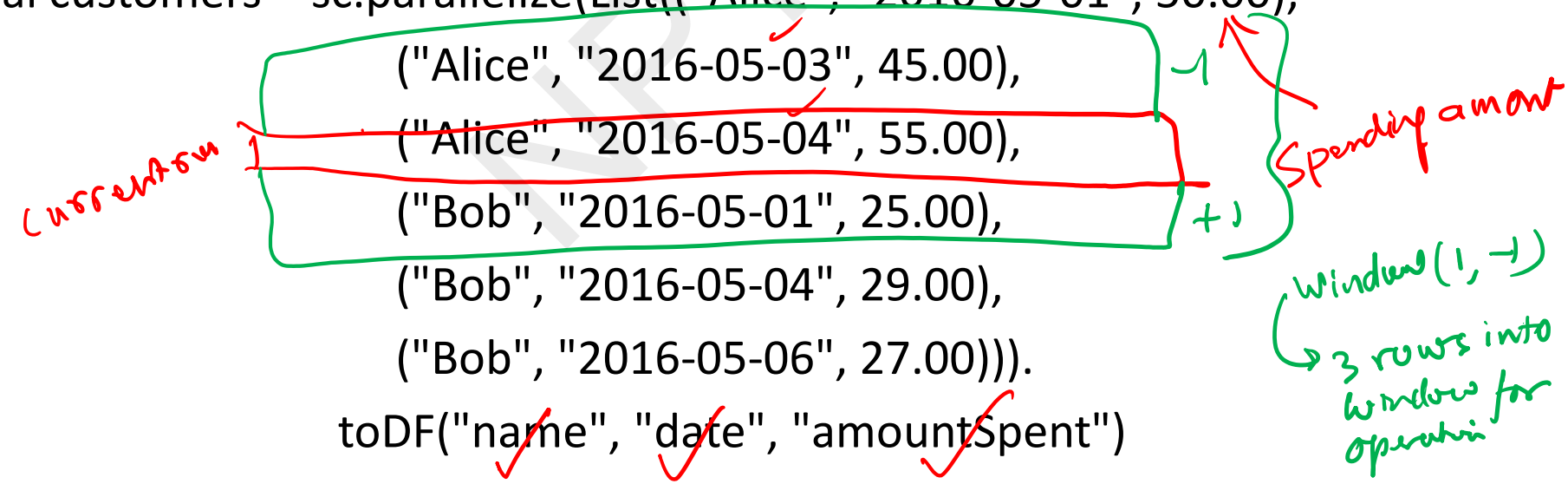
Window functions are complementary to existing DataFrame operations: aggregates, such as sum and avg, and UDFs. To review, aggregates calculate one result, a sum or average, for each group of rows, whereas UDFs calculate one result for each row based on only data in that row. In contrast, window functions calculate one result for each row based on a window of rows. For example, in a moving average, you calculate for each row the average of the rows surrounding the current row; this can be done with window functions.

Moving Average Example

- Let us dive right into the moving average example. In this example dataset, there are two customers who have spent different amounts of money each day.

- // Building the customer DataFrame. All examples are written in Scala with Spark 1.6.1, but the same can be done in Python or SQL.

```
val customers = sc.parallelize(List(("Alice", "2016-05-01", 50.00),
                ("Alice", "2016-05-03", 45.00),
                ("Alice", "2016-05-04", 55.00),
                ("Bob", "2016-05-01", 25.00),
                ("Bob", "2016-05-04", 29.00),
                ("Bob", "2016-05-06", 27.00))).
            toDF("name", "date", "amountSpent")
```



Moving Average Example

```
// Import the window functions.
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._

// Create a window spec.
val wSpec1 =
Window.partitionBy("name").orderBy("date").rowsBetween(-1, 1)
```

- In this window spec, the data is partitioned by customer. Each customer's data is ordered by date. And, the window frame is defined as starting from -1 (one row before the current row) and ending at 1 (one row after the current row), for a total of 3 rows in the sliding window.

Moving Average Example

```
// Calculate the moving average
customers.withColumn( "movingAvg",
                      avg(customers("amountSpent")).over(wSpec1) ).show()
```

This code adds a new column, "movingAvg", by applying the avg function on the sliding window defined in the window spec:

name	date	amountSpent	movingAvg
Alice	5/1/2016	50	47.5
Alice	5/3/2016	45	50
Alice	5/4/2016	55	50
Bob	5/1/2016	25	27
Bob	5/4/2016	29	27
Bob	5/6/2016	27	28

Window function and Window Spec definition

- As shown in the above example, there are two parts to applying a window function: (1) specifying the window function, such as avg in the example, and (2) specifying the window spec, or wSpec1 in the example. For (1), you can find a full list of the window functions here:
- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$
- You can use functions listed under “Aggregate Functions” and “Window Functions”.

- For (2) specifying a window spec, there are three components: partition by, order by, and frame.

  1. “Partition by” defines how the data is grouped; in the above example, it was by customer. You have to specify a reasonable grouping because all data within a group will be collected to the same machine. Ideally, the DataFrame has already been partitioned by the desired grouping.
  2. “Order by” defines how rows are ordered within a group; in the above example, it was by date.
  3. “Frame” defines the boundaries of the window with respect to the current row; in the above example, the window ranged between the previous row and the next row.

Cumulative Sum

Next, let us calculate the cumulative sum of the amount spent per customer.

// Window spec: the frame ranges from the beginning (Long.MinValue) to the current row (0).

```
val wSpec2 =
Window.partitionBy("name").orderBy("date").rowsBetween(Long.MinValue, 0)
```

// Create a new column which calculates the sum over the defined window frame.

```
customers.withColumn( "cumSum",
  sum(customers("amountSpent")).over(wSpec2)  ).show()
```

name	date	amountSpent	cumSum
Alice	5/1/2016	50	50
Alice	5/3/2016	45	95
Alice	5/4/2016	55	150
Bob	5/1/2016	25	25
Bob	5/4/2016	29	54
Bob	5/6/2016	27	81

Data from previous row

In the next example, we want to see the amount spent by the customer in their previous visit.

```
// Window spec. No need to specify a frame in this case.
val wSpec3 = Window.partitionBy("name").orderBy("date")

// Use the lag function to look backwards by one row.
customers.withColumn("prevAmountSpent",
 lag(customers("amountSpent"), 1).over(wSpec3) ).show()
```

name	date	amountSpent	prevAmountSpent
Alice	5/1/2016	50	null
Alice	5/3/2016	45	50
Alice	5/4/2016	55	45
Bob	5/1/2016	25	null
Bob	5/4/2016	29	25
Bob	5/6/2016	27	29

Rank

- In this example, we want to know the order of a customer's visit (whether this is their first, second, or third visit).

// The rank function returns what we want.

```
customers.withColumn( "rank", rank().over(wSpec3) ).show()
```

name	date	amountSpent	rank
Alice	5/1/2016	50	1
Alice	5/3/2016	45	2
Alice	5/4/2016	55	3
Bob	5/1/2016	25	1
Bob	5/4/2016	29	2
Bob	5/6/2016	27	3

Case Study: Twitter Sentiment Analysis with Spark Streaming

Case Study: Twitter Sentiment Analysis

- Trending Topics can be used to create campaigns and attract larger audience. Sentiment Analytics helps in crisis management, service adjusting and target marketing.
- Sentiment refers to the emotion behind a social media mention online.
- Sentiment Analysis is categorising the tweets related to particular topic and performing data mining using Sentiment Automation Analytics Tools.
- We will be performing Twitter Sentiment Analysis as an Use Case or Spark Streaming.

Figure: *Facebook And Twitter Trending Topics*

Problem Statement

- To design a Twitter Sentiment Analysis System where we populate real-time sentiments for crisis management, service adjusting and target marketing.

Sentiment Analysis is used to:

- Predict the success of a movie
- Predict political campaign success
- Decide whether to invest in a certain company
- Targeted advertising
- Review products and services

Importing Packages

```
//Import the necessary packages into the Spark Program
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkContext._
import org.apache.spark.streaming.twitter._
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark._
import org.apache.spark.rdd._
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext._
import org.apache.spark.sql
import org.apache.spark.storage.StorageLevel
import scala.io.Source
import scala.collection.mutable.HashMap
import java.io.File
```

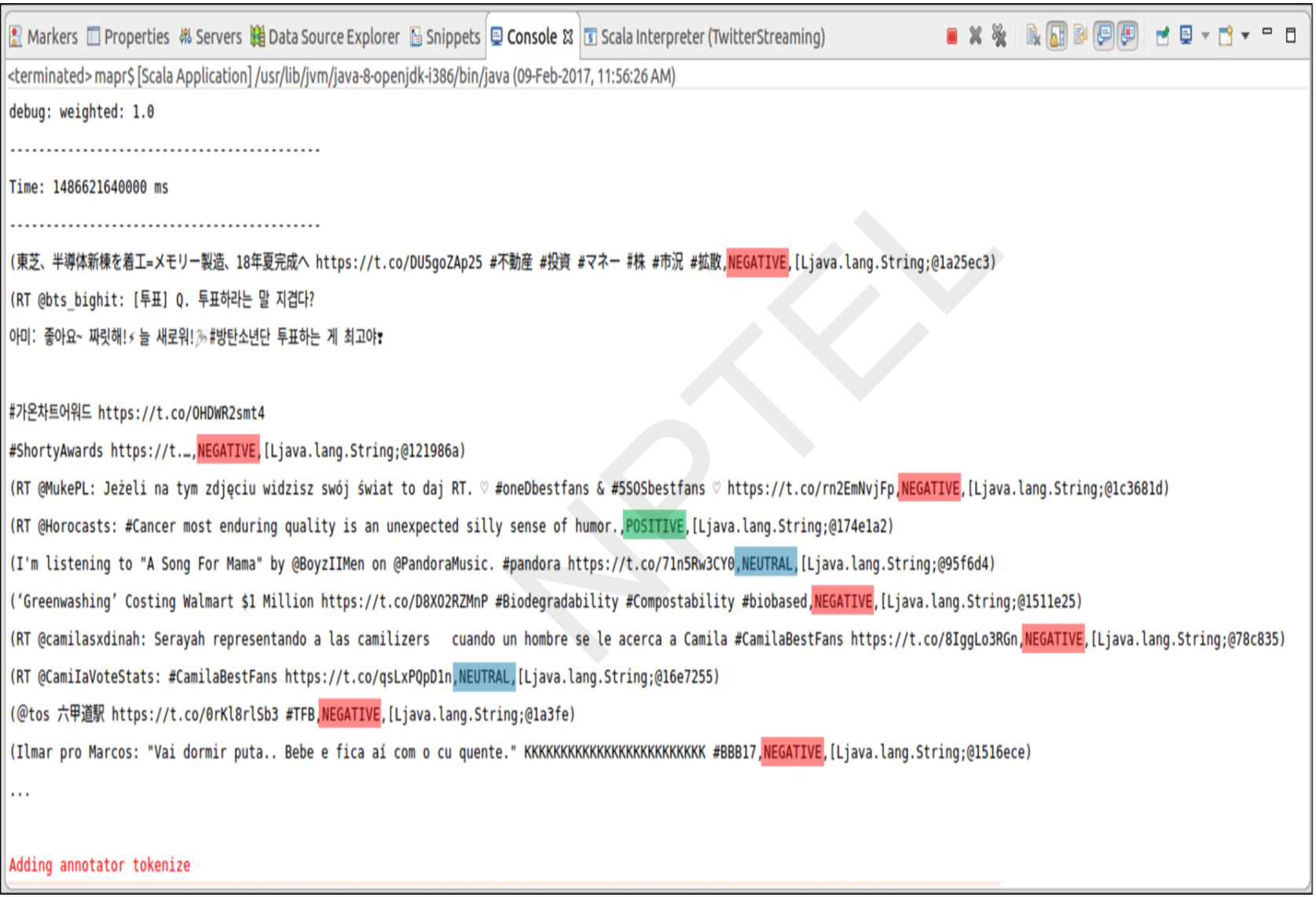
Twitter Token Authorization

```
object mapr {

 def main(args: Array[String]) {
 if (args.length < 4) {
 System.err.println("Usage: TwitterPopularTags <consumer key>
<consumer secret> " +
 "<access token> <access token secret> [<filters>]")
 System.exit(1)
 }

 StreamingExamples.setStreamingLogLevels()
 //Passing our Twitter keys and tokens as arguments for authorization
 val Array(consumerKey, consumerSecret, accessToken,
accessTokenSecret) = args.take(4)
 val filters = args.takeRight(args.length - 4)
```

DStream Transformation

```
// Set the system properties so that Twitter4j library used by twitter stream
// Use them to generate OAuth credentials
System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
System.setProperty("twitter4j.oauth.consumerSecret", consumerSecret)
System.setProperty("twitter4j.oauth.accessToken", accessToken)
System.setProperty("twitter4j.oauth.accessTokenSecret",
accessTokenSecret)

val sparkConf = new
SparkConf().setAppName("Sentiments").setMaster("local[2]")
val ssc = new StreamingContext(sparkConf, Seconds(5))
val stream = TwitterUtils.createStream(ssc, None, filters)

//Input DStream transformation using flatMap
val tags = stream.flatMap { status =>
status.getHashtagEntities.map(_.getText)}
```

Results

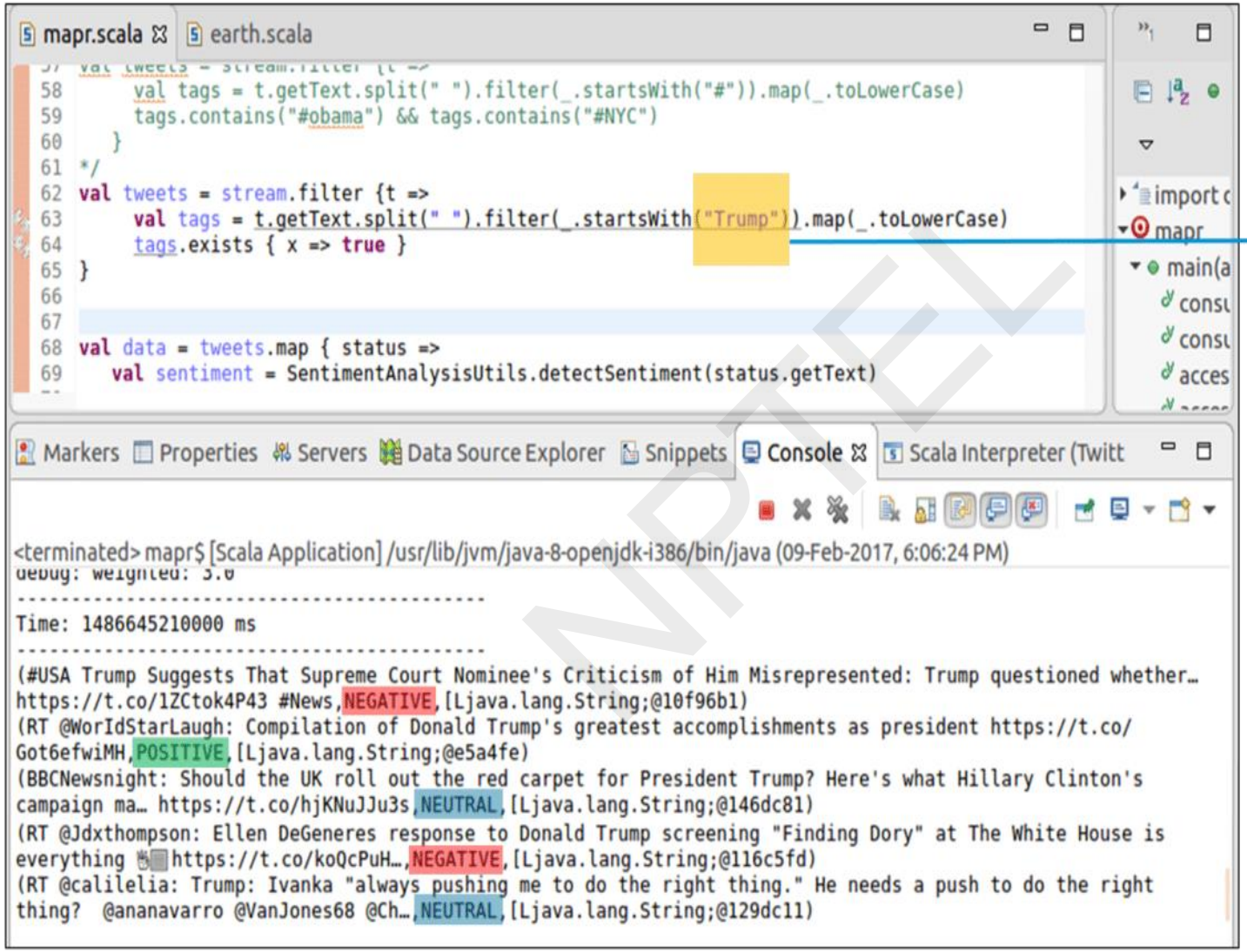


Positive

Neutral

Negative

Sentiment for Trump



Applying Sentiment Analysis

- As we have seen from our Sentiment Analysis demonstration, we can extract sentiments of particular topics just like we did for 'Trump'. Similarly, Sentiment Analytics can be used in crisis management, service adjusting and target marketing by companies around the world.

- Companies using Spark Streaming for Sentiment Analysis have applied the same approach to achieve the following:

1. Enhancing the customer experience
2. Gaining competitive advantage
3. Gaining Business Intelligence
4. Revitalizing a losing brand

References

- https://spark.apache.org/streaming/
- Streaming programming guide – spark.incubator.apache.org/docs/latest/streaming-programming-guide.html
- https://databricks.com/speaker/tathagata-das

Conclusion

- Stream processing framework that is ...
  - Scalable to large clusters
  - Achieves second-scale latencies
  - Has simple programming model
  - Integrates with batch & interactive workloads
  - Ensures efficient fault-tolerance in stateful computations

Introduction to Kafka

Data ingestion phase of Big data Computing

Kafka can be used for Data ingestion

Dr. Rajiv Misra

Dept. of Computer Science & Engg.

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Content of this Lecture:

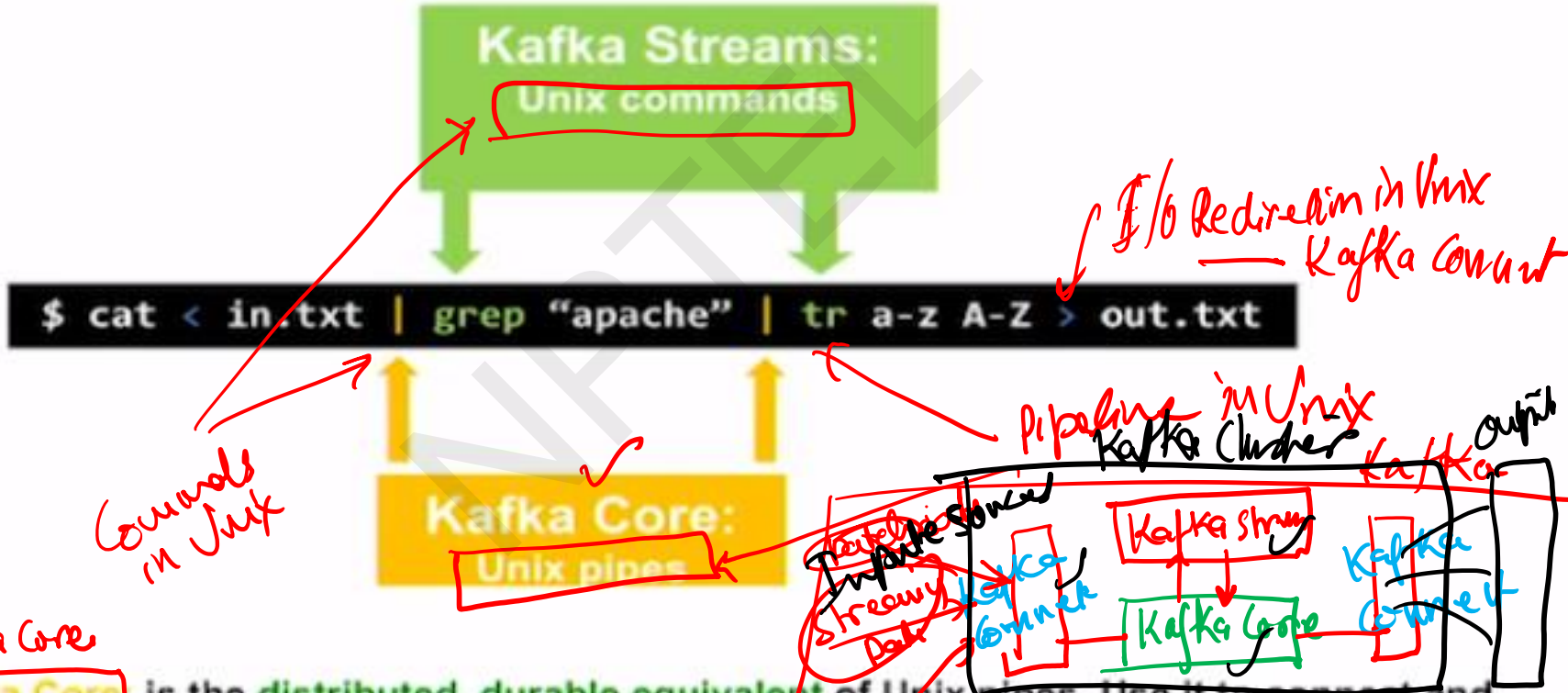
- Define Kafka
- Use cases for Kafka
- Kafka data model
- Kafka architecture
- Types of messaging systems
- Importance of brokers

Batch vs. Streaming

Batch

Streaming

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



- **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.**

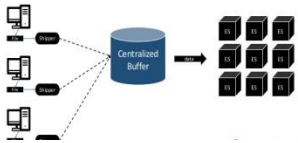
Kafka History

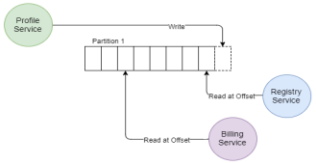
- Apache Kafka was originally developed by LinkedIn and later, handed over to the open source community in early 2011.
  - It became a main Apache project in October, 2012.
  - A stable Apache Kafka version 0.8.2.0 was release in Feb, 2015.
  - A stable Apache Kafka version 0.8.2.1 was released in May, 2015, which is the latest version.

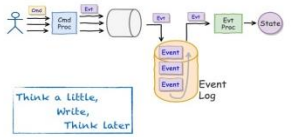
Kafka Use Cases

- 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, … (Stream data)
  - **IoT**: sensor readings, …

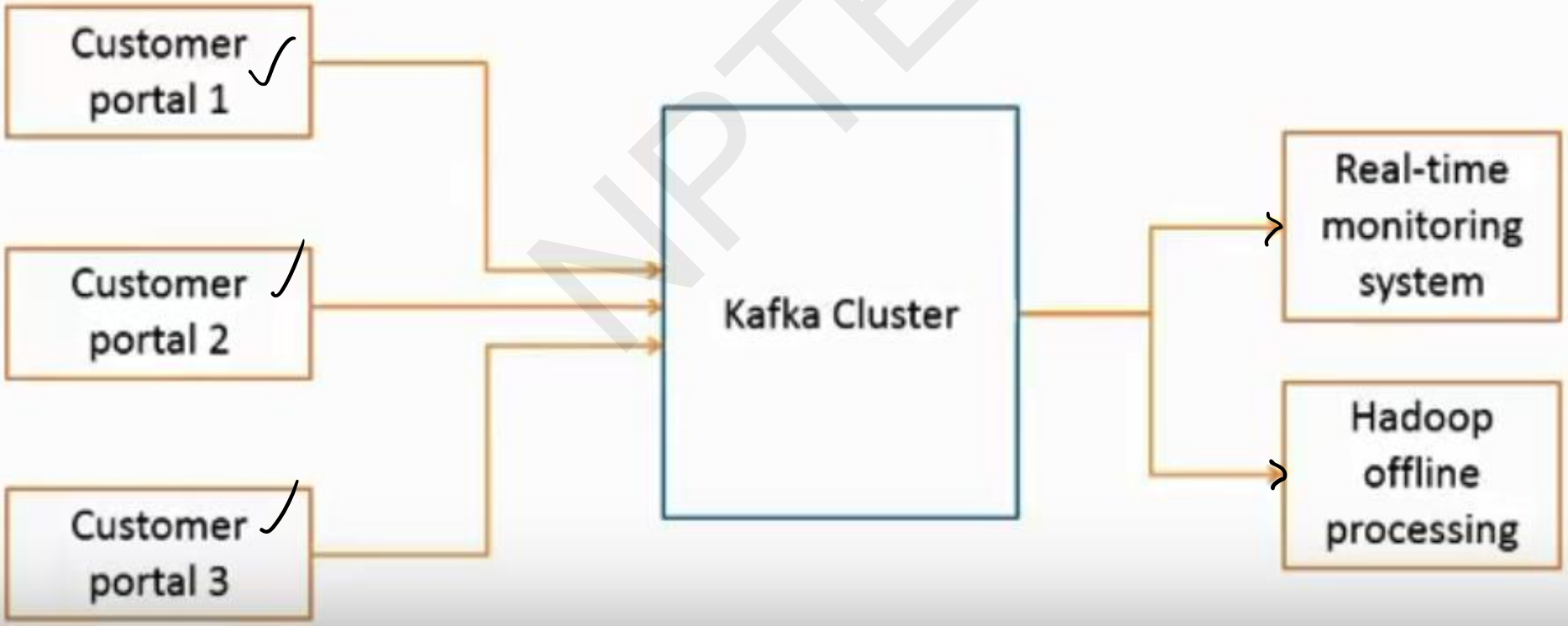
and so on.

Enter Kafka

- Adopted at 1000s of companies worldwide

Aggregating User Activity Using Kafka-Example

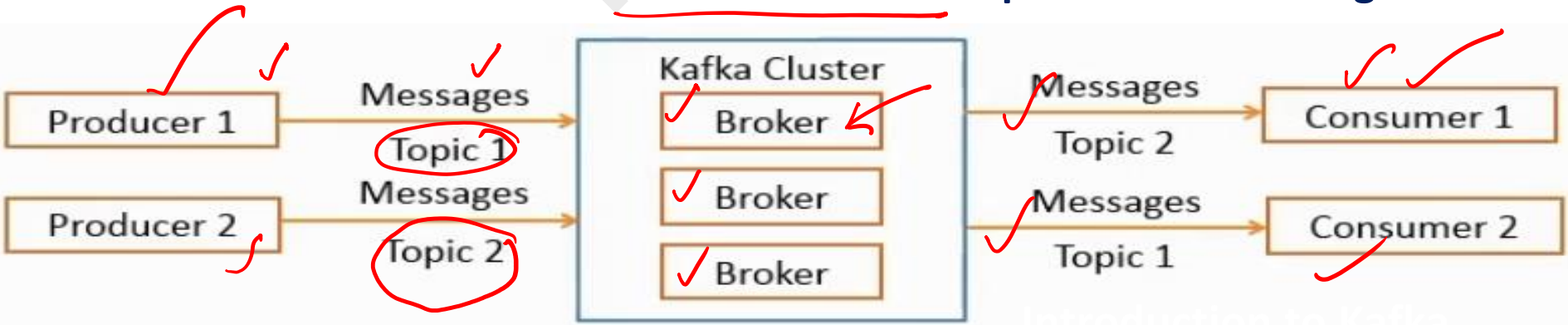
- Kafka can be used to aggregate user activity data such as clicks, navigation, and searches from different websites of an organization; such user activities can be sent to a real-time monitoring system and hadoop system for offline processing.



Kafka Data Model

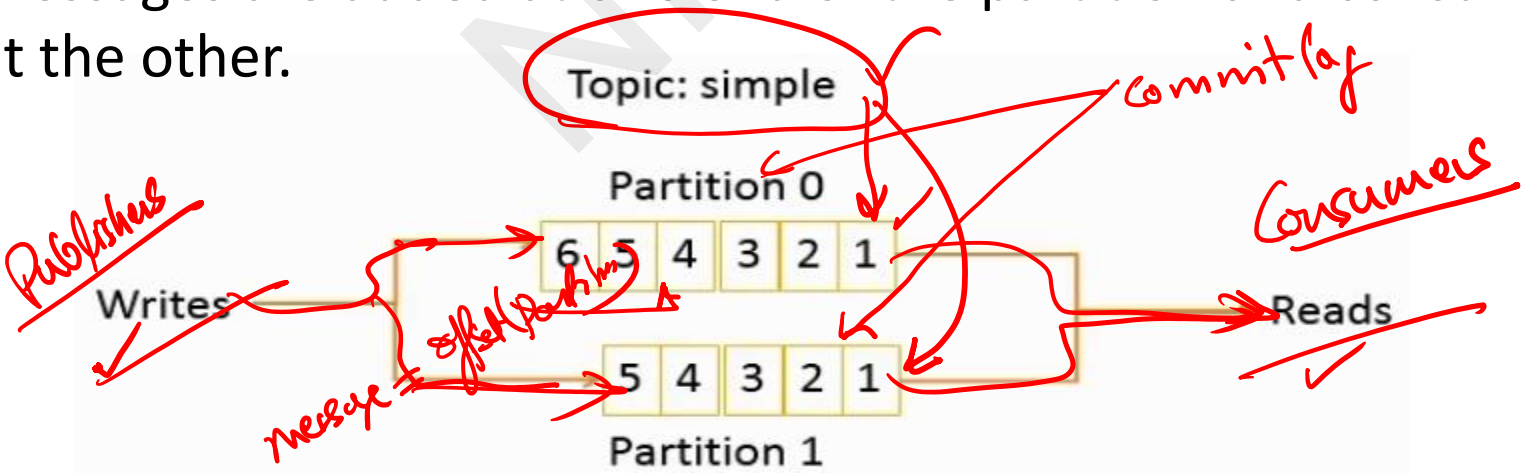
The Kafka data model consists of messages and topics.

- 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.



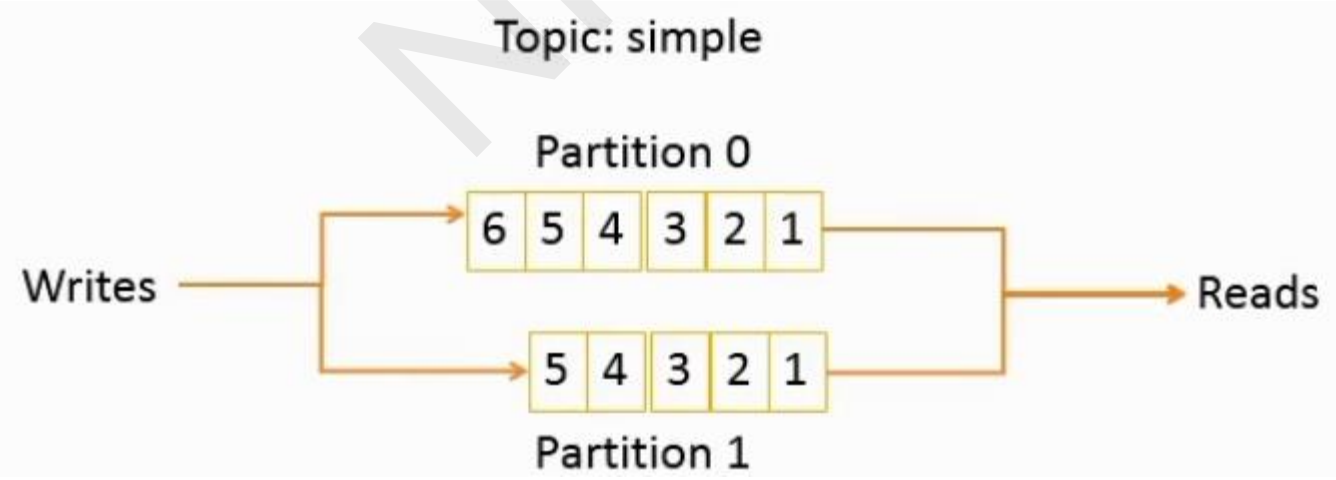
Topics (Data model in Kafka)

- 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.



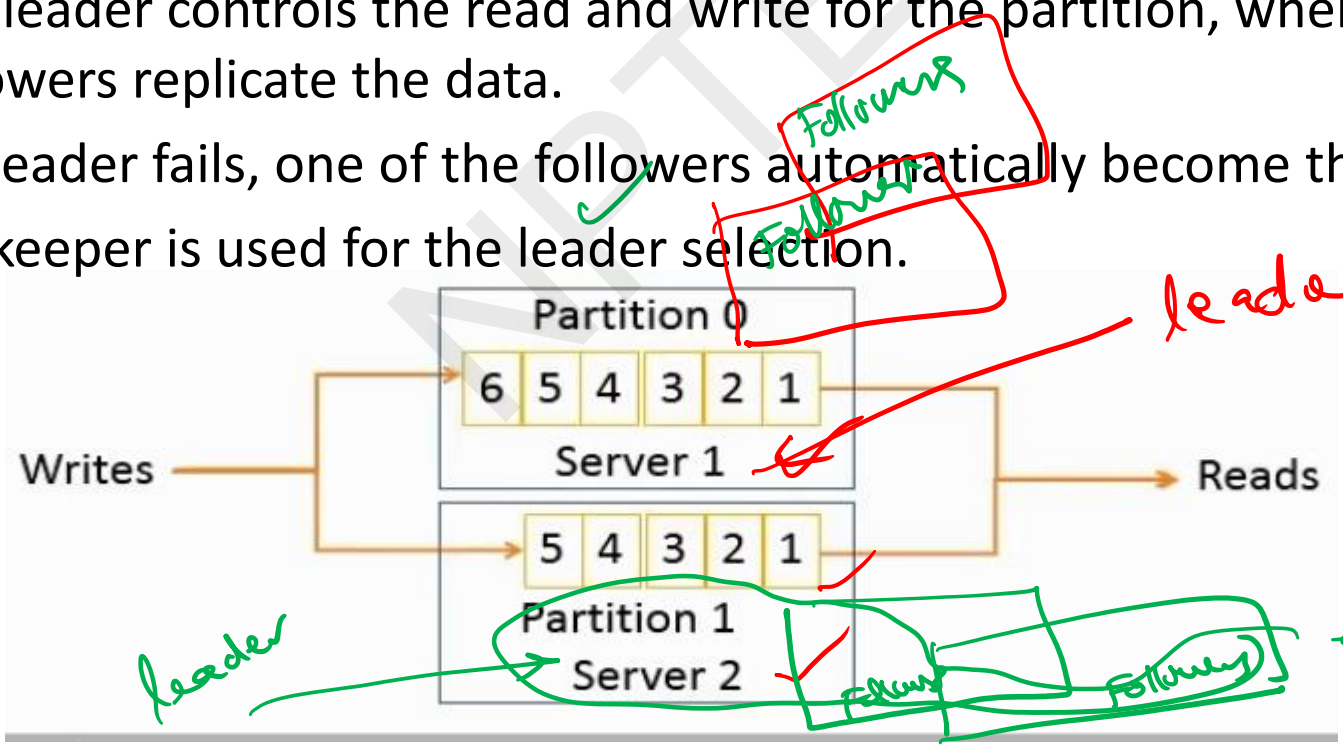
Partitions

- 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

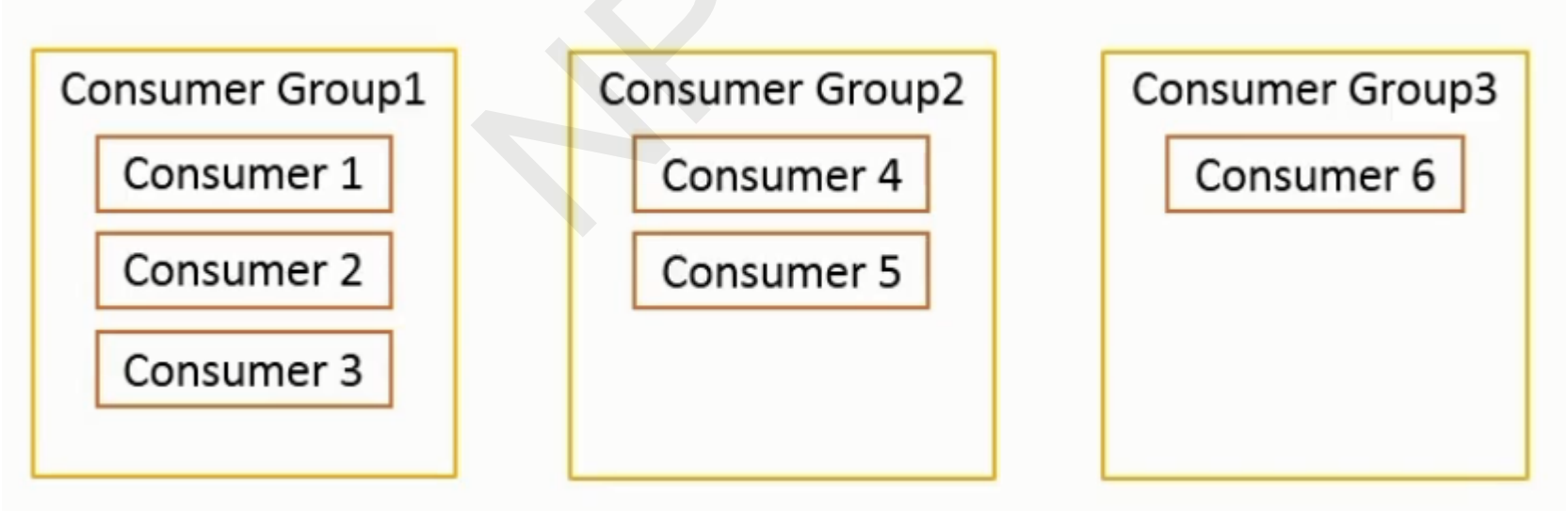
The producer is the creator of the message in Kafka.

- 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.

Consumers

The consumer is the receiver of the message in Kafka.

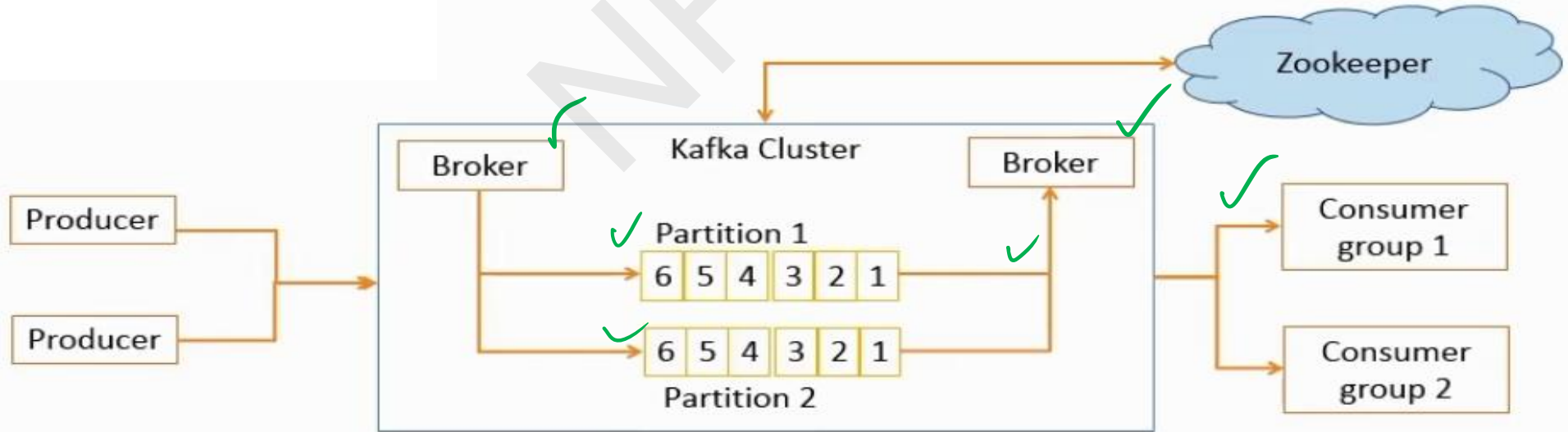
- Each consumer belongs to a consumer group.
- A consumer group may have one or more consumers.
- The consumers specify what topics they want to listen to.
- A message is sent to all the consumers in a consumer group.
- The consumer groups are used to control the messaging system.



Kafka Architecture

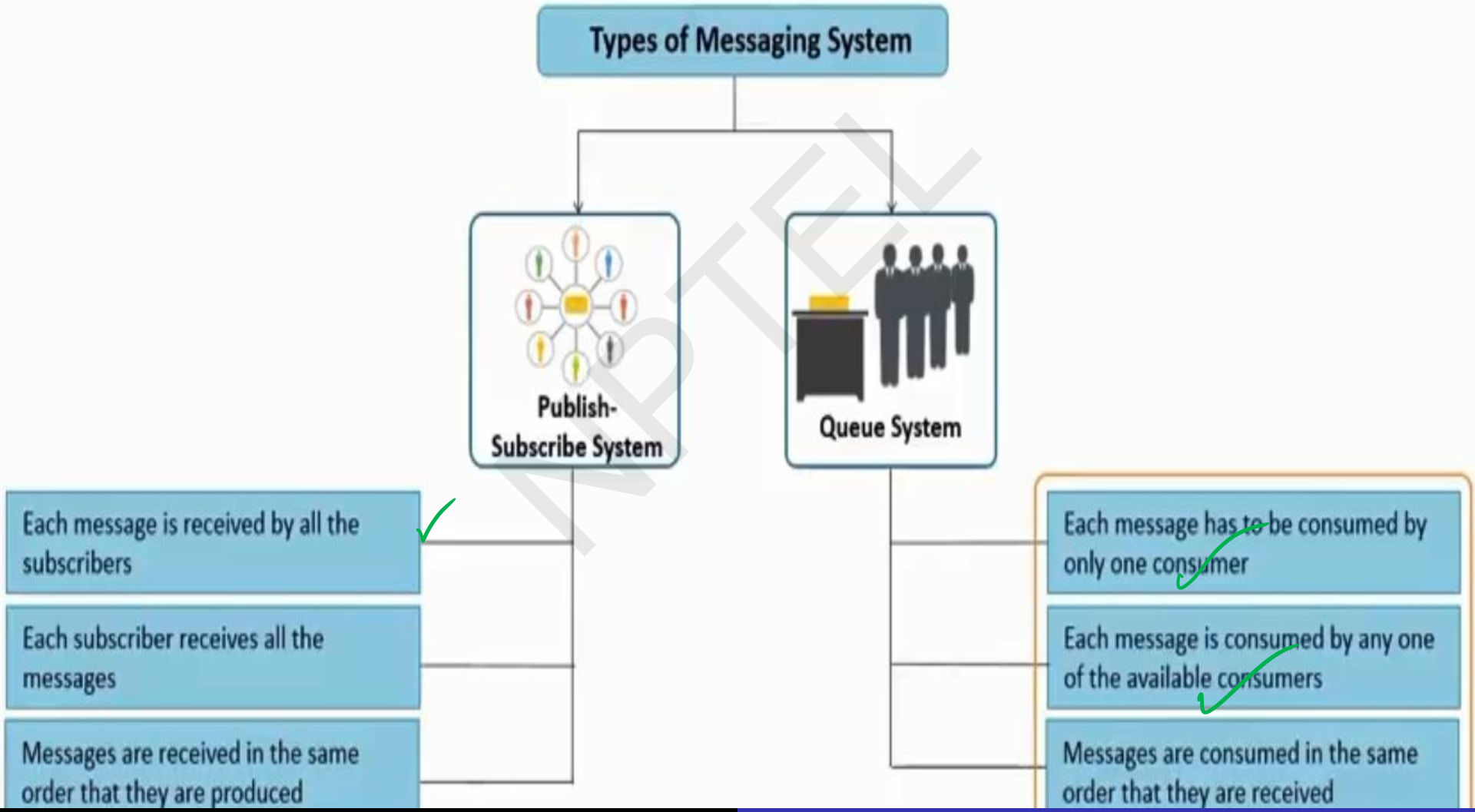
Kafka architecture consists of brokers that take messages from the producers and add to a partition of a topic. Brokers provide the messages to the consumers from the partitions.

- 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.

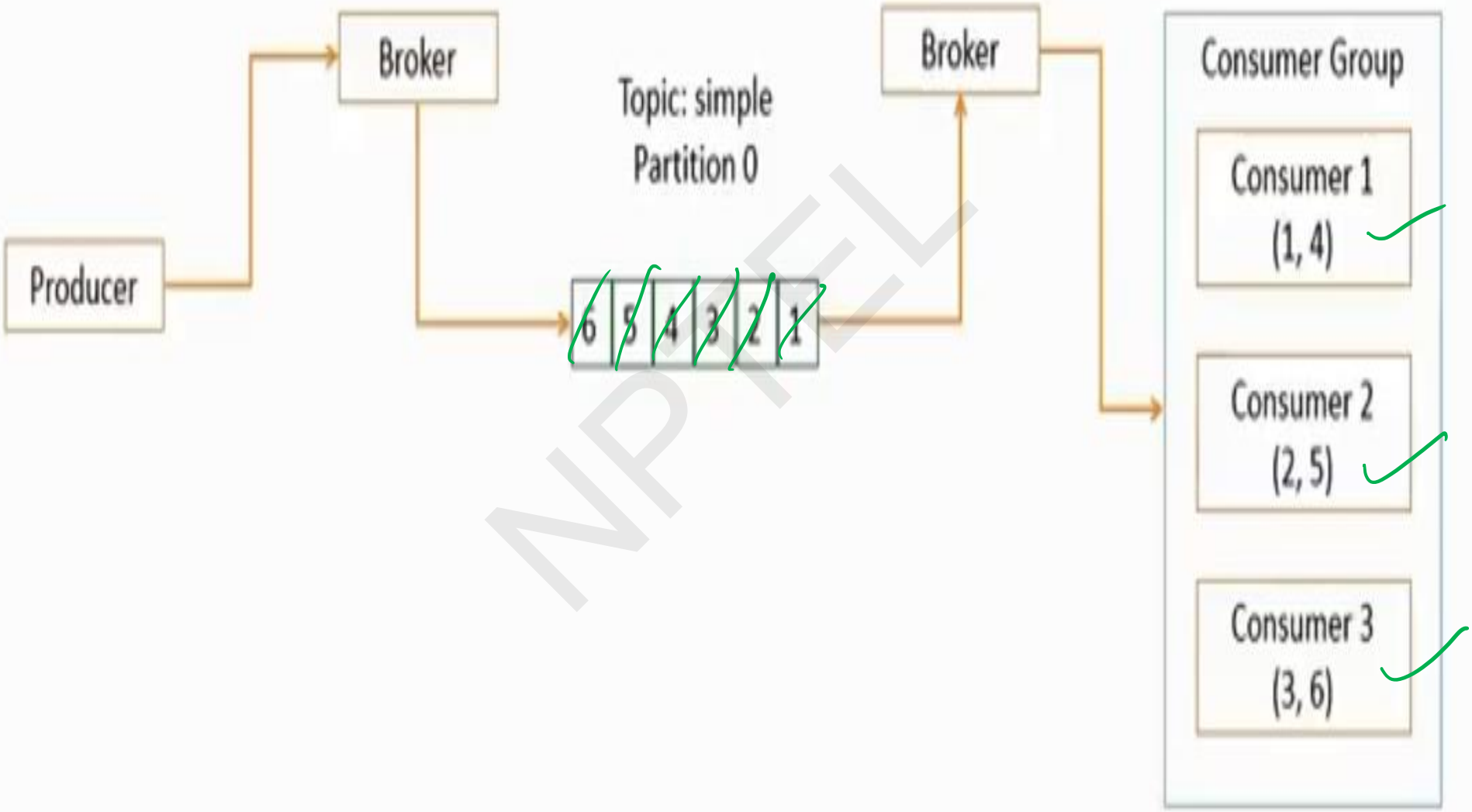


Types of Messaging Systems

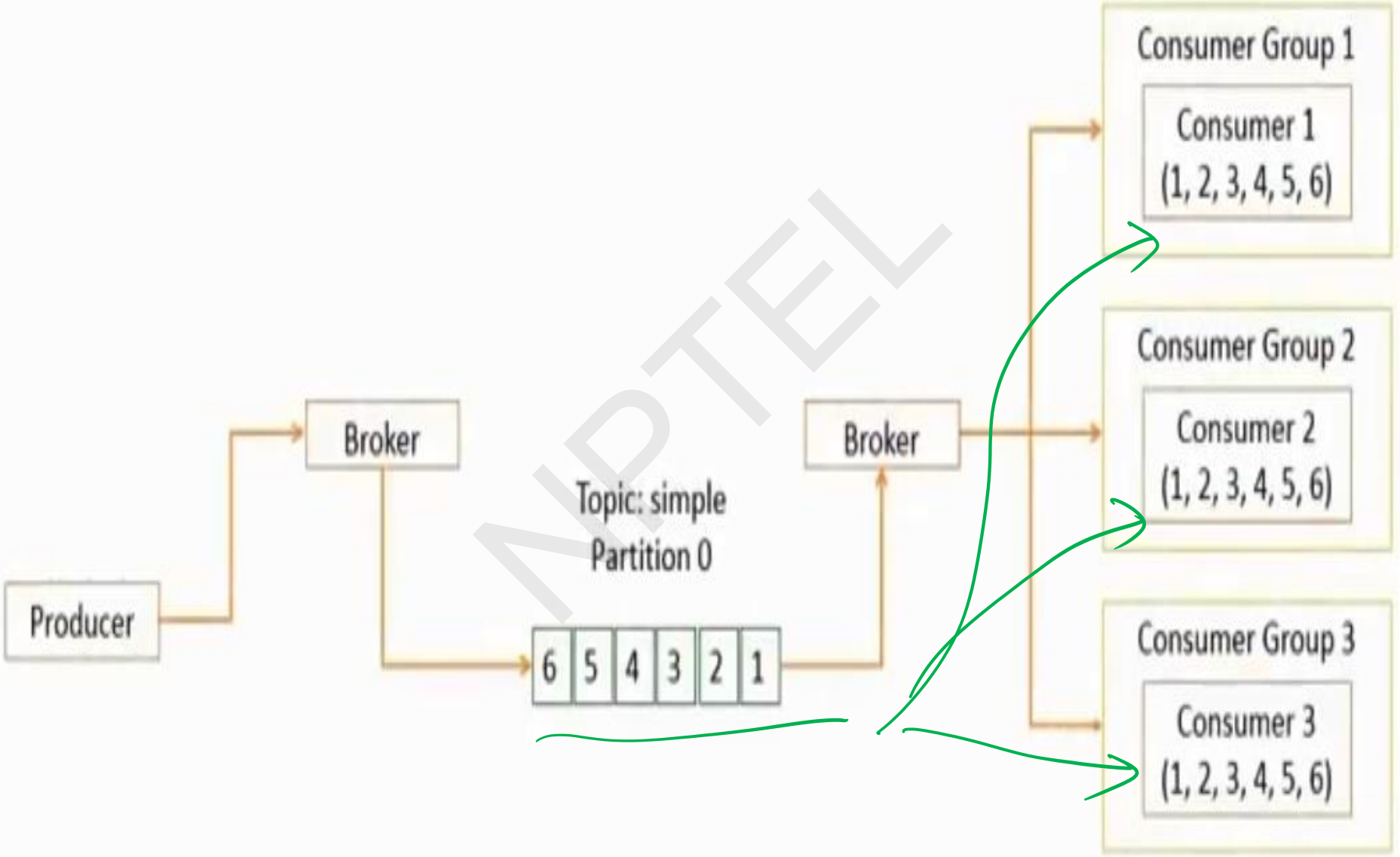
Kafka architecture supports the publish-subscribe and queue system.



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’

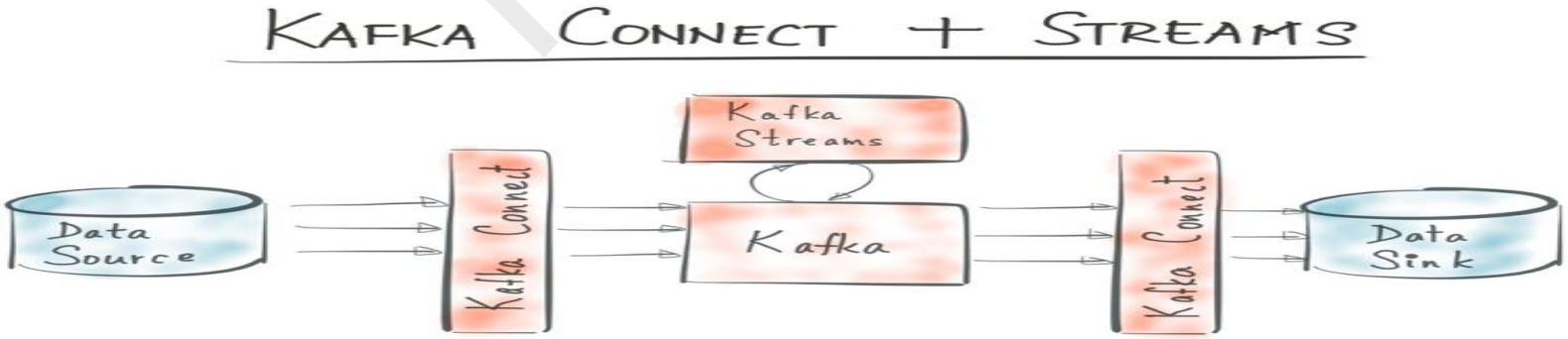
Persistence in Kafka

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.