COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 8: Spark III

Part 1: Spark SQL and DataFrames

Spark SQL Overview

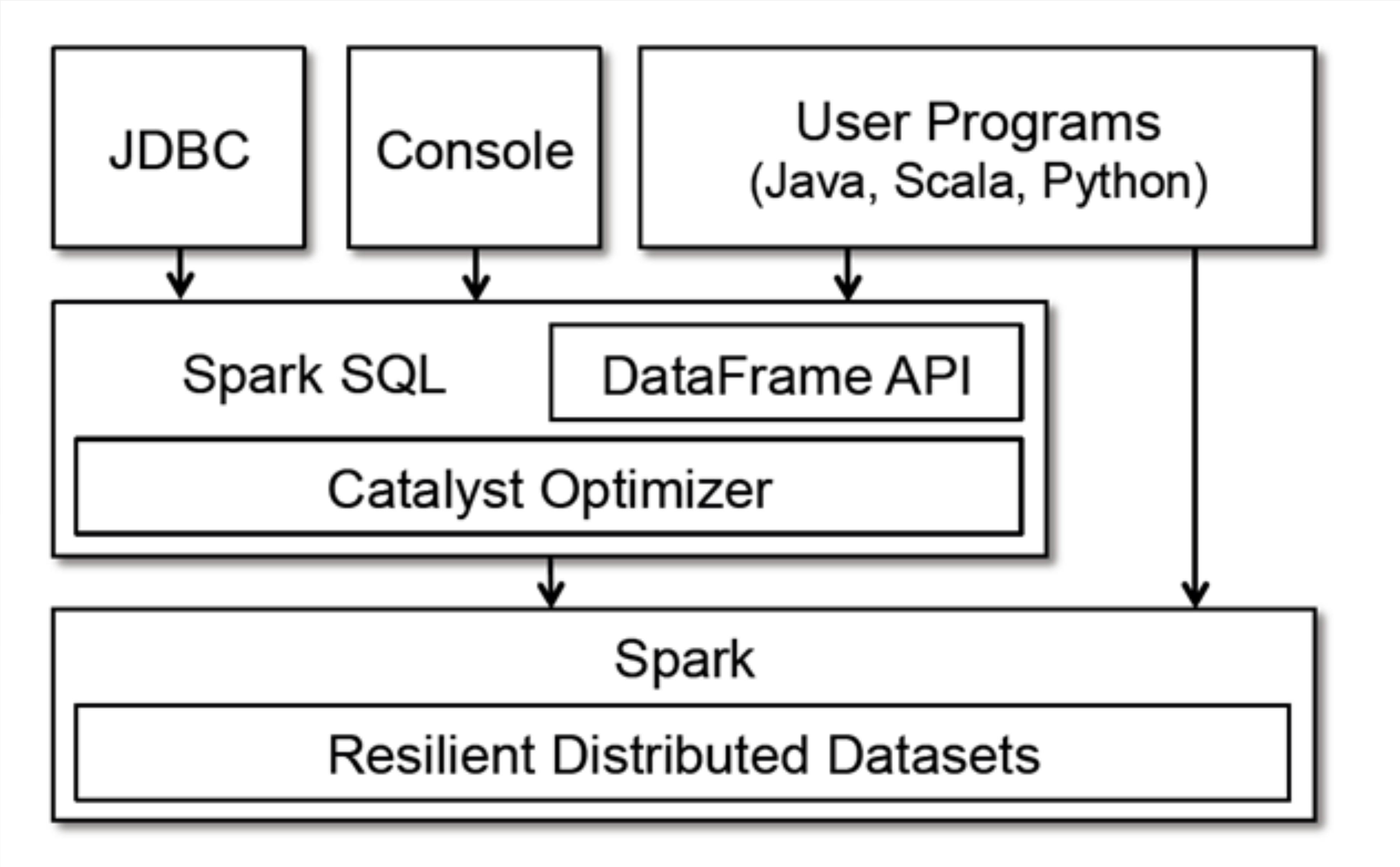
- Part of the core distribution since Spark 1.0 (April 2014)
- Tightly integrated way to work with structured data (tables with rows/columns)
- Transform RDDs using SQL
- Data source integration: Hive, Parquet, JSON, and more

- Spark SQL is **not** about SQL.
    - Aims to Create and Run Spark Programs Faster:

Relationship to Shark

- Shark has been subsumed by Spark SQL
- Shark modified the Hive backend to run over Spark, but had two challenges:
  - Limited integration with Spark programs
  - Hive optimizer not designed for Spark
- Spark SQL reuses the best parts of Shark:
  - Borrows
    - Hive data loading
    - In-memory column store
  - Adds
    - RDD-aware optimizer
    - Rich language interfaces

Spark Programming Interface



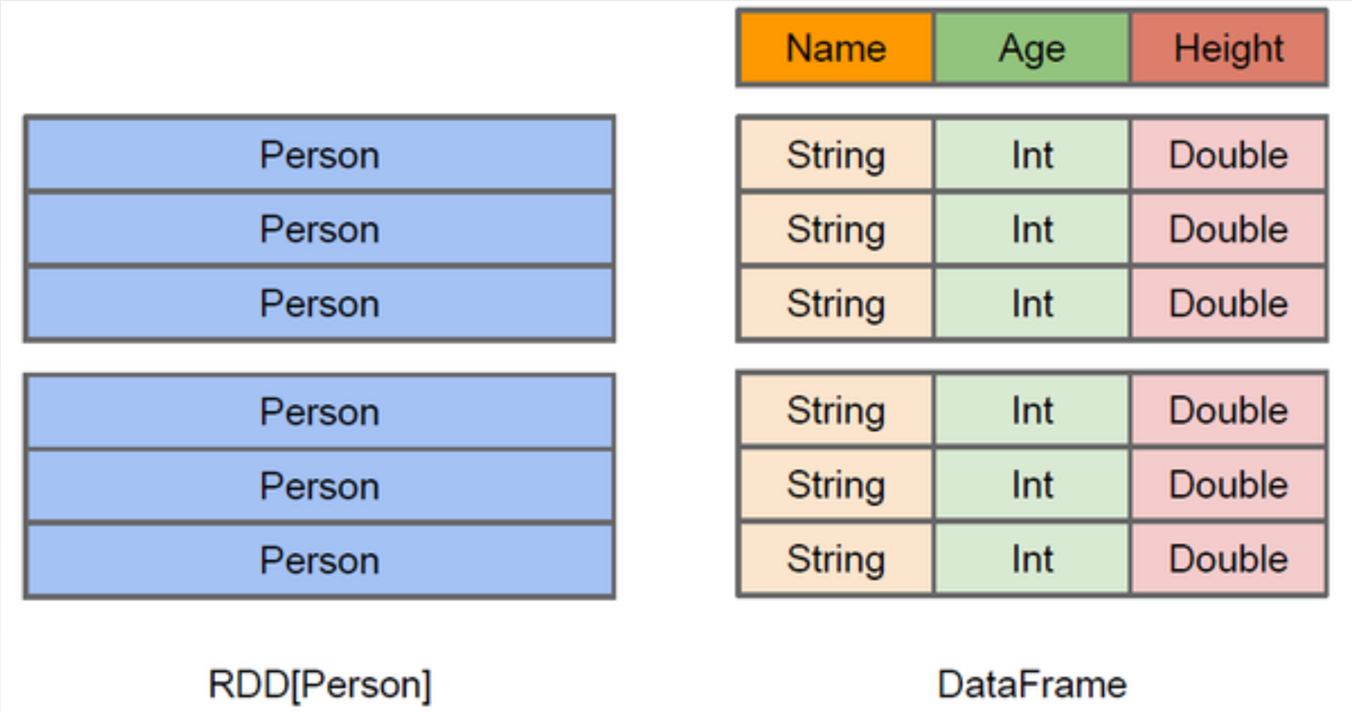
Datasets and DataFrames

- A *Dataset* is a distributed collection of data
  - provides the benefits of RDDs (e.g., strong typing) with the benefits of Spark SQL's optimized execution engine
  - A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, etc.)
  - The Dataset API is available in Scala and Java

- A *DataFrame* is a *Dataset* organized into named columns
  - It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations
  - An abstraction for selecting, filtering, aggregating and plotting structured data
  - In Scala and Java, a DataFrame is represented by a Dataset of Rows
    - Scala: DataFrame is simply a type alias of Dataset[Row]
    - Java: use Dataset<Row> to represent a DataFrame

Difference between DataFrame and RDD

- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema



Person
Person
Person

Person
Person
Person

RDD[Person]

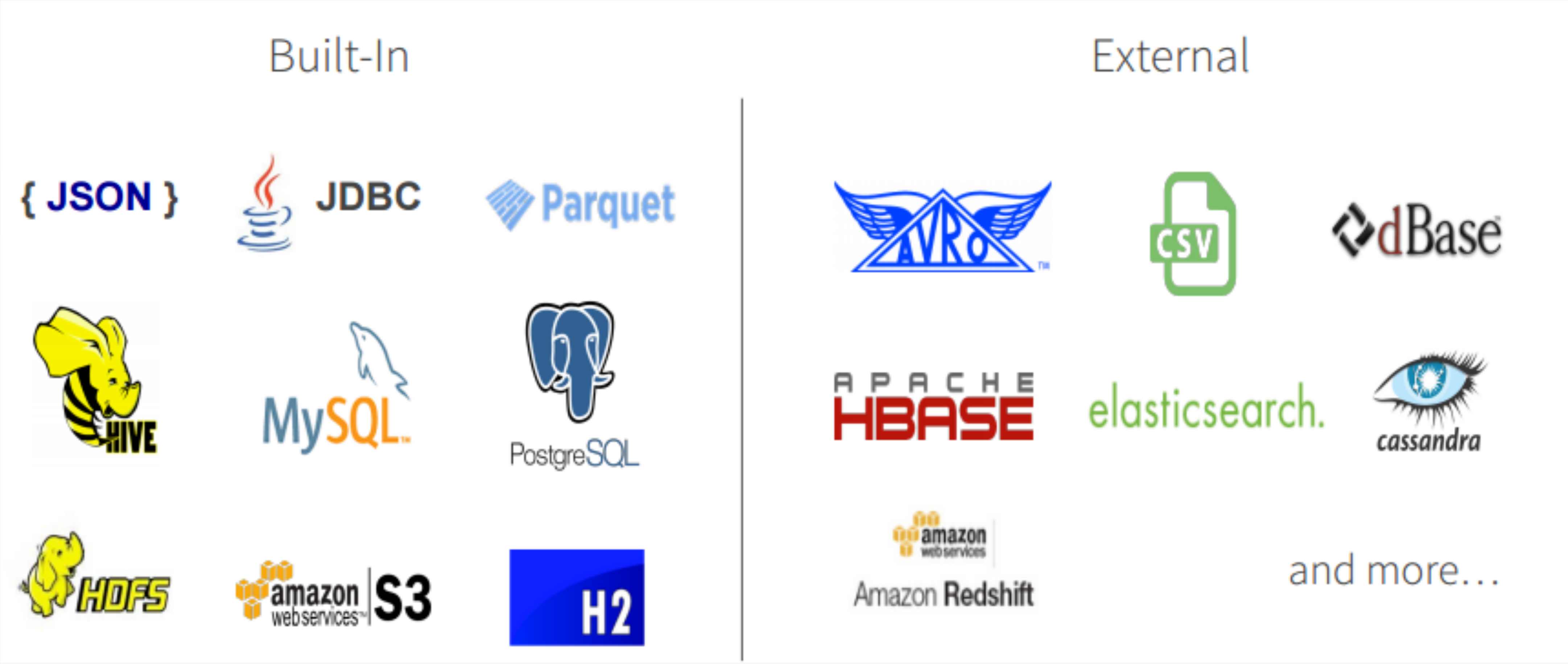
Name	Age	Height
String	Int	Double
String	Int	Double
String	Int	Double
String	Int	Double
String	Int	Double
String	Int	Double

DataFrame

  - RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of Person class
  - DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus Spark SQL query optimizer can target optimization

DataFrame Data Sources

- Spark SQL's Data Source API can read and write DataFrames using a variety of formats.
  - E.g., structured data files, tables in Hive, external databases, or existing RDDs
  - In the Scala API, DataFrame is simply a type alias of Dataset[Row]



Starting Point: SparkSession

- The entry point into all functionality in Spark is the SparkSession class.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

  - SparkSession in Spark 2.0 provides built-in support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables

Creating DataFrames

- With a SparkSession, applications can create DataFrames from *an existing RDD*, from *a Hive table*, or from *Spark data sources*.
  - creates a DataFrame based on the content of a JSON file:

```
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout

df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame Operations

- DataFrames are just Dataset of Rows in Scala and Java API.
  - These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets

```
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

DataFrame Operations

```
// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Running SQL Queries Programmatically

- The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Global Temporary View

- Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates

- Global temporary view: a temporary view that is shared among all sessions and keep alive until the Spark application terminates

- Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g. SELECT * FROM global_temp.view1

Global Temporary View Example

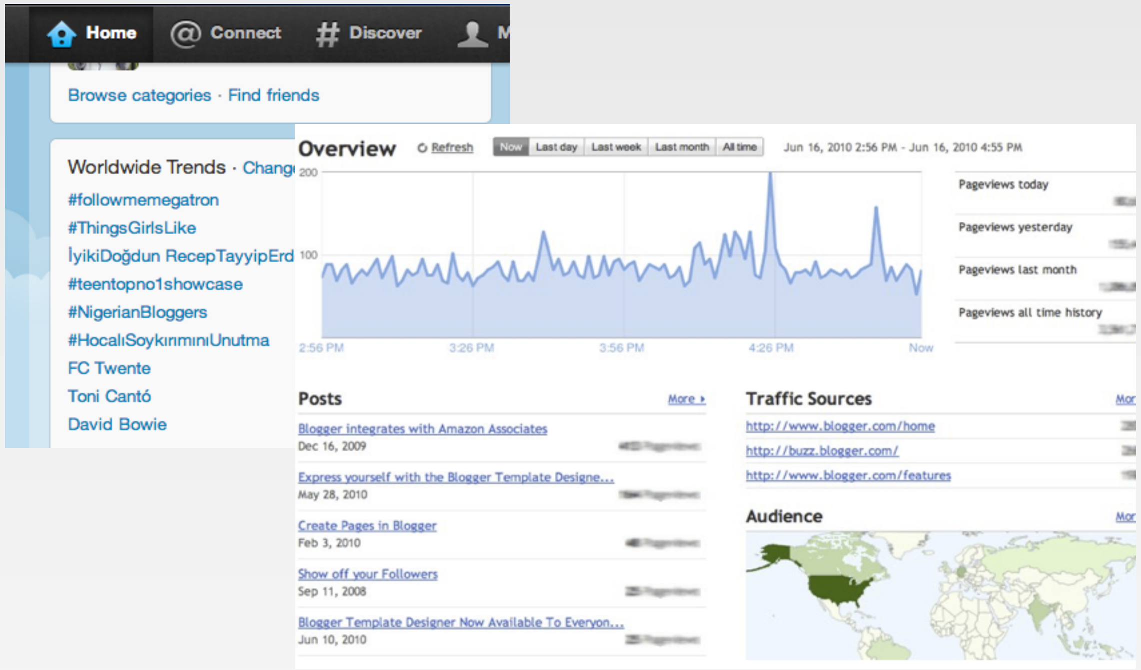
```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Find full example code at https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala

Part 2: Spark Streaming

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Ad impressions
  - … …



#followmemegatron
#ThingsGirlsLike
İyikiDoğdun RecepTayyipErd
#teentopno1showcase
#NigerianBloggers
#HocalıSoykırımınıUnutma
FC Twente
Toni Cantó
David Bowie

Blogger integrates with Amazon Associates
Dec 16, 2009

Express yourself with the Blogger Template Designe...
May 28, 2010

Create Pages in Blogger
Feb 3, 2010

Show off your Followers
Sep 11, 2008

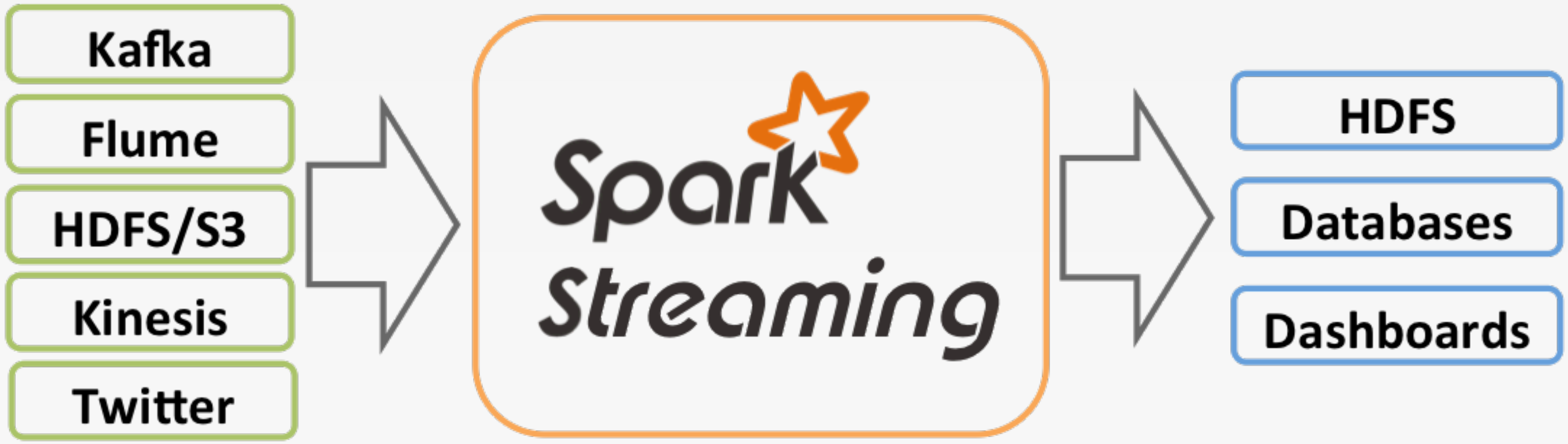
Blogger Template Designer Now Available To Everyon...
Jun 10, 2010

http://www.blogger.com/home
http://buzz.blogger.com/
http://www.blogger.com/features

- Distributed stream processing framework is required to
  - Scale to large clusters (100s of machines)
  - Achieve low latency (few seconds)

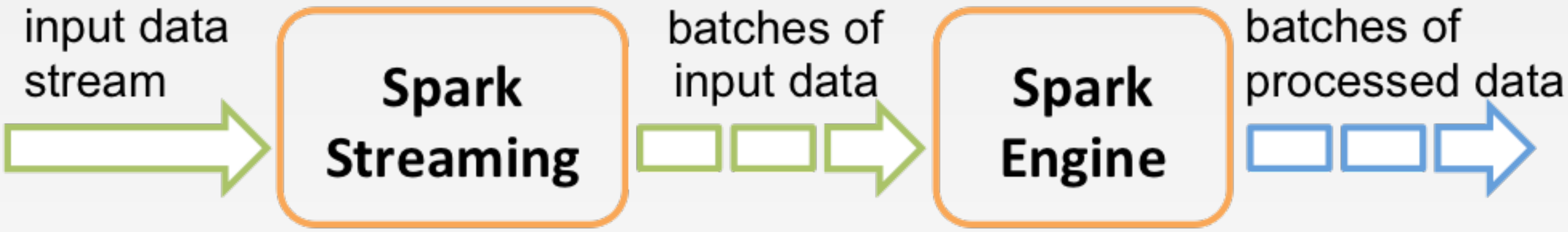
What is Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables **scalable**, **high-throughput**, **fault-tolerant** stream processing of live data streams
- Receive data streams from input sources, process them in a cluster, push out to filesystems, databases, and live dashboards
  - Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets
  - Data can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window
  - Processed data can be pushed out to filesystems, databases, and live dashboards



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



Spark Streaming Programming Model

- Spark Streaming provides a high-level abstraction called *discretized stream* (*Dstream*)
  - Represents a continuous stream of data.
  - DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams.
  - Internally, a DStream is represented as a sequence of RDDs.

- DStreams API very similar to RDD API
  - Functional APIs in Scala, Java
  - Create input DStreams from different sources
  - Apply parallel operations

An Example: Streaming WordCount

- Use StreamingContext, rather then SparkContext

```
import org.apache.spark._
import org.apache.spark.streaming._

object NetworkWordCount {
        val conf = new
        SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
        val ssc = new StreamingContext(conf, Seconds(10))

        val lines = ssc.socketTextStream("localhost", 9999)
        val words = lines.flatMap(_.split(" "))
        val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
        wordCounts.print()
        ssc.start()
        ssc.awaitTermination()
}
```

Streaming WordCount

- Linking with Apache Spark
  - The first step is to explicitly import the required spark classes into your Spark program

```
import org.apache.spark._
import org.apache.spark.streaming._
```

- Create a local StreamingContext with two working thread and batch interval of 10 second.

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(10))
```

  - A **StreamingContext** object has to be created which is the main entry point of all Spark Streaming functionality.
  - At least two local threads must be used (two cores)
  - Do the count for each 10 seconds
    - The batch interval must be set based on the latency requirements of your application and available cluster resources

Streaming WordCount

- After a streaming context is defined, you have to do the following:
  - Define the input sources by creating input DStreams.
  - Define the streaming computations by applying transformation and output operations to DStreams.
  - Start receiving data and processing it using streamingContext.start().
  - Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
  - The processing can be manually stopped using streamingContext.stop().

Streaming WordCount

- Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

```
val lines = ssc.socketTextStream("localhost", 9999)
```

  - This lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text.

- Split the lines by space characters into words and do the count

```
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
```

- No real processing has started yet now. To start the processing after all the transformations have been setup, we finally call

```
ssc.start()
ssc.awaitTermination()
```

- The complete code can be found in the Spark Streaming example NetworkWordCount.

Linking the Application

- Add the following dependency to your SBT configuration:
  - libraryDependencies += "org.apache.spark" %% "spark-streaming_2.11" % "2.3.0"

- For data sources like Kafka, Flume, and Kinesis that are not present in the Spark Streaming core API, you will have to add the corresponding artifact spark-streaming-xyz_2.11 to the dependencies

Kafka	spark-streaming-kafka-0-8_2.11
Flume	spark-streaming-flume_2.11
Kinesis	spark-streaming-kinesis-asl_2.11 [Amazon Software License]

Run Streaming WordCount

- First need to run Netcat (a small utility found in most Unix-like systems) as a data server by using:

```
$ nc -lk 9999
```

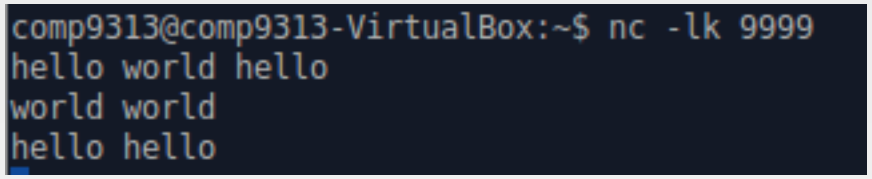
- sbt configuration file:

```
name := “Network WordCount"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-
streaming" % "2.3.0"
```

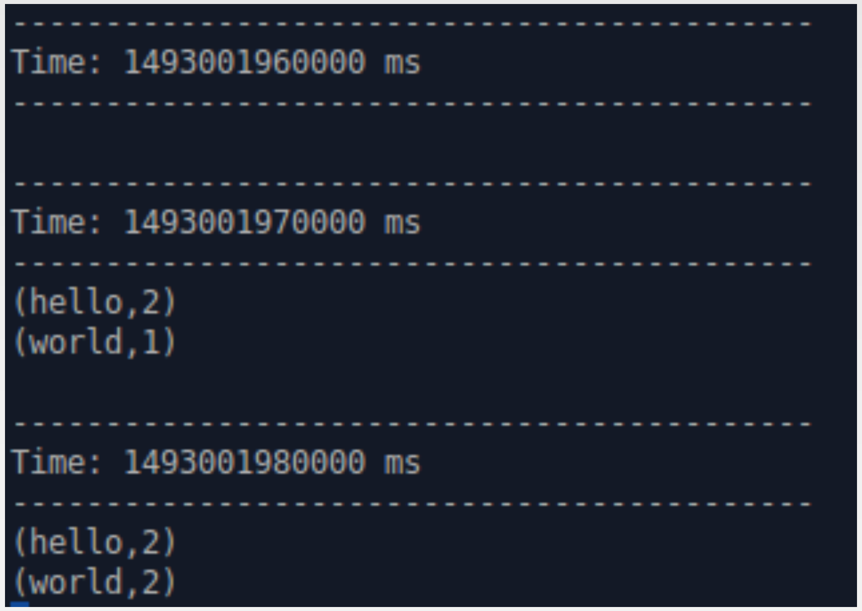
- Then, in a different terminal, you can start the example by using

```
$ spark-submit --class NetworkWordCount ~/sparkapp/target/scala-
2.11/network-wordcount_2.11-1.0.jar
```

Results of Streaming WordCount



```
comp9313@comp9313-VirtualBox:~$ nc -lk 9999
hello world hello
world world
hello hello
```



```
-------------------------------------------
Time: 1493001960000 ms
-------------------------------------------

-------------------------------------------
Time: 1493001970000 ms
-------------------------------------------
(hello,2)
(world,1)

-------------------------------------------
Time: 1493001980000 ms
-------------------------------------------
(hello,2)
(world,2)
```

- The first 10 seconds, receives no data
- The next 10 seconds, receives one line
- The last 10 seconds, receives two lines

Get Streaming Data From Files

- Spark streaming can also get streaming data from a specified file folder
- It can only ingest files that have been moved to the directory
- Create a local StreamingContext with two working thread and batch interval of 10 second.

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(10))
```

- Using this context, we can create a DStream that represents streaming data from files in a folder

```
val lines = ssc.textFileStream(“file:///home/comp9313/logs”)
```

- Next do similarly as in NetworkWordCount

Get Streaming Data From Files

- The first 10 seconds, receives no data
- The next 10 seconds, do:

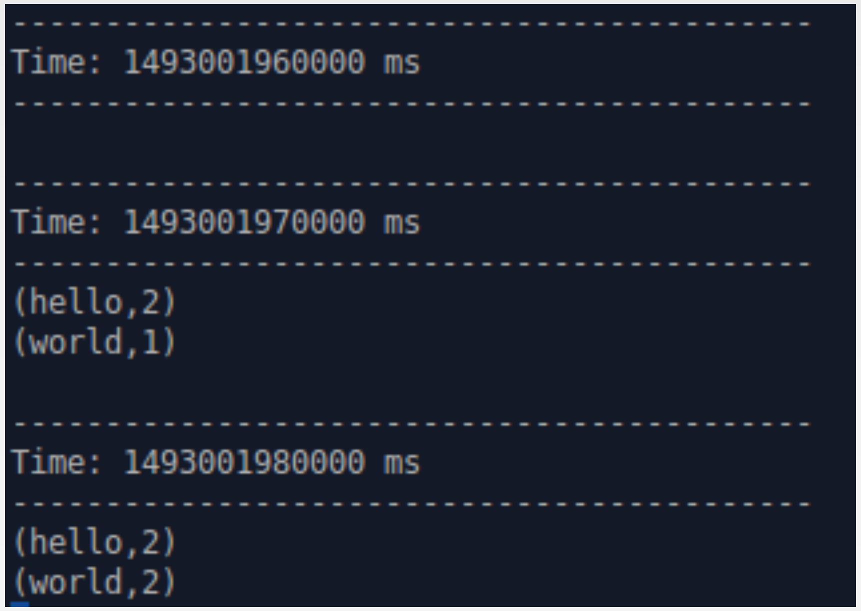
  echo “hello hello world” > log1

  mv log1 ~/logs
- The next 10 seconds, do:

  echo “hello world” > log2

  echo “hello world” > log3

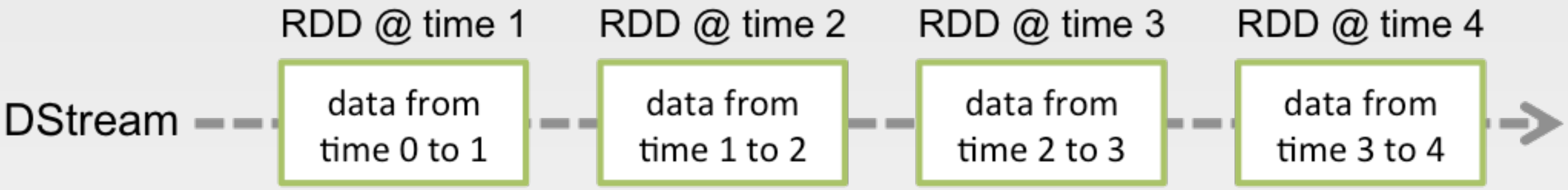
  mv log2 log3 ~/logs



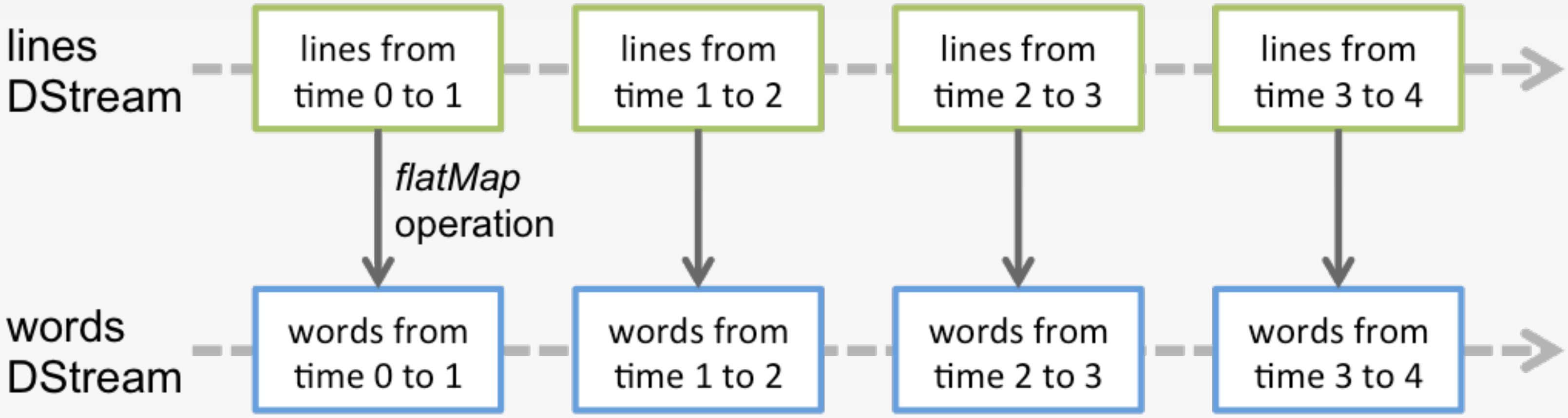
```
-------------------------------------------
Time: 1493001960000 ms
-------------------------------------------

-------------------------------------------
Time: 1493001970000 ms
-------------------------------------------
(hello,2)
(world,1)

-------------------------------------------
Time: 1493001980000 ms
-------------------------------------------
(hello,2)
(world,2)
```

Discretized Streams (DStreams)

- A DStream is represented by a continuous series of RDDs
  - Each RDD in a DStream contains data from a certain interval, as shown in the following figure.



- Any operation applied on a DStream translates to operations on the underlying RDDs.
  - in the earlier example of converting a stream of **lines** to **words**, the flatMap operation is applied on each RDD in the **lines** DStream to generate the RDDs of the **words** DStream



Input DStreams and Receivers

- Input DStreams are DStreams representing the stream of input data received from streaming sources.
  - E.g., **lines** was an input DStream as it represented the stream of data received from the netcat server
- Every input DStream is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing
- Spark Streaming provides two categories of built-in streaming sources.
  - *Basic sources*: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
  - *Advanced sources*: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies

Input DStreams and Receivers

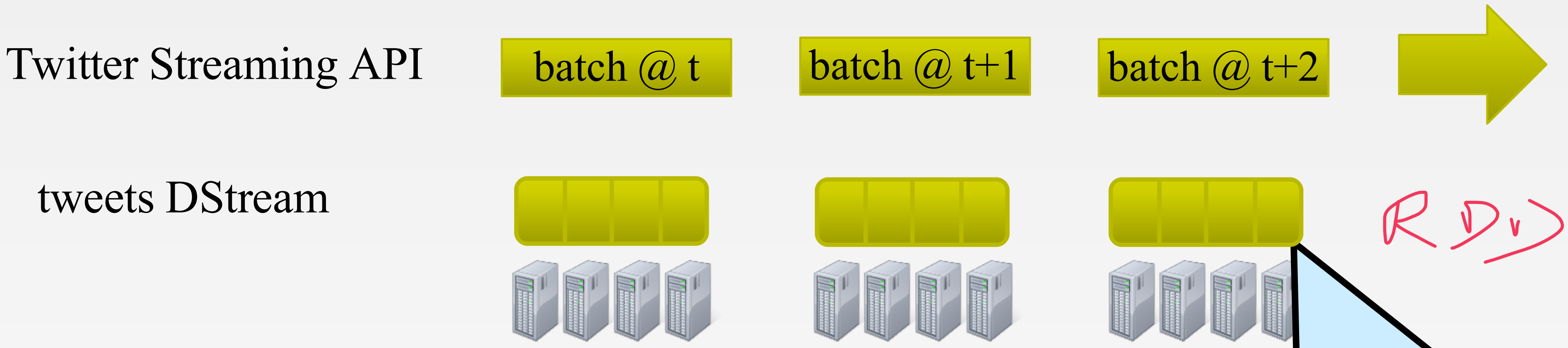
- When running a Spark Streaming program locally, do not use “local” or “local[1]” as the master URL
  - If you are using an input DStream based on a receiver (e.g., sockets), then the single thread will be used to run the receiver, leaving no thread for processing the received data
  - When running locally, always use “local[*n*]” as the master URL, where *n* > number of receivers to run

- The number of cores allocated to the Spark Streaming application must be more than the number of receivers. Otherwise the system will only receive data, but not be able to process it

Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(conf, Seconds(10))
val tweets :DStream[Status] = TwitterUtils.createStream(ssc, None)
```

DStream: a sequence of RDDs representing a stream of data



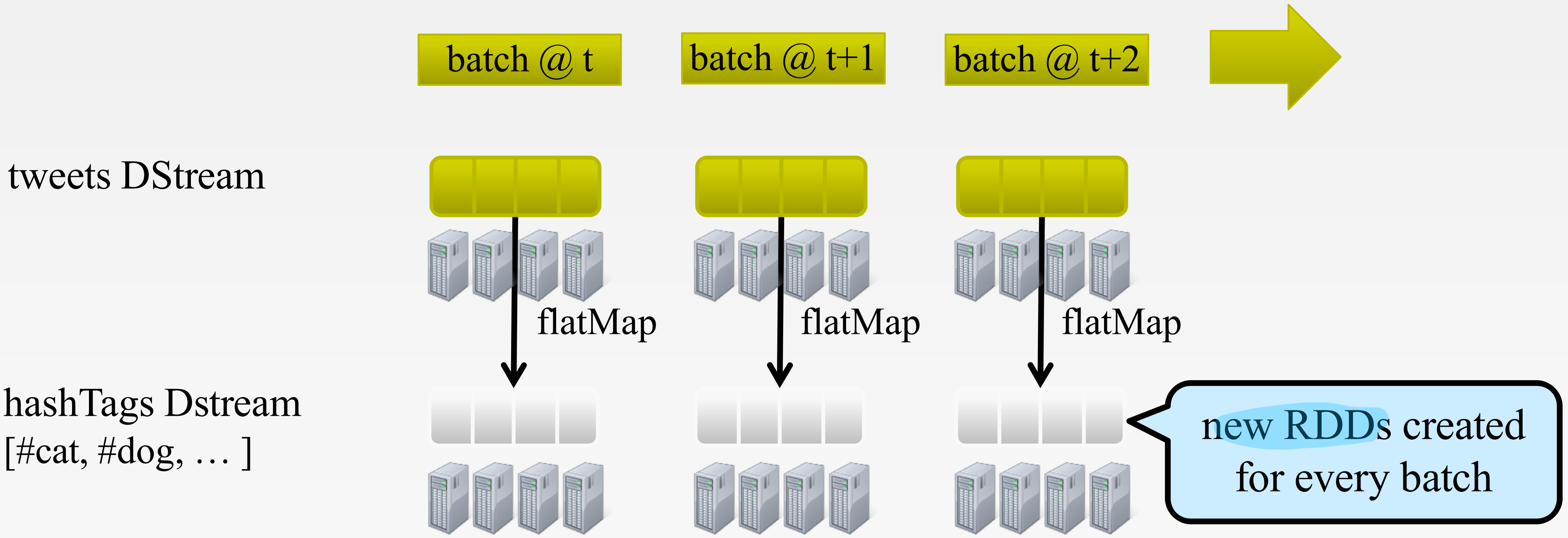
stored in memory as an RDD
(immutable, distributed)

Example – Get hashtags from Twitter

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

new DStream

transformation: modify data in one DStream to create another DStream

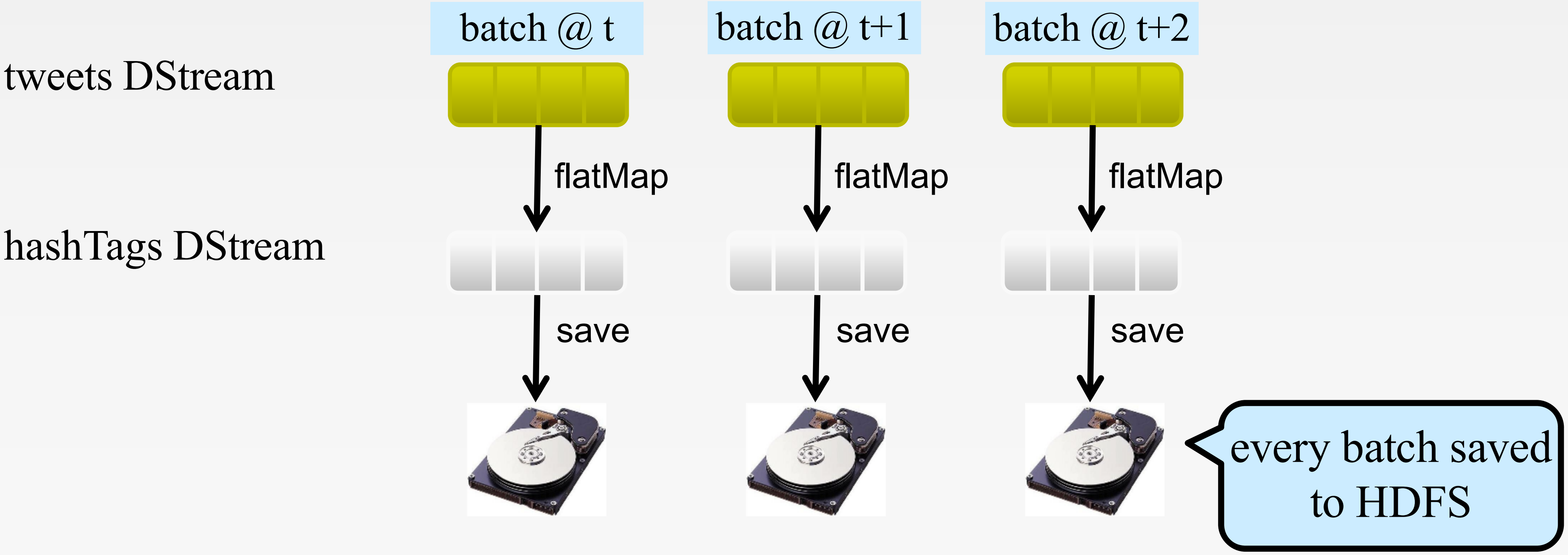


new RDDs created for every batch

Example – Get hashtags from Twitter

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

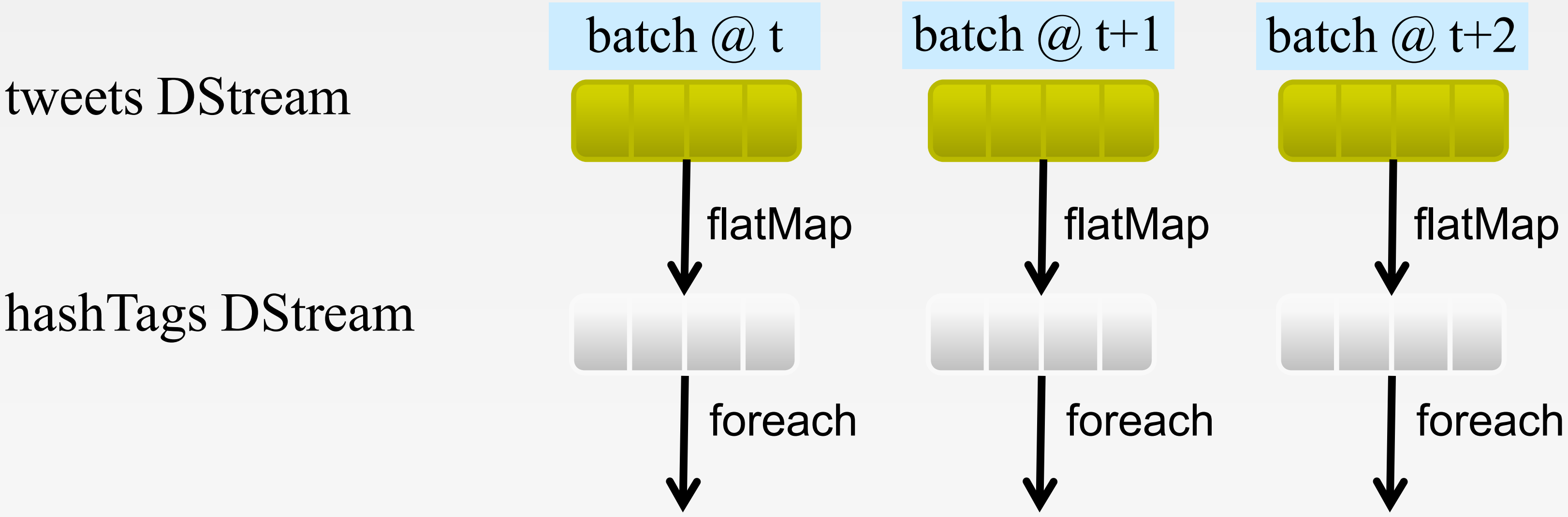
output operation: to push data to external storage



Example – Get hashtags from Twitter

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

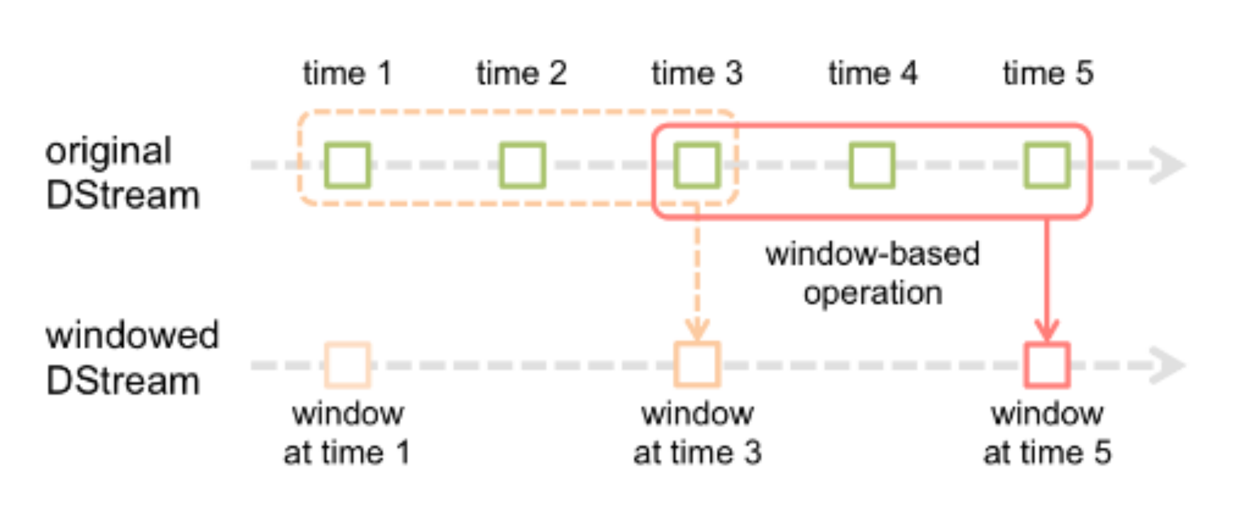
foreach: do whatever you want with the processed data



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

Window Operations

- Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data



- Every time the window *slides* over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.
  - E.g., the operation is applied over the last 3 time units of data, and slides by 2 time units
  - Any window operation needs to specify two parameters
    - *window length* - The duration of the window (3 in the figure).
    - *sliding interval* - The interval at which the window operation is performed (2 in the figure).

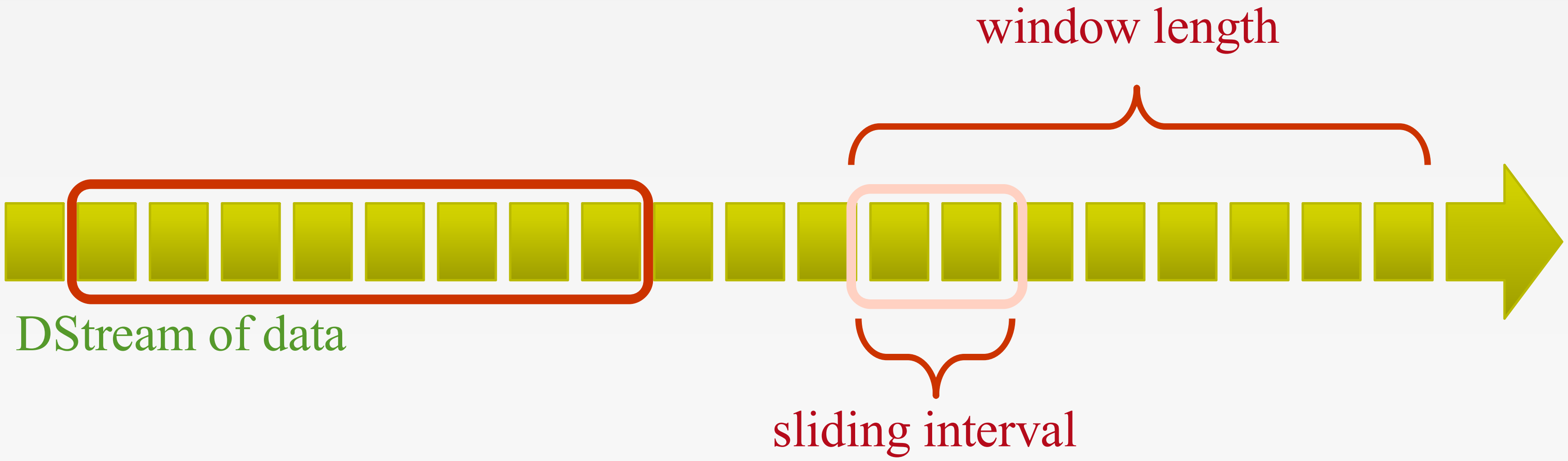
Window-based Transformations

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

sliding window operation

window length

sliding interval



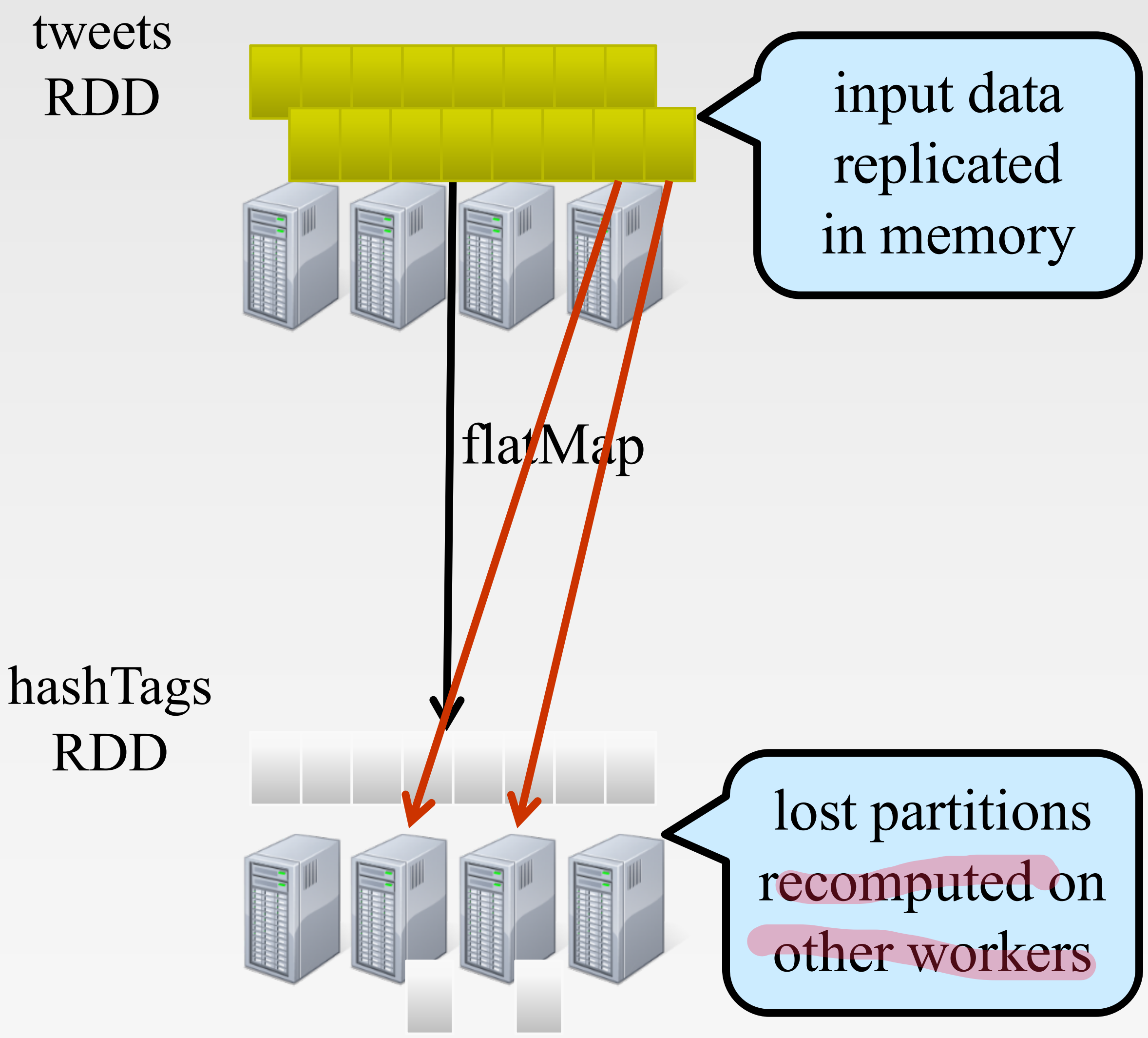
Fault-tolerance: Worker

- RDDs remember the operations that created them

- 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 transformed data is fault-tolerant, and exactly-once transformations



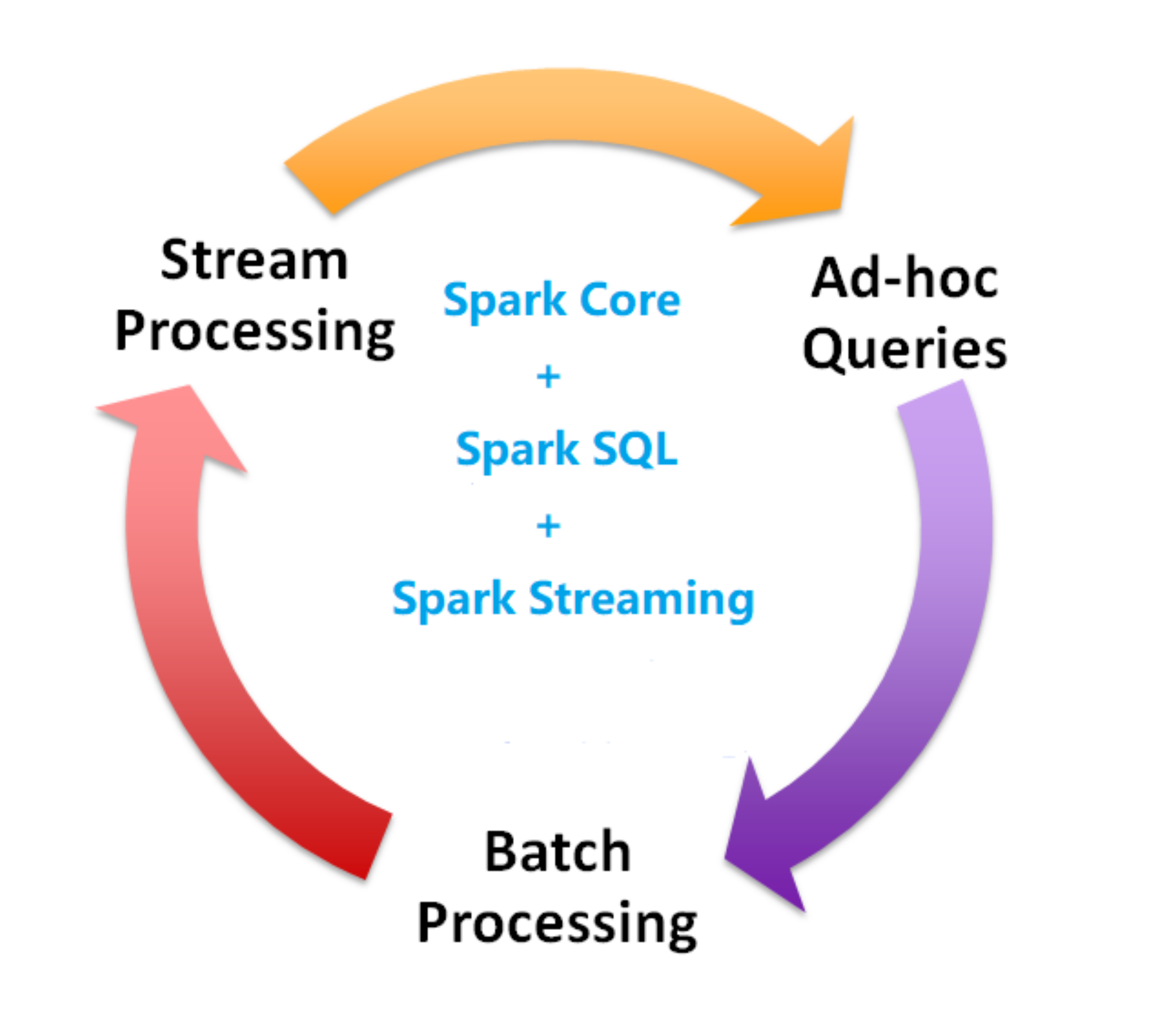
input data replicated in memory

lost partitions recomputed on other workers

Fault-tolerance: Master

- Master saves the state of the DStreams to a checkpoint file
  - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file
- More information in the Spark Streaming guide
- Automated master fault recovery coming soon

Vision - one stack to rule them all



Part 3: Spark Structured Streaming

Spark SQL + Streaming

Structured Streaming

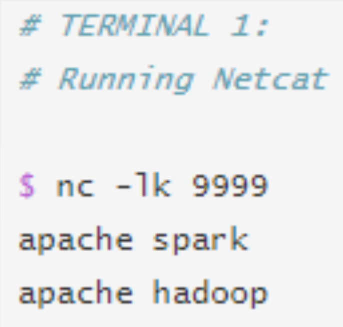
- Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.
  - Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.
  - You can express your streaming computation the same way you would express a batch computation on static data.
  - The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

An Example: Streaming WordCount

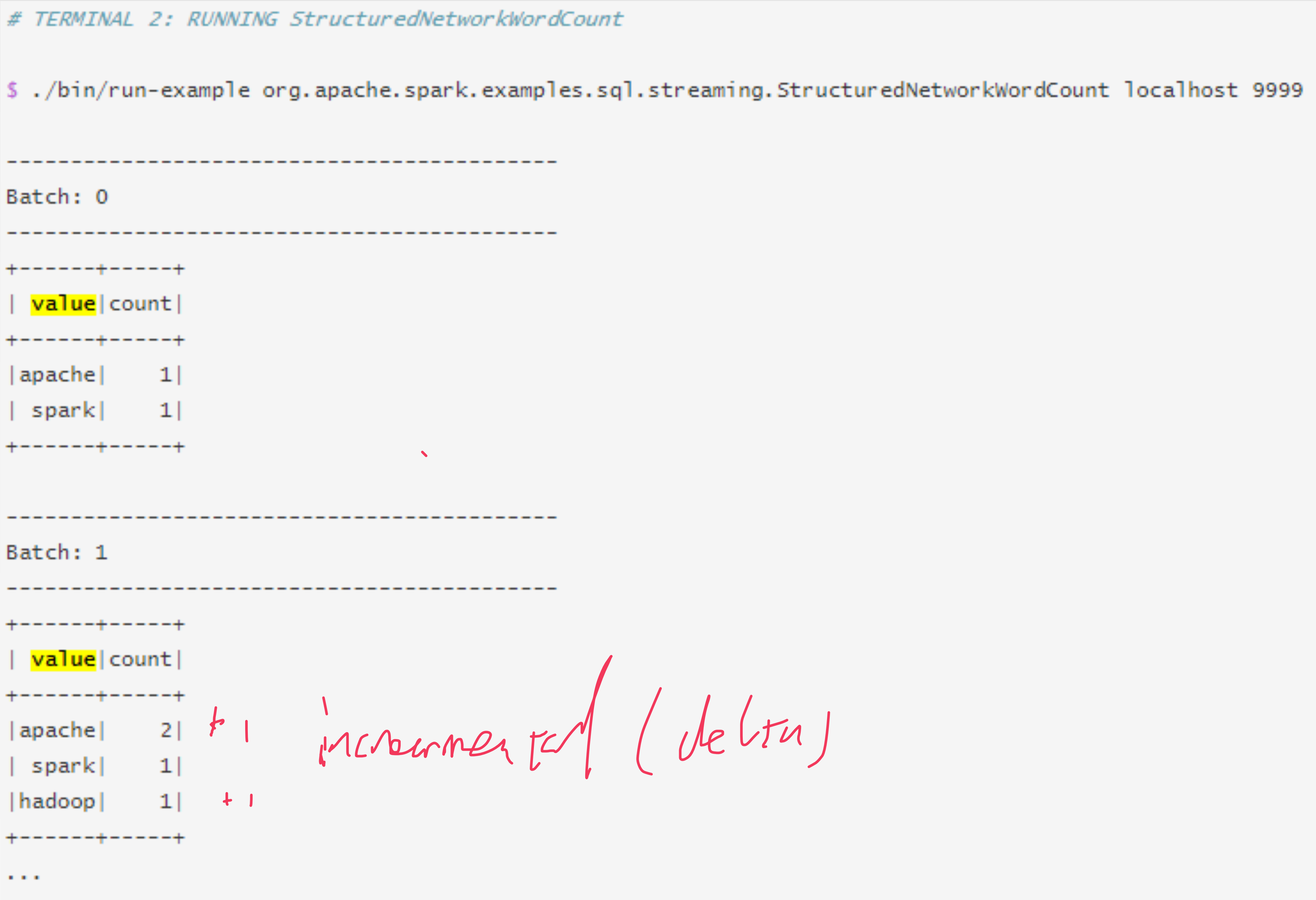
- Use StreamingContext, rather then SparkContext

```
import org.apache.spark._
import org.apache.spark.streaming._

object StructuredNetworkWordCount {
        val spark = SparkSession.builder
                .appName("StructuredNetworkWordCount").getOrCreate()
        import spark.implicits._

        // Create DataFrame representing the stream of input lines from connection to localhost:9999
        val lines = spark.readStream.format("socket").option("host",
        "localhost").option("port", 9999).load()

        // Split the lines into words
        val words = lines.as[String].flatMap(_.split(" "))

        // Generate running word count
        val wordCounts = words.groupBy("value").count()

        // Start running the query that prints the running counts to the console
        val query = wordCounts.writeStream.outputMode("complete")
                .format("console").start()

        query.awaitTermination()
}
```

An Example: Streaming WordCount



```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999
apache spark
apache hadoop
```



```
# TERMINAL 2: RUNNING StructuredNetworkWordCount

$ ./bin/run-example org.apache.spark.examples.sql.streaming.StructuredNetworkWordCount localhost 9999

-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    1|
| spark|    1|
+------+-----+


-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    2|
| spark|    1|
|hadoop|    1|
+------+-----+
...
```

+1 incremental (delta)

+1

References

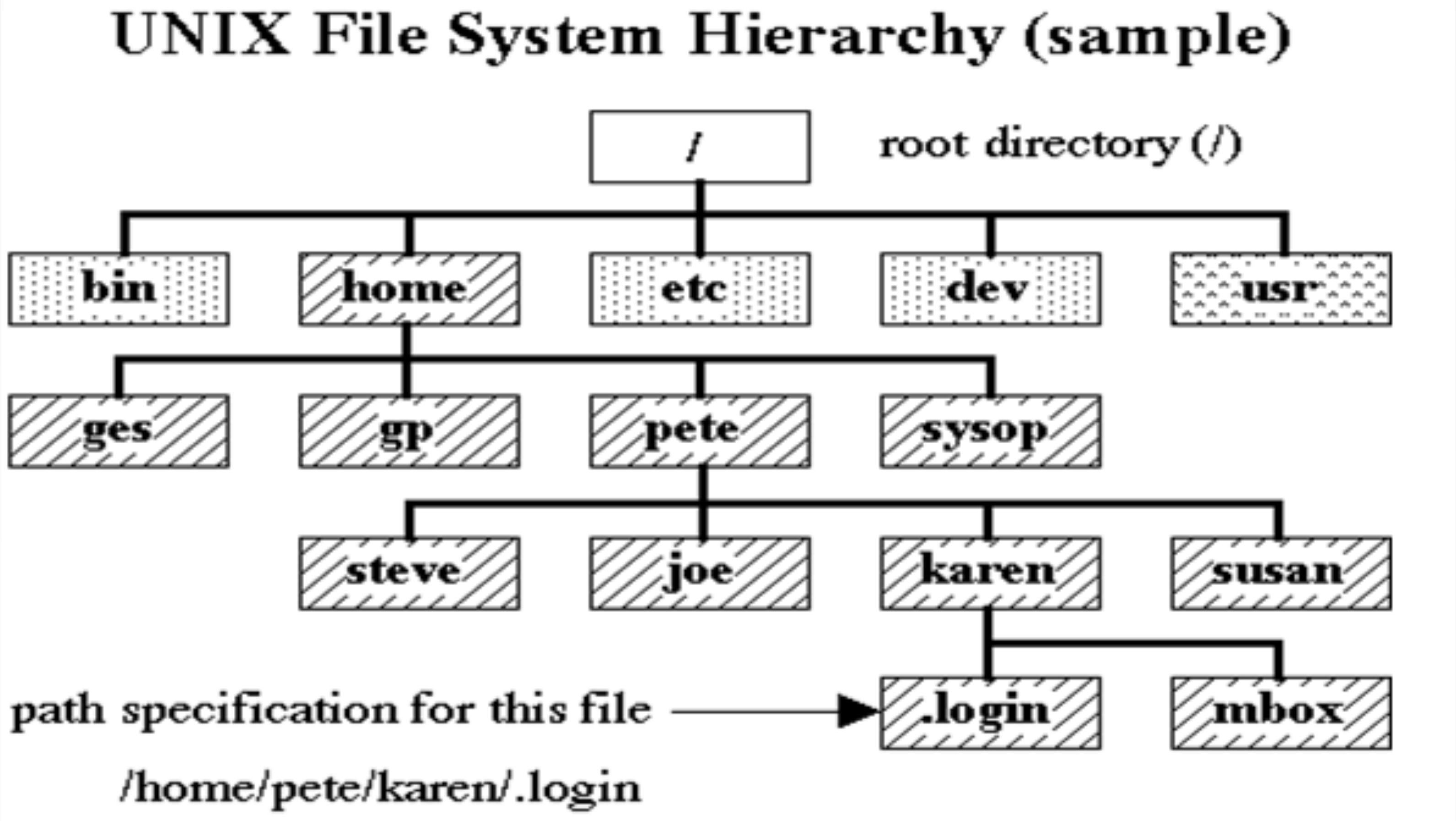
- http://spark.apache.org/docs/latest/index.html
- Spark SQL guide: http://spark.apache.org/docs/latest/sql-programming-guide.html
- Spark Streaming guide: http://spark.apache.org/docs/latest/streaming-programming-guide.html
- Spark Structured Streaming: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
- Learning Spark. O'Reilly Media.

End of Chapter 8

HDFS Introduction

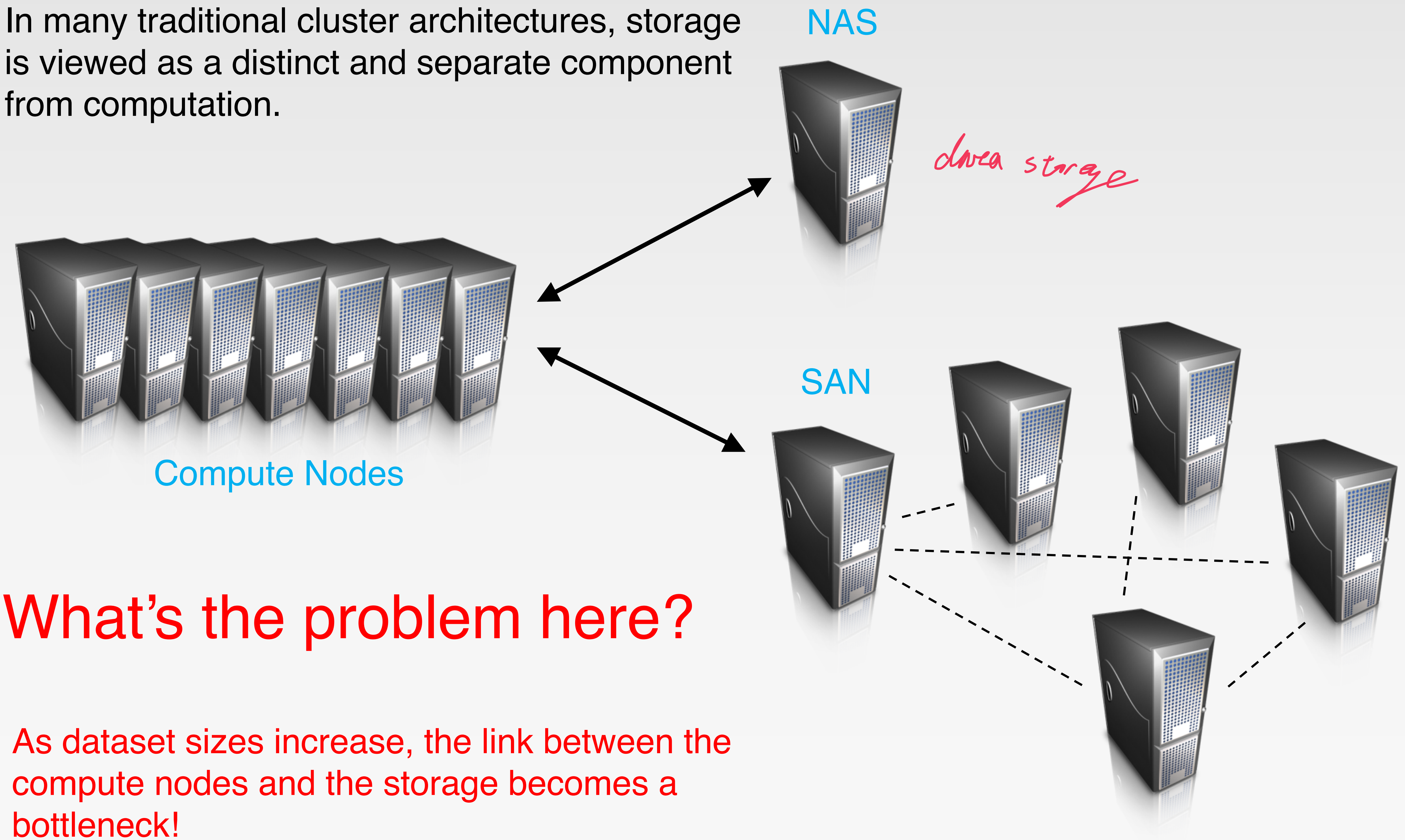
File System

- A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk.



How to Move Data to Workers?

In many traditional cluster architectures, storage is viewed as a distinct and separate component from computation.



What’s the problem here?

As dataset sizes increase, the link between the compute nodes and the storage becomes a bottleneck!

Distributed File System

- Don’t move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow (low-latency), but disk throughput is reasonable (high throughput)
- A distributed file system is the answer
  - A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

Latency and Throughput

- ***Latency*** is the time required to perform some action or to produce some result.
  - Measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.
  - I/O latency: the time that it takes to complete a single I/O.

- ***Throughput*** is the number of such actions executed or results produced per unit of time.
  - Measured in units of whatever is being produced (e.g., data) per unit of time.
  - Disk throughput: the maximum rate of sequential data transfer, measured by Mb/sec etc.

as much as possible

time

Assumptions and Goals of HDFS

- Very large datasets
  - 10K nodes, 100 million files, 10PB
- Streaming data access
  - Designed more for batch processing rather than interactive use by users
  - more data
    The emphasis is on high throughput of data access rather than low latency of data access.
    real time
- Simple coherency model
  - Built around the idea that the most efficient data processing pattern is a write-once read-many-times pattern
  - A file once created, written, and closed need not be changed except for appends and truncates
- "Moving computation is cheaper than moving data" ?
  - Data locations exposed so that computations can move to where data resides

Assumptions and Goals of HDFS (Cont')

- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Hardware failure is normal rather than exception. Detect failures and recover from them
- Portability across heterogeneous hardware and software platforms
  - designed to be easily portable from one platform to another

- HDFS is not suited for:
  - Low-latency data access (HBase is a better option)
  - Lots of small files (NameNodes hold metadata in memory)

HDFS Features

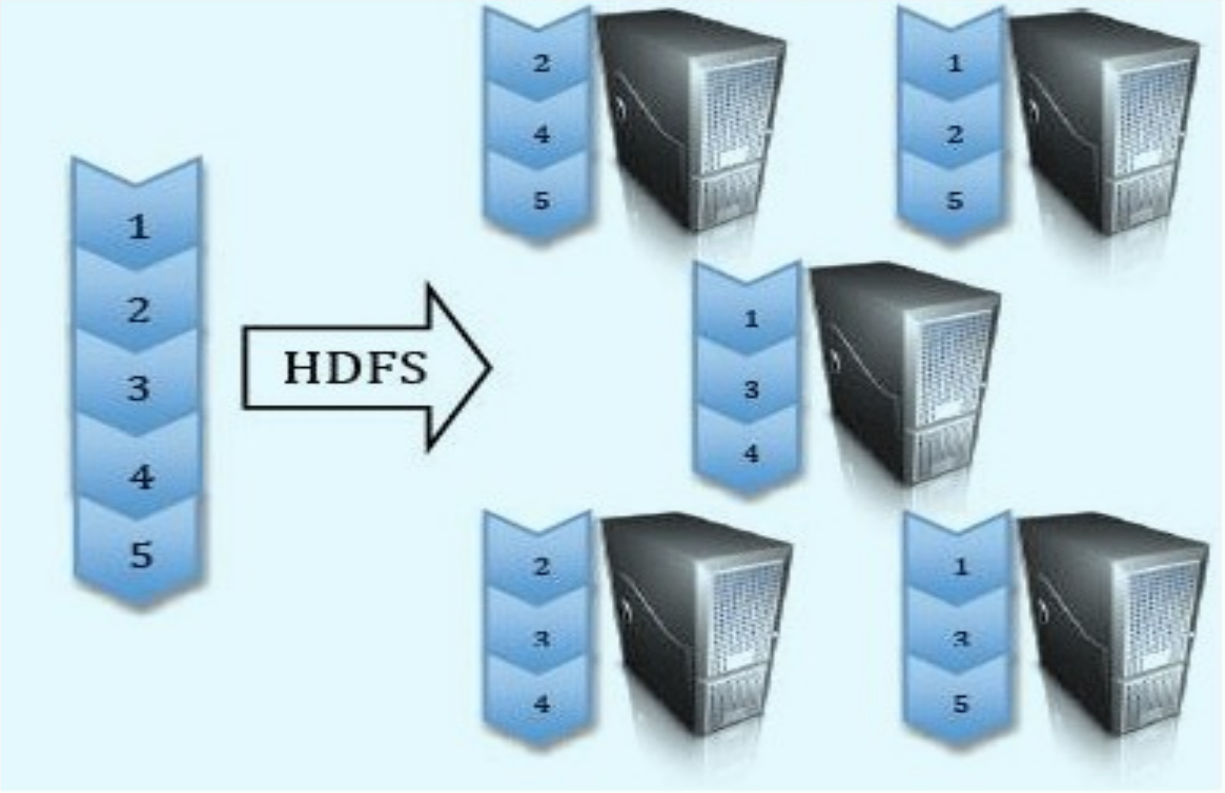
- The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.

- Basic Features:
  - Suitable for applications with large data sets
  - Streaming access to file system data

    from start to end (rather then random access)
  - High throughput
  - Can be built out of commodity hardware
  - Highly fault-tolerant

HDFS Architecture

- HDFS is a block-structured file system: Files broken into blocks of 64MB or 128MB
- A file can be made of several blocks, and they are stored across a cluster of one or more machines with data storage capacity.
- Each block of a file is 3 replicated across a number of machines, To prevent loss of data.

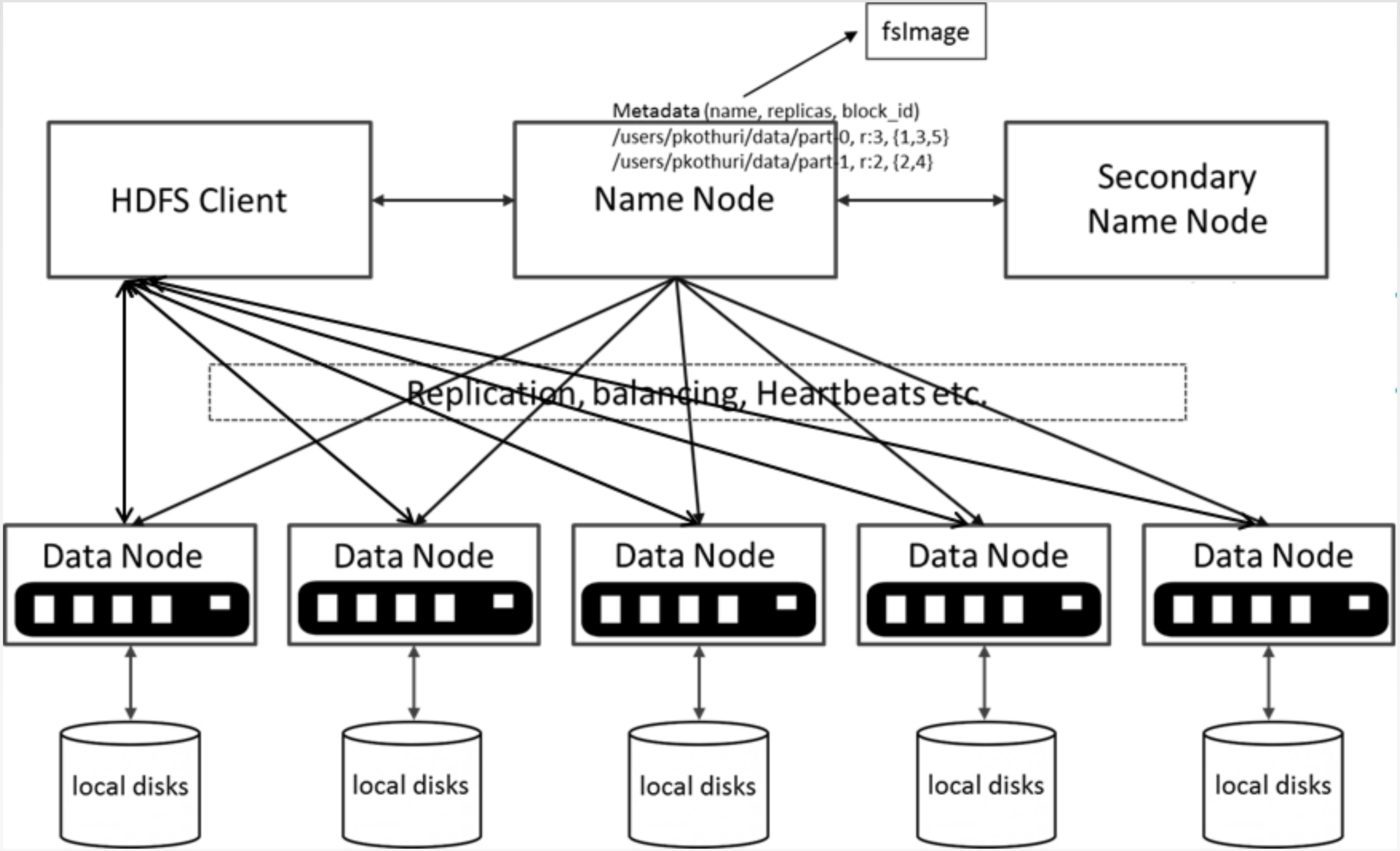


HDFS Architecture

- HDFS has a master/slave architecture.

- There are two types (and a half) of machines in a HDFS cluster
  - NameNode: the heart of an HDFS filesystem, it maintains and manages the file system metadata. E.g., what blocks make up a file, and on which datanodes those blocks are stored.
    - Only one in an HDFS cluster
  - DataNode: where HDFS stores the actual data. Serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode
    - A number of DataNodes usually one per node in a cluster.
    - A file is split into one or more blocks and set of blocks are stored in DataNodes.
  - Secondary NameNode: **NOT** a backup of NameNode!!
    - Checkpoint node. Periodic merge of Transaction log
    - Help NameNode start up faster next time

HDFS Architecture


fsImage
Metadata (name, replicas, block_id)
/users/pkothuri/data/part-0, r:3, {1,3,5}
/users/pkothuri/data/part-1, r:2, {2,4}
HDFS Client
Name Node
Secondary Name Node
Replication, balancing, Heartbeats etc.
Data Node
Data Node
Data Node
Data Node
Data Node
local disks
local disks
local disks
local disks
local disks

Functions of a NameNode

- Managing the file system namespace:
  - Maintain the namespace tree operations like opening, closing, and renaming files and directories.
  - Determine the mapping of file blocks to DataNodes (the physical location of file data).
  - Store file metadata.
- Coordinating file operations:
  - Directs clients to DataNodes for reads and writes
  - No data is moved through the NameNode
- Maintaining overall health:
  - Collect block reports and heartbeats from DataNodes
  - Block re-replication and rebalancing
  - Garbage collection

NameNode Metadata

- HDFS keeps the entire namespace in RAM, allowing fast access to the metadata.
  - 4GB of local RAM is sufficient
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log (EditLog)
  - Records file creations, file deletions etc

Functions of DataNodes

- Responsible for serving read and write requests from the file system's clients.
- Perform block creation, deletion, and replication upon instruction from the NameNode.
- Periodically sends a report of all existing blocks to the NameNode (Blockreport)
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

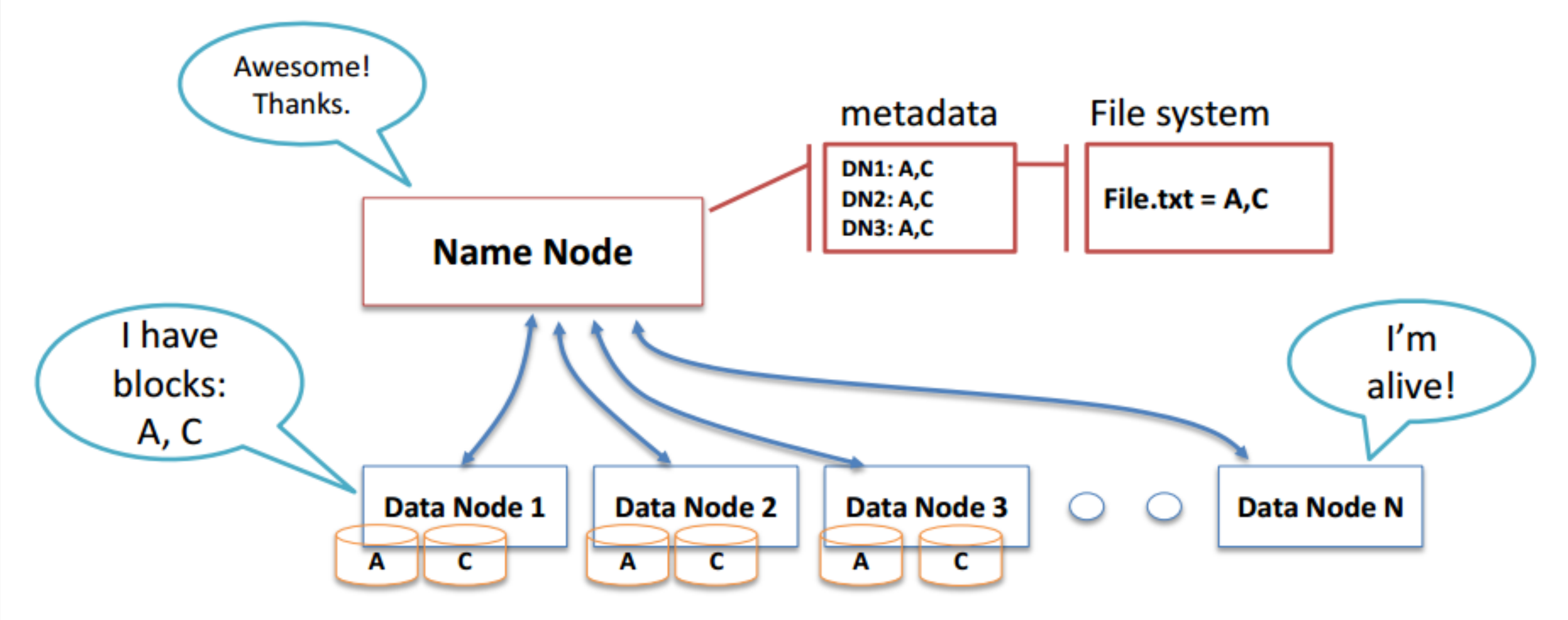
Communication between NameNode and DataDode

- Heartbeats
  - DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available.
    - Once every 3 seconds
  - The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them

- Blockreports
  - A Blockreport contains a list of all blocks on a DataNode

- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster periodically

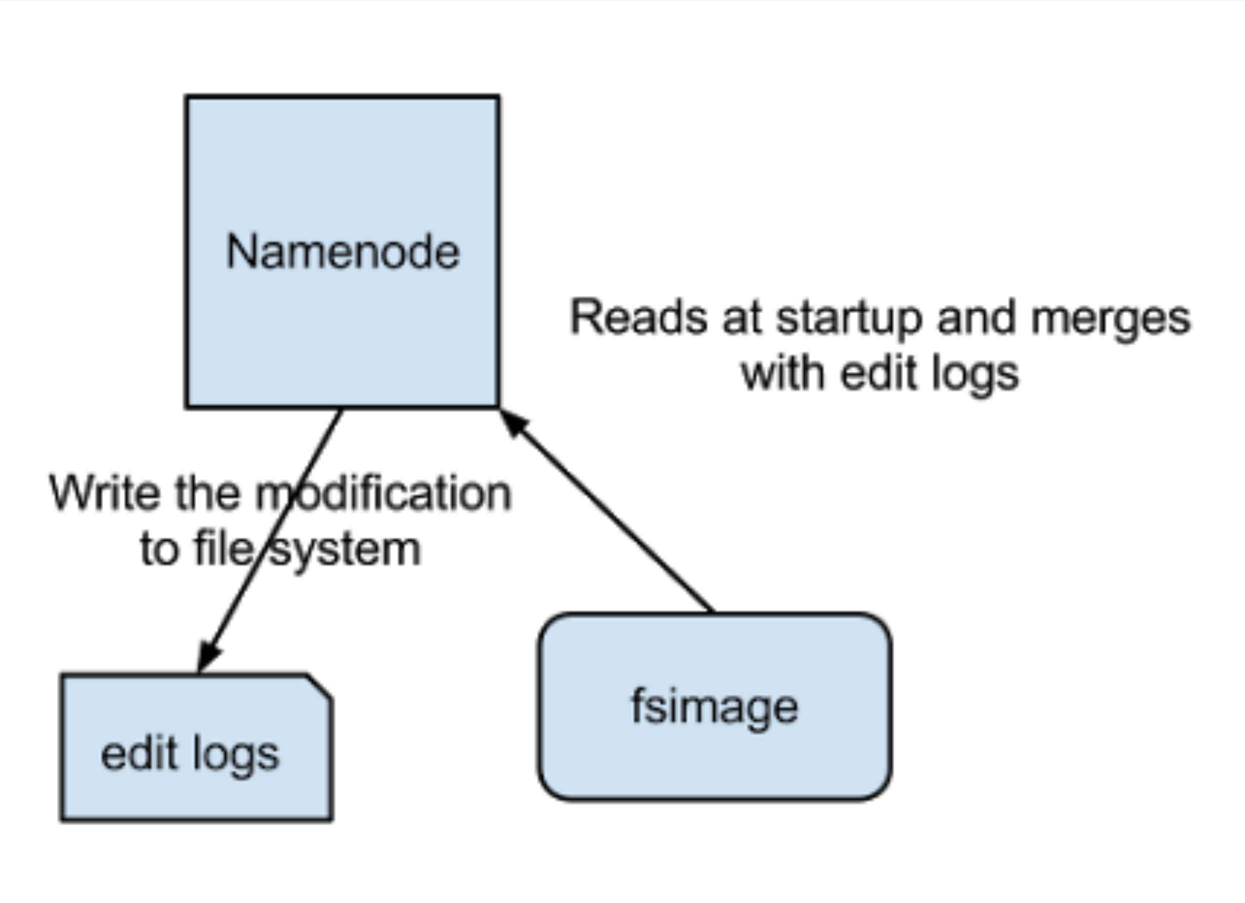
Communication between NameNode and DataDode



- TCP – every 3 seconds a Heartbeat
- Every 10th heartbeat is a Blockreport
- Name Node builds metadata from Blockreports
- If Name Node is down, HDFS is down

Inside NameNode

- FsImage - the snapshot of the filesystem when NameNode started
  - A master copy of the metadata for the file system
- EditLogs - the sequence of changes made to the filesystem after NameNode started


Namenode
Reads at startup and merges with edit logs
Write the modification to file system
edit logs
fsimage

Inside NameNode

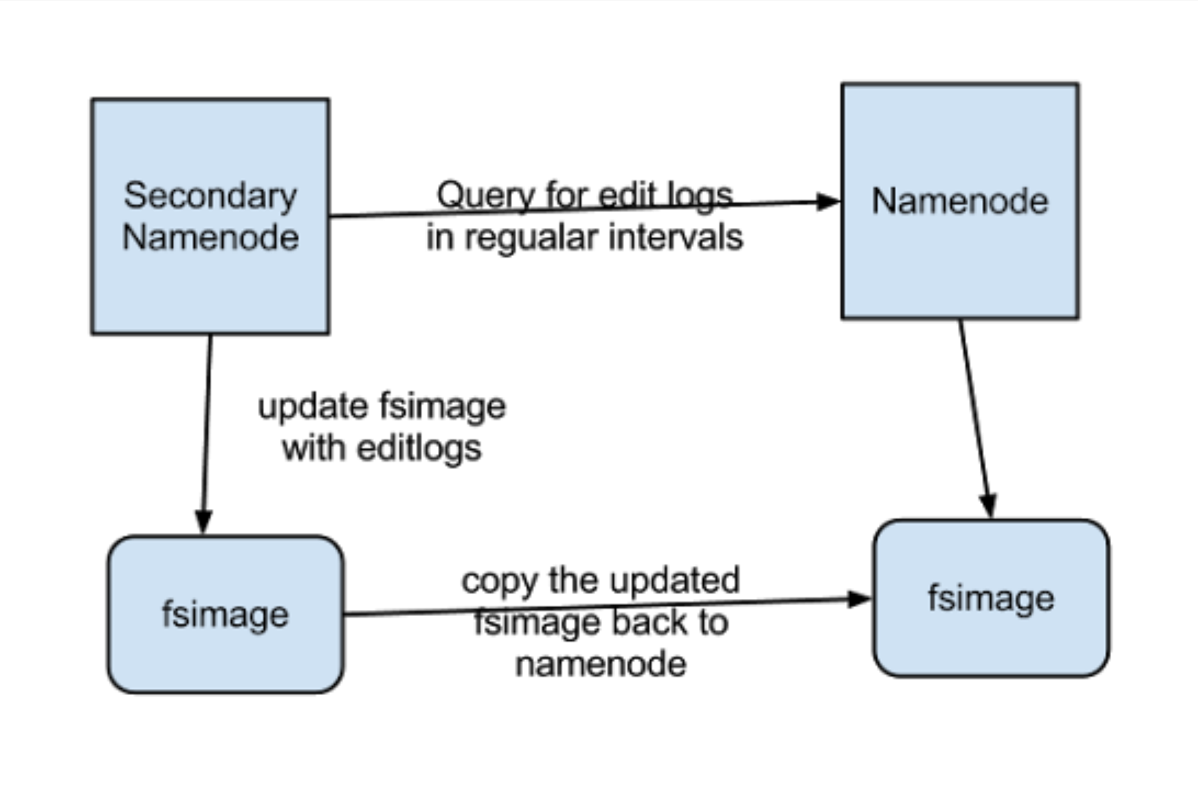
cost lots of time

- Only in the restart of NameNode, EditLogs are applied to FsImage to get the latest snapshot of the file system.
- But NameNode restart are rare in production clusters which means EditLogs can grow very large for the clusters where NameNode runs for a long period of time.
  - EditLog become very large , which will be challenging to manage it
  - NameNode restart takes long time because lot of changes has to be merged
  - In the case of crash, we will lost huge amount of metadata since FsImage is very old

- How to overcome this issue?

Secondary NameNode

- Secondary NameNode helps to overcome the above issues by taking over responsibility of merging EditLogs with FsImage from the NameNode.
  - It gets the EditLogs from the NameNode periodically and applies to FsImage
  - Once it has new FsImage, it copies back to NameNode
  - NameNode will use this FsImage for the next restart, which will reduce the startup time

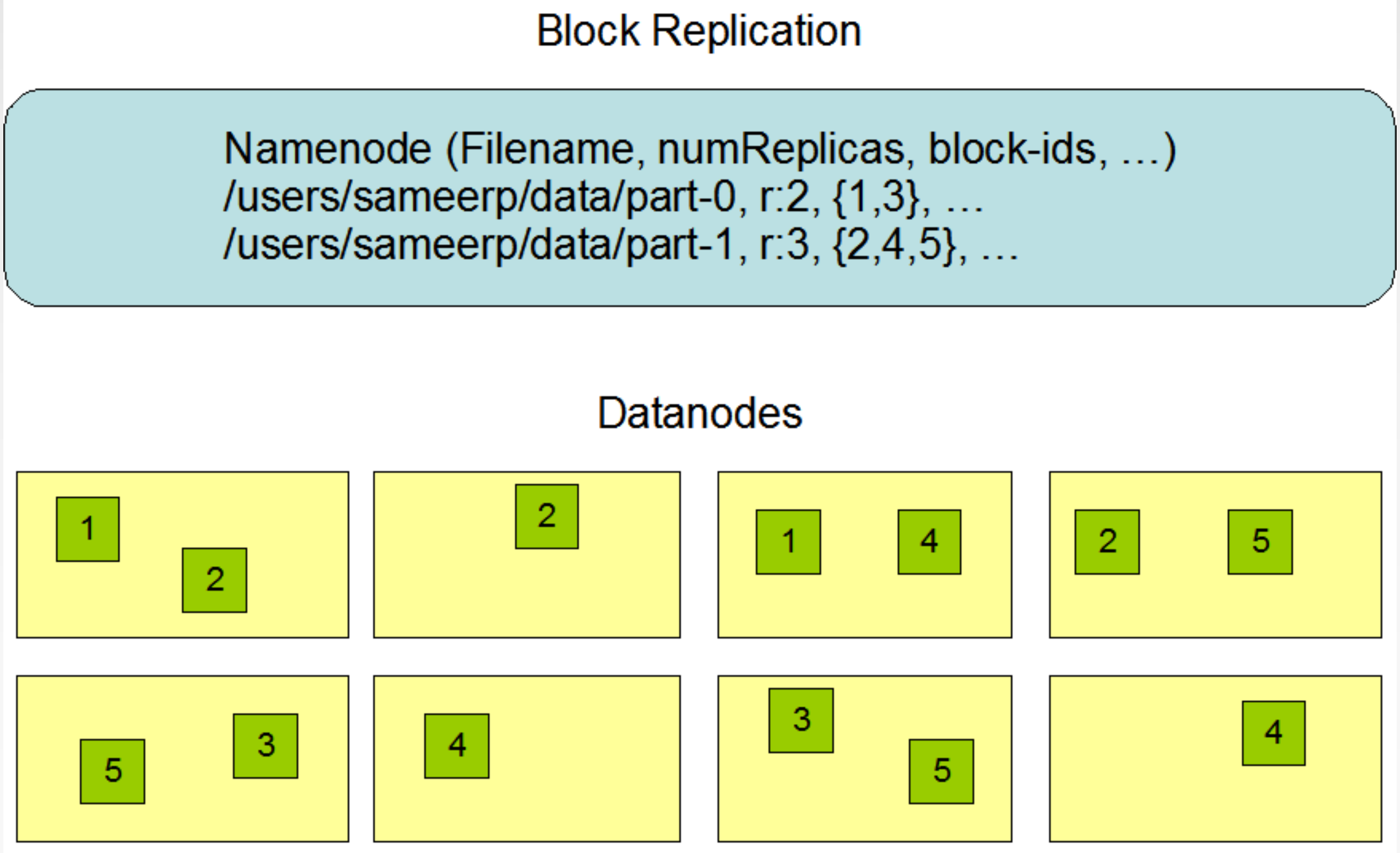

Secondary Namenode
Query for edit logs in regualar intervals
Namenode
update fsimage with editlogs
fsimage
copy the updated fsimage back to namenode
fsimage

File System Namespace

- Hierarchical file system with directories and files
  - /user/comp9313
- Create, remove, move, rename etc.
- NameNode maintains the file system
- Any meta information changes to the file system recorded by the NameNode (EditLog).
- An application can specify the number of replicas of the file needed: replication factor of the file.

Data Replication

- The NameNode makes all decisions regarding replication of blocks.

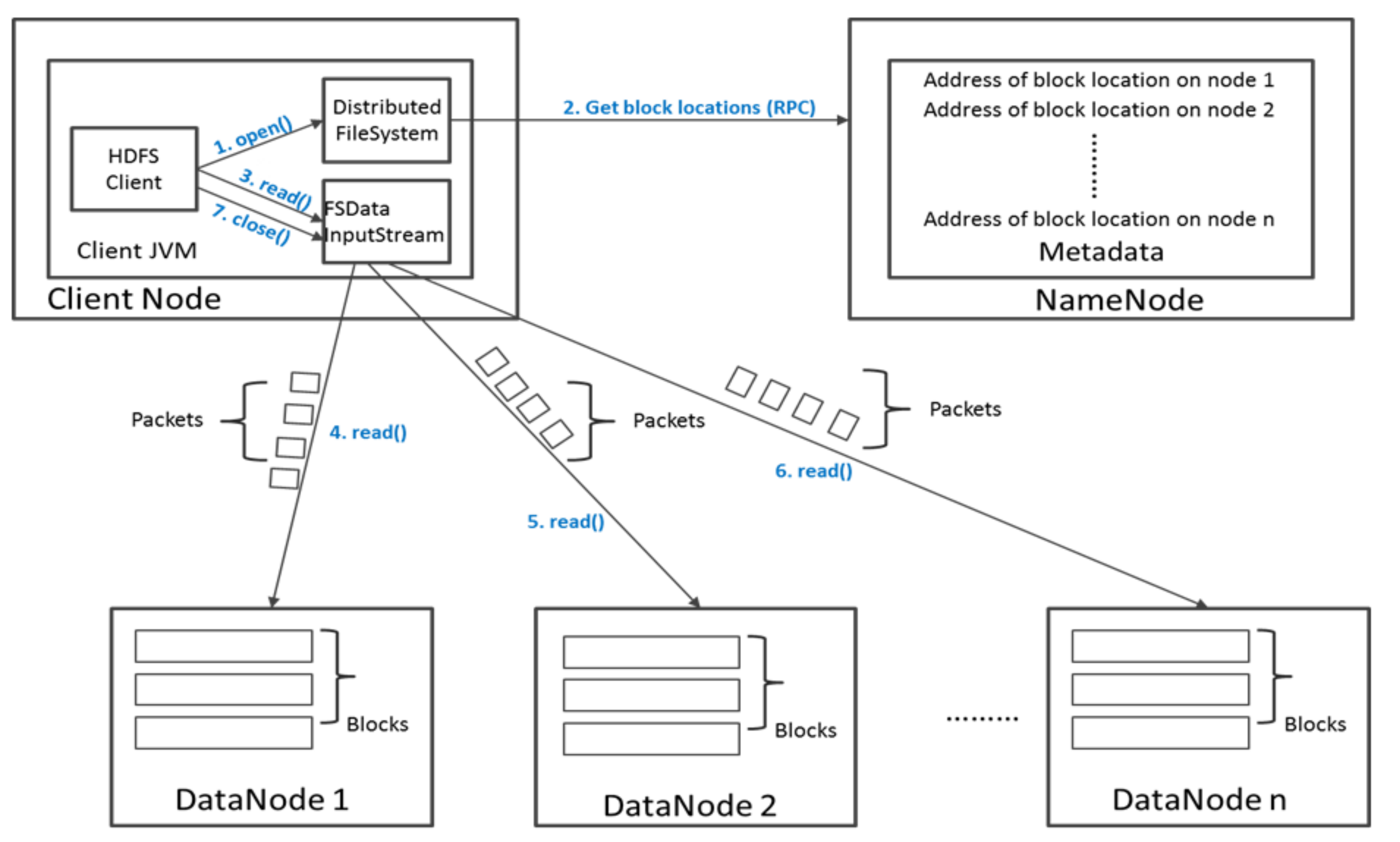


Block Replication

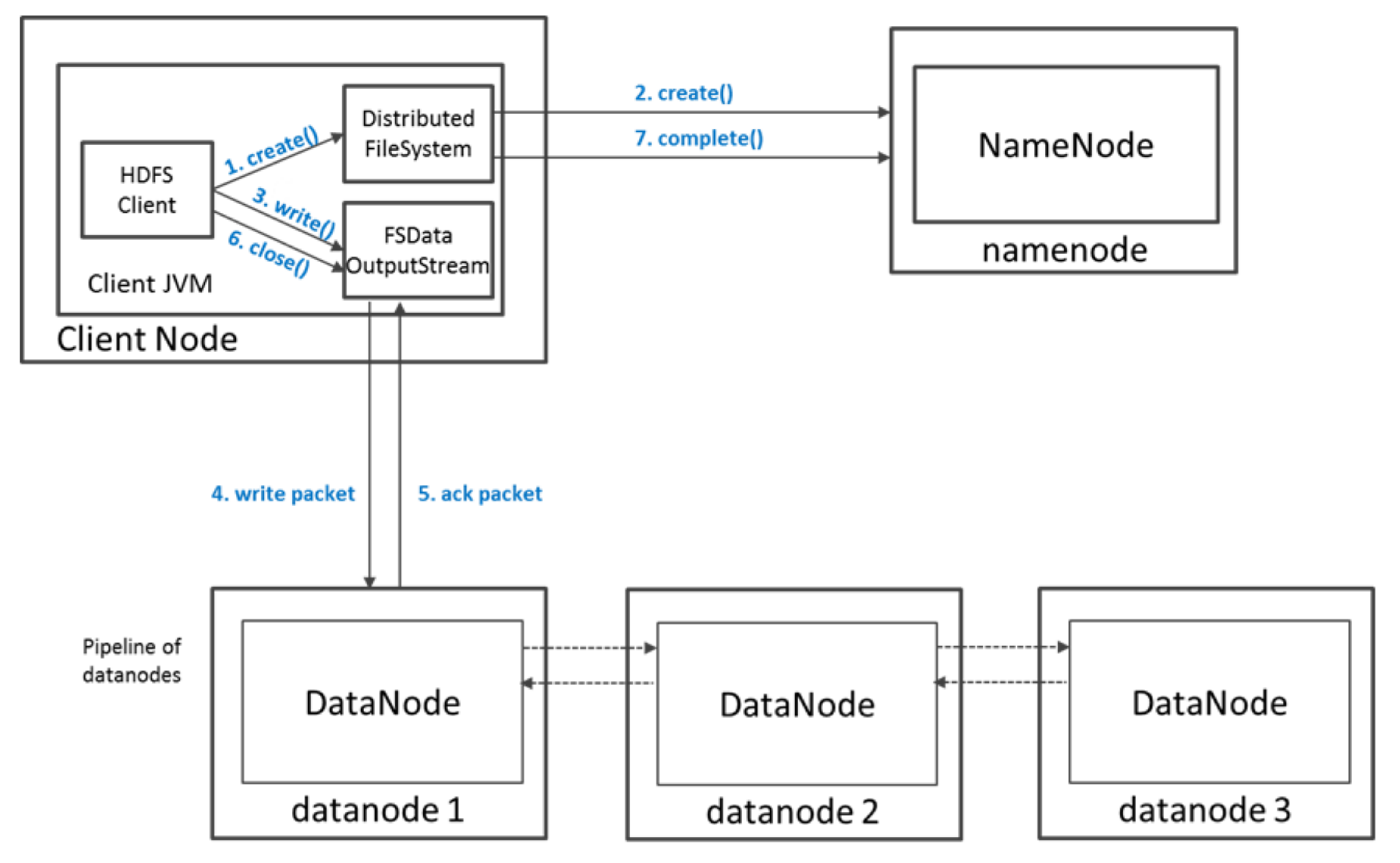
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …

Datanodes

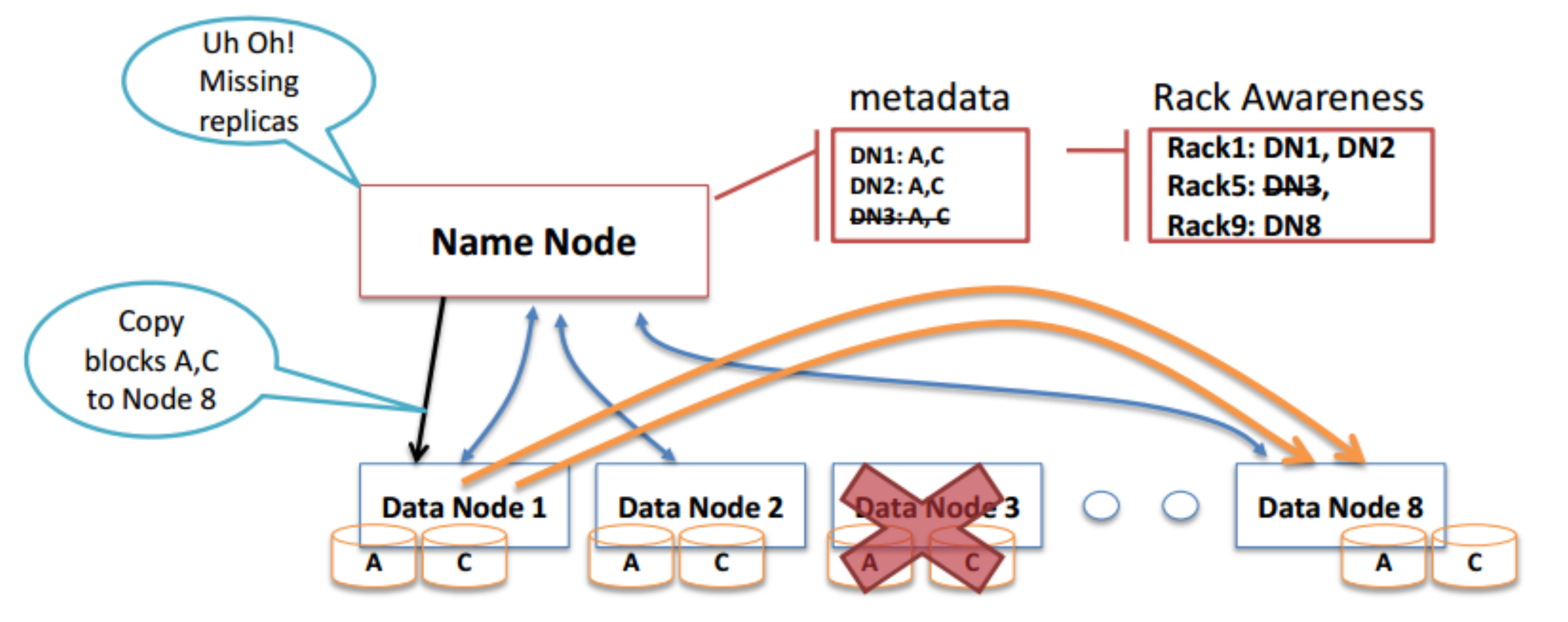
File Read Data Flow in HDFS



File Write Data Flow in HDFS



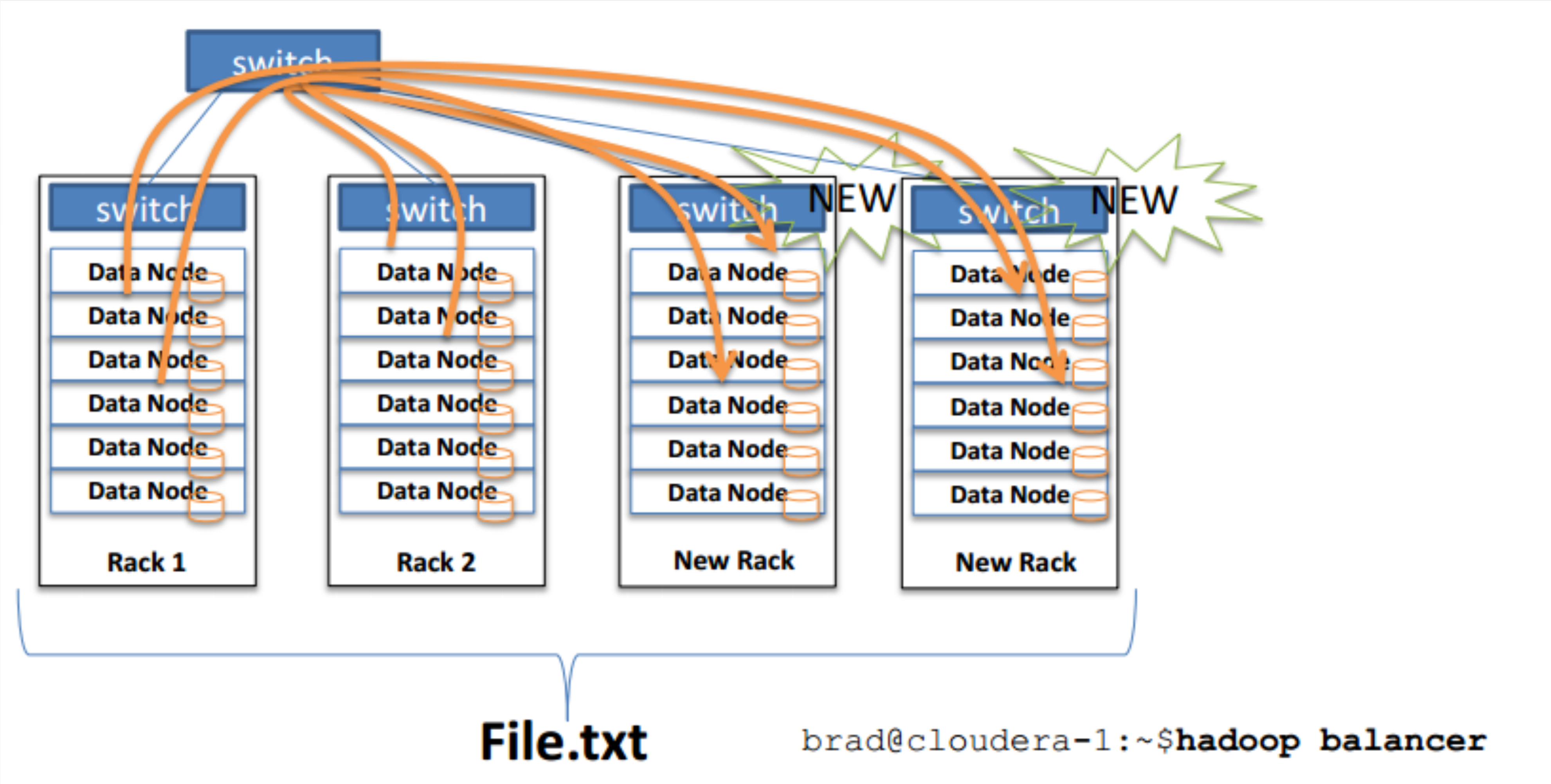
Replication Engine



- NameNode detects DataNode failures
  - Missing Heartbeats signify lost Nodes
  - NameNode consults metadata, finds affected data
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

Cluster Rebalancing

- Goal: % disk full on DataNodes should be similar
  - Usually run when new DataNodes are added
  - Rebalancer is throttled to avoid network congestion
  - Does not interfere with MapReduce or HDFS
  - Command line tool



```
brad@cloudera-1:~$hadoop balancer
```

Fault tolerance

- Failure is the norm rather than exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system’s data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Metadata Disk Failure

- FsImage and EditLog are central data structures of HDFS. A corruption of these files can cause a HDFS instance to be non-functional.
  - A NameNode can be configured to maintain multiple copies of the FsImage and EditLog
  - Multiple copies of the FsImage and EditLog files are updated synchronously

Unique features of HDFS

- HDFS has a bunch of unique features that make it ideal for distributed systems:
    - Failure tolerant - data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).
    - Scalability - data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes
    - Space - need more disk space? Just add more DataNodes and re-balance
    - Industry standard - Other distributed applications are built on top of HDFS (HBase, Map-Reduce)

- HDFS is designed to process large data sets with write-once-read-many semantics, it is not for low latency access