Designing Distributed Geospatial Data-Intensive Applications

Ph.D. Course, 2022

Instructors:

Prof. Luca Foschini, Associate Professor &

Dr. Isam Mashhour Al Jawarneh, Postdoctoral Research Fellow

{isam.aljawarneh3, Luca.foschini}@unibo.it

Department of Computer Science and Engineering (DISI), Università di Bologna

Part 1
section 2

Introduction

19th July 2022

Spark

- **It is not a modified version of Hadoop** but a **separate**, **fast, MapReduce-like engine:**
- **New optimized version of Hadoop**
  - **In-memory** data storage for very fast iterative queries
  - **General execution** of graphs and powerful optimizations
  - Up to **40 times faster than Hadoop**

- Compatible with Hadoop storage APIs
  - Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc.

Why Spark?

- MapReduce greatly simplified big data analysis
- But when it becomes popular, users wanted more:
- More **complex**, **multi-stage applications** (e.g., iterative graph algorithms and machine learning)
- **More interactive ad-hoc queries**
- Both multi-stage and interactive apps require faster **data sharing** across parallel jobs
- Use of sharing and **caching of data** with the goal **of speed**

Spark Basics

- **Various types of data processing computations available in one single tool**
  - **Batch/streaming** analysis, interactive queries and iterative algorithms
  - Previously, these would require **several distinct and independent tools**

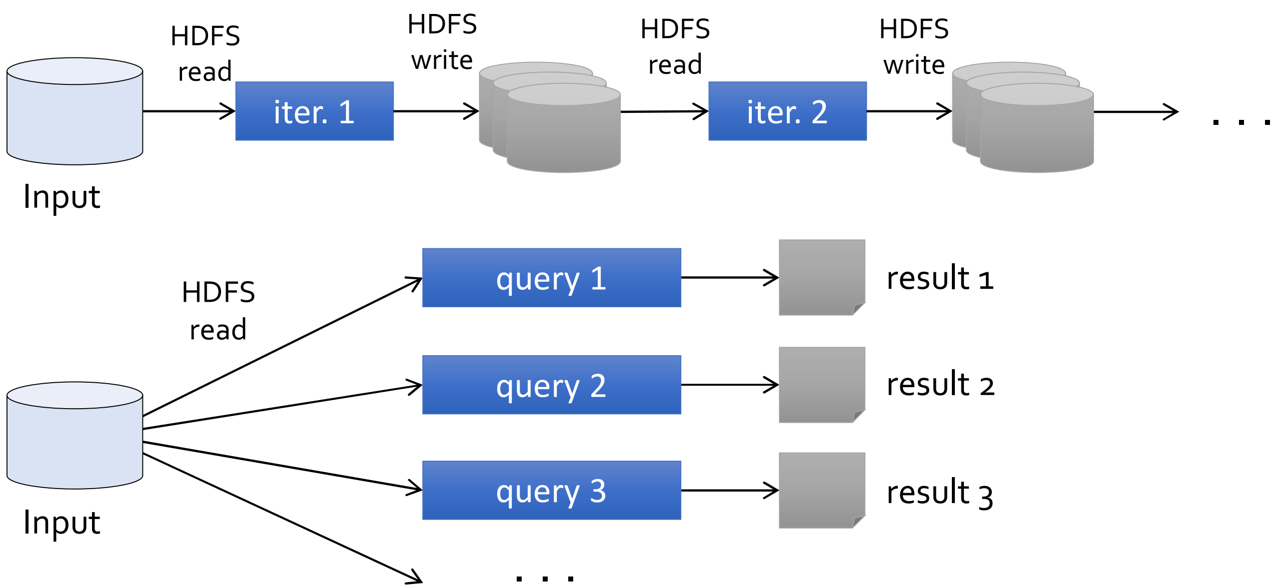
- Supports **several storage options and streaming inputs for parsing**
- **APIs available in Java, Scala, Python, R, …**
  - Also R language supported, for data scientists with moderate programming experience

Spark at a glance

- **Leverages on in-memory data processing**:
- Removes the MapReduce overhead of writing **intermediate results on disk**
- Fault-tolerance is still achieved through the concept of **lineage**

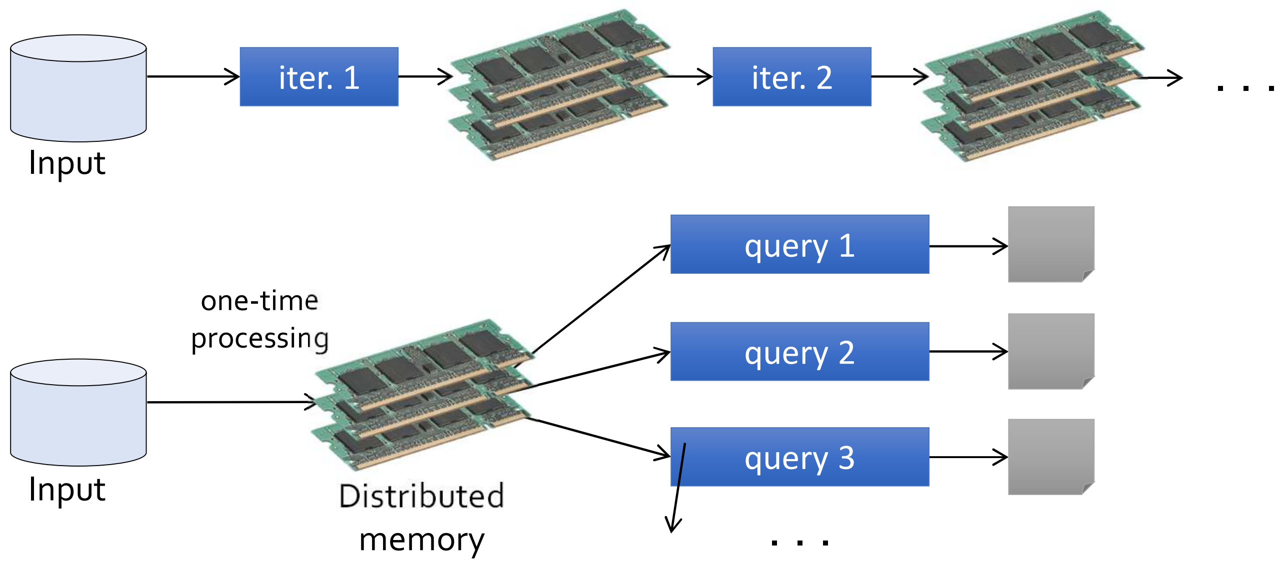
- **Master/Worker cluster architecture**
- Easily deployable in most environments, including existing Hadoop clusters
- Widely configurable **for performance optimization**, both in terms of resource usage and application behavior

Data Sharing in Hadoop



Slow due to replication, **serialization**, and **disk IO**

Data sharing in Spark



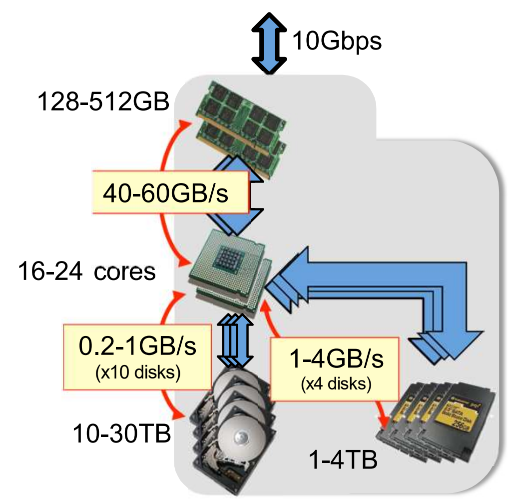
10-100× faster than network and disk

How does Spark gain efficiency?

- Exploit Memory - Network & Disk I/O are the bottleneck

- Many datasets fit into memory
  - ➢The inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory
  - ➢1TB = 1 billion records @ 1 KB

- Memory density (still) grows with Moore's law
  - ➢RAM/SSD hybrid memories at horizon



High-end datacenter node

Spark programming model

- Programs can be run both
- From **compiled sources**, with proper Spark dependencies, with the *Spark-submit* script
- *Interactively* from **Spark Shell**, a console available for Scala and Python languages

- Key idea:
- ***Resilient Distributed Datasets (RDDs)*** **kept in memory**

- **Distributed, immutable collections of objects**
- **Can be cached in memory across cluster nodes**

RDD Transformation

- In addiction to **being *lazily* evaluated**, all **transformations** are computed again on every **action** requested

Transformation

Action

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Until the third line, no operation is performed

The *reduce()* will then force a read from the text file and the *map()* transformation

Persisting RDDs

- In addiction to **being *lazily* evaluated**, all **transformations** are computed again on every **action** requested

Action Transformation Action

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
println(lineLengths.count())
val totalLength = lineLengths.reduce((a, b) => a + b)
```

This effect is expensive, but can be avoided by using the ***persist()*** method

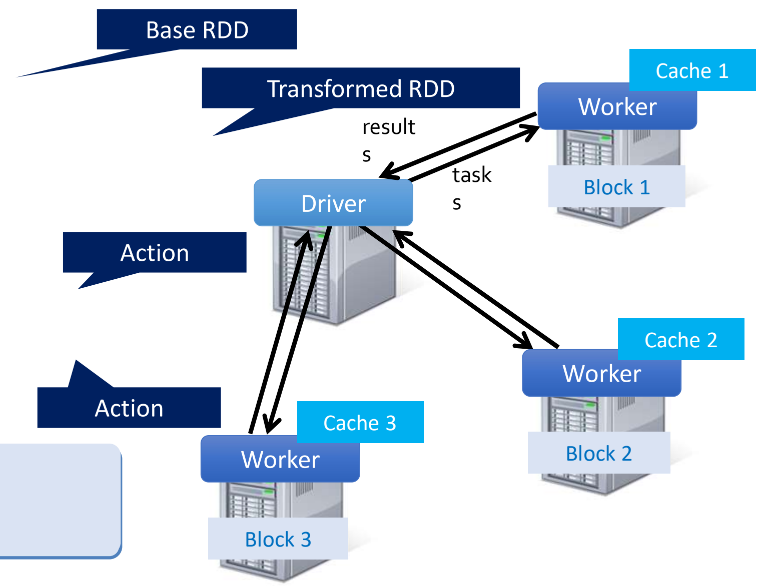
```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
lineLengths.persist()
```

The RDD data read and mapped will then be saved for future actions

Example log Mining

Load error messages from a log into memory, then interactively search for various patterns
→ Spark is conveniently used in Industry 4.0 scenarios!

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```

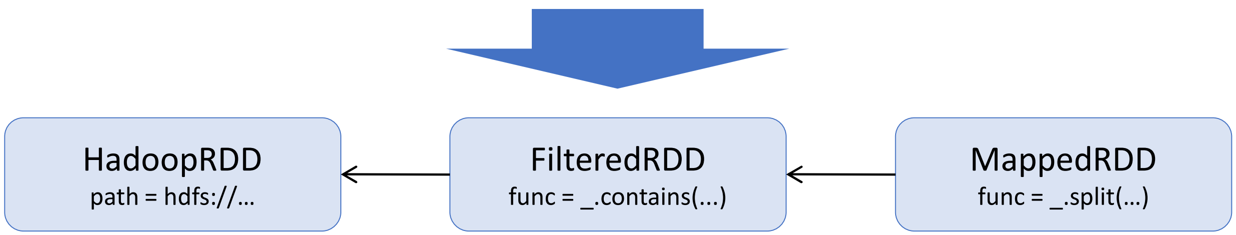


Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Fault Tolerance

RDDs track the series of transformations used to build them (their ***lineage***) to re-compute lost data

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



Example: logistic regression

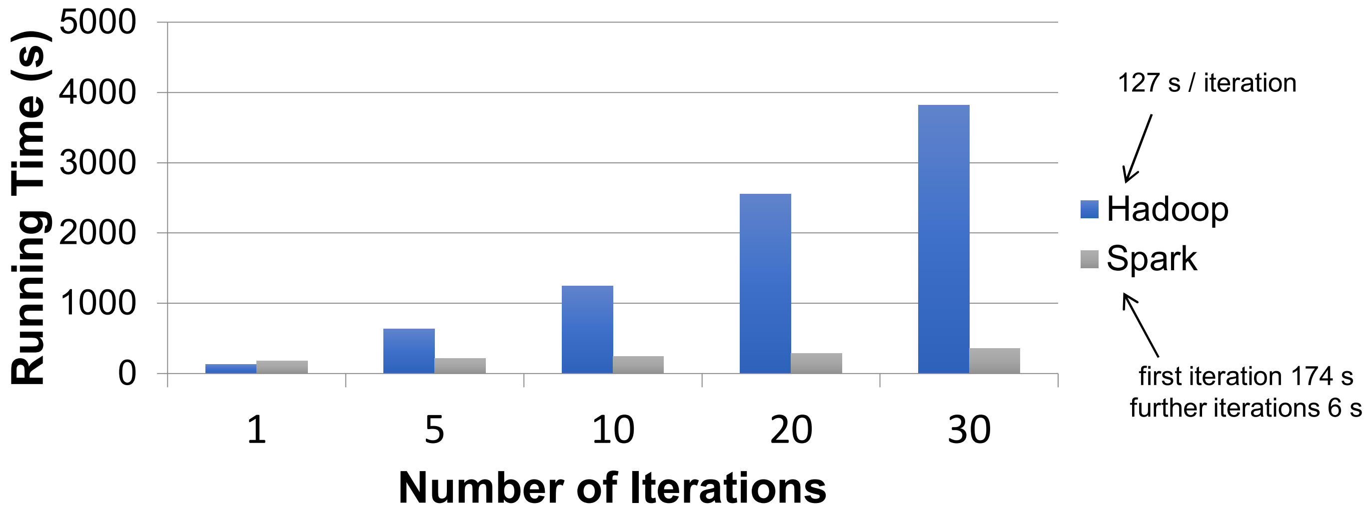
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Load data in memory once

Initial parameter vector

Repeated MapReduce steps
to do gradient descent

Logistic regression performances



Supported operators

- `map`
- `filter`
- `groupBy`
- `sort`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`
- `reduce`
- `count`
- `reduceByKey`
- `groupByKey`
- `first`
- `union`
- `cross`
- `sample`
- `cogroup`
- `take`
- `partitionBy`
- `pipe`
- `save`
- `...`

Supported operators

- `map`
- `filter`
- `groupBy`
- `sort`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`
- `reduce`
- `count`
- `reduceByKey`
- `groupByKey`
- `first`
- `union`
- `cross`
- `sample`
- `cogroup`
- `take`
- `partitionBy`
- `pipe`
- `save`
- `...`

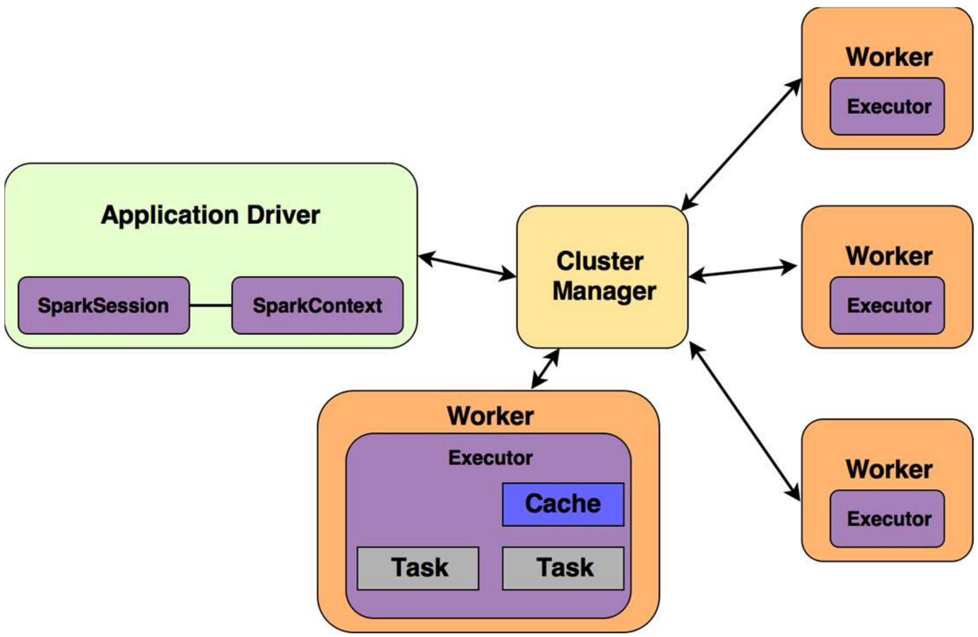
Spark Architecture

- Once submitted, Spark programs create ***directed acyclic graphs (DAGs)*** of all transformations and actions, internally optimized for the execution
- The graph is then split into **stages**, in turn composed by **tasks**, the smallest unit of work

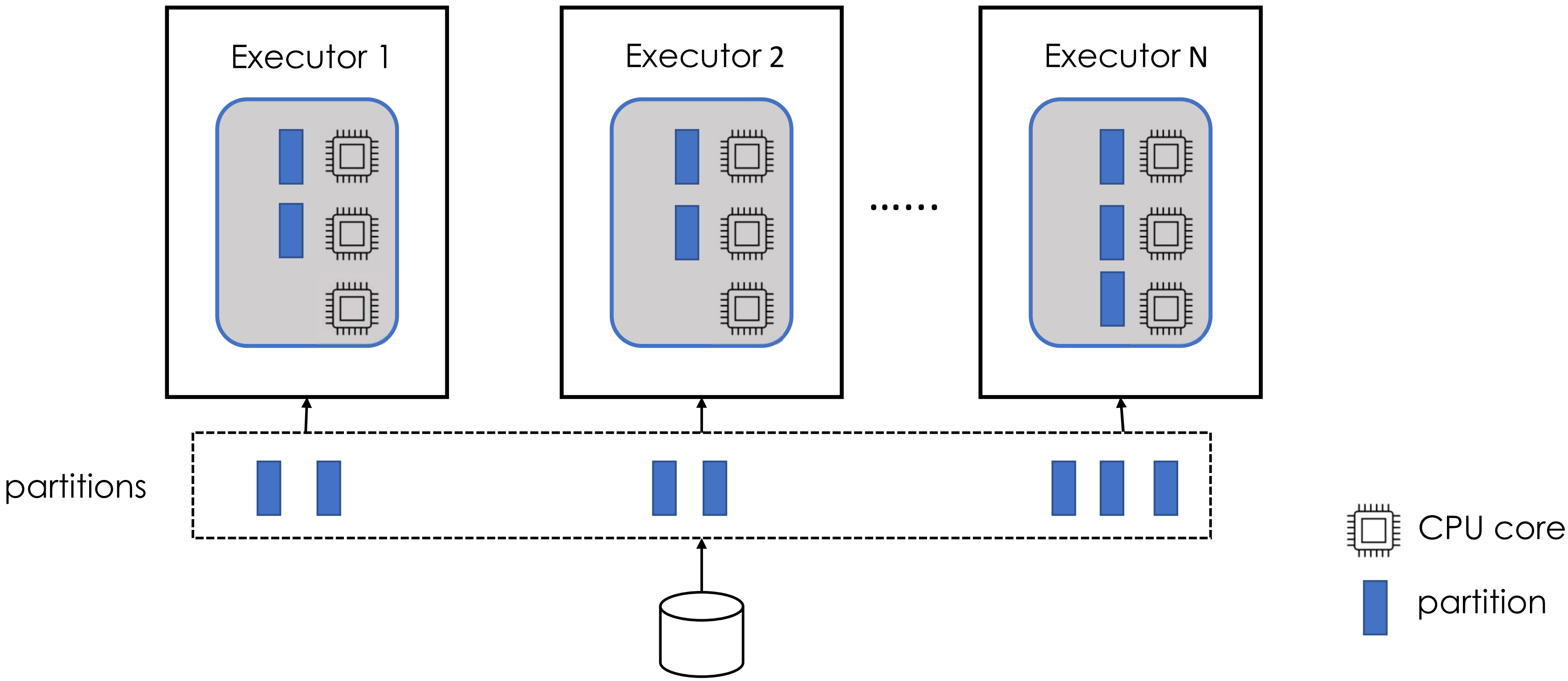
- Thus, Spark is a master/slave system composed by:
- **Driver,** central coordinator node running the *main()* method of the program and dispatching tasks
- **Cluster Master**, node that launches and manages actual executors
- **Executors**, responsible for running tasks

Spark Architecture

- Each executor spawns at least one dedicated **JVM**, to which a certain share of resources is assigned, in terms of:
- **Number of CPU threads**
- **Amount of RAM memory**
- **The number of JVMs and their resources can be customized**



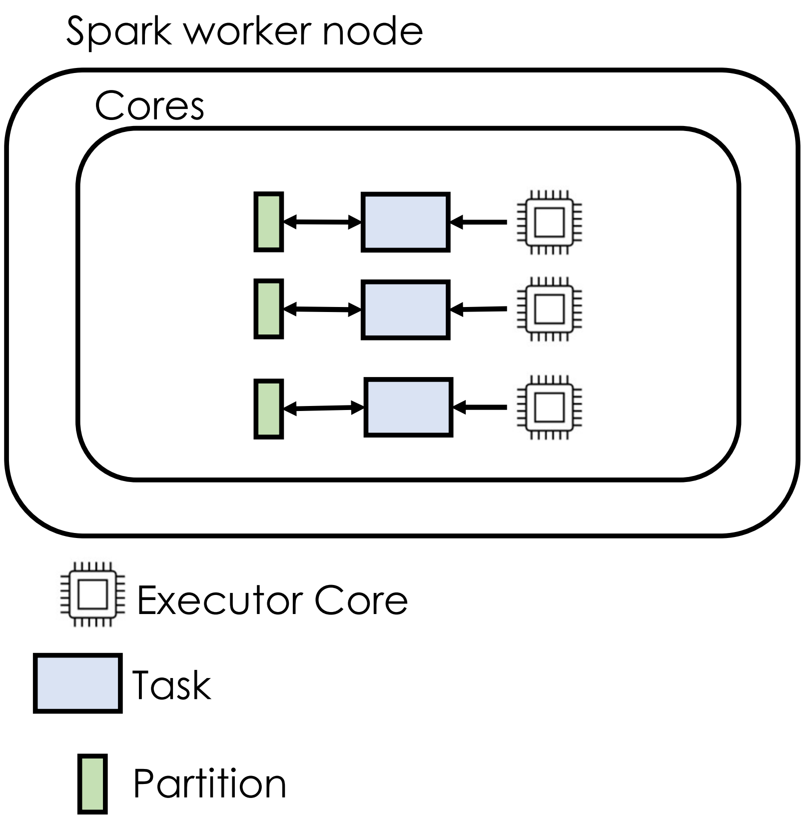
Partitioning in Spark

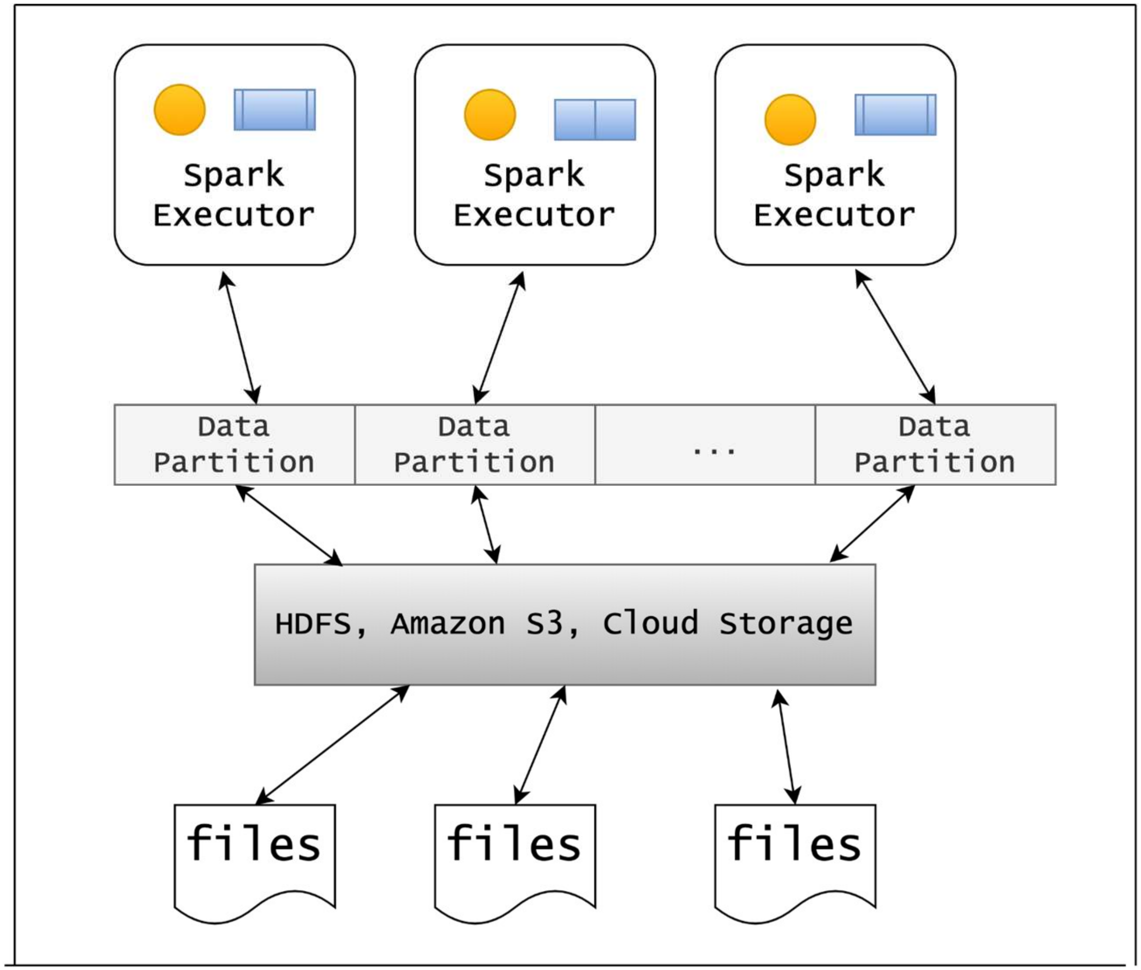


Every **executor core** is assigned a **data partition** to work on, minimizing network bandwidth

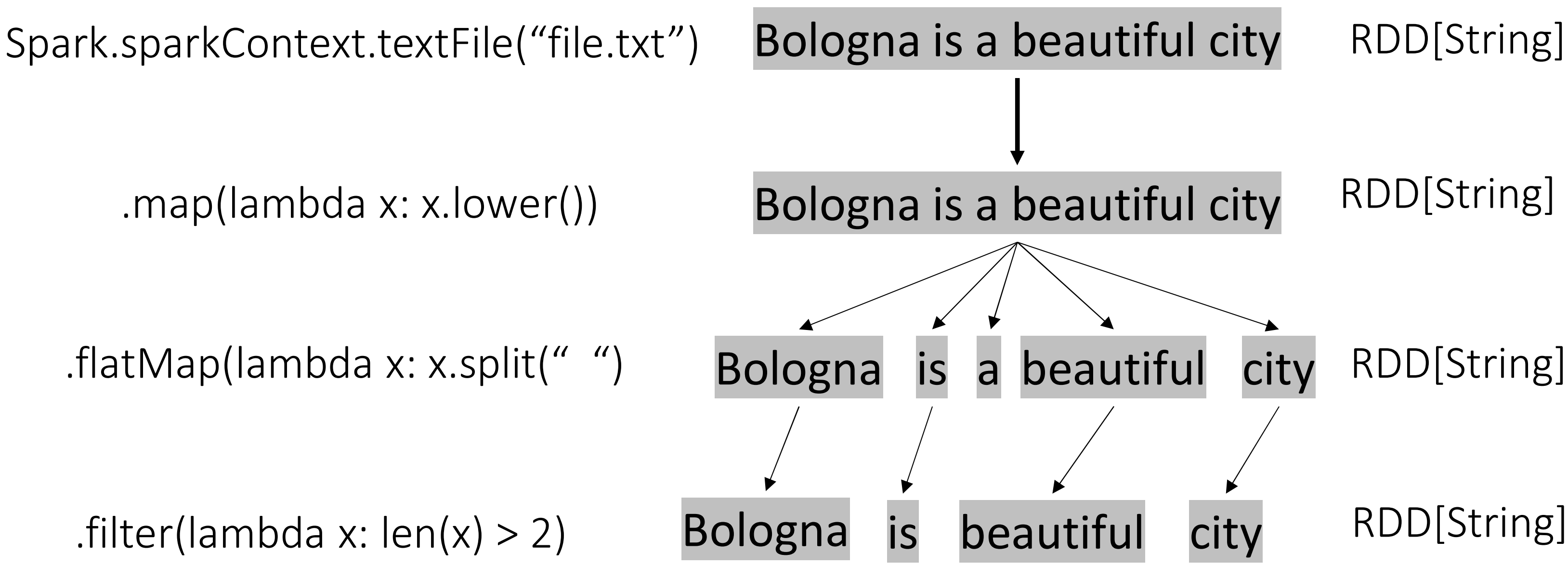
Cores to Tasks to data partitioning relationships on Spark Executors

- Each **task** that is allocated to a Spark **core** works on a single **partition**
  - More **partitions** achieves more **parallelism**



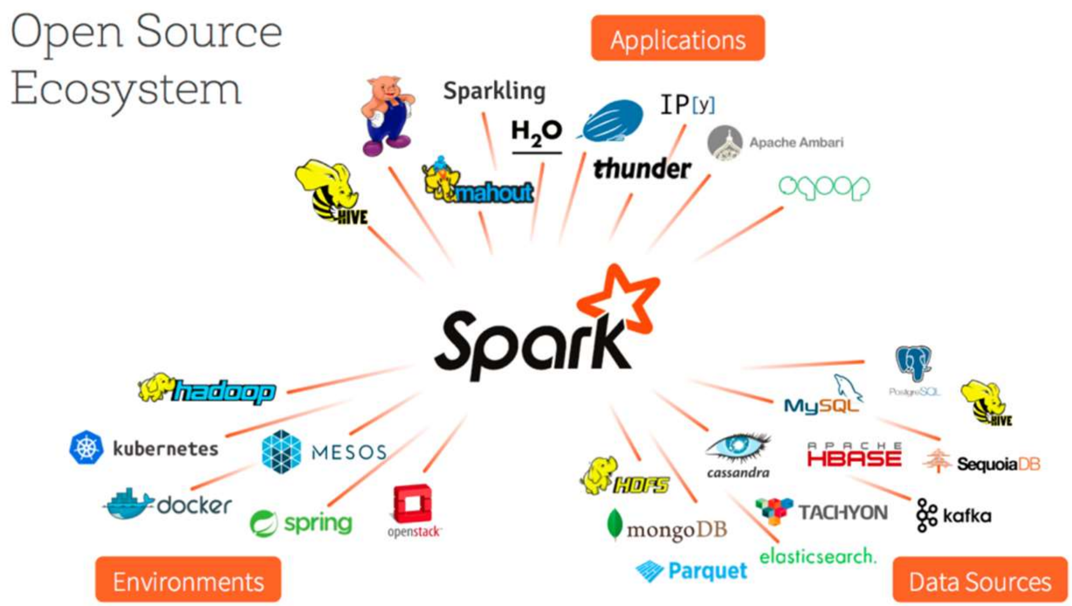

Spark Executor
Spark Executor
Spark Executor
Data Partition
Data Partition
...
Data Partition
HDFS, Amazon S3, Cloud Storage
files
files
files

Spark RDD Transformation

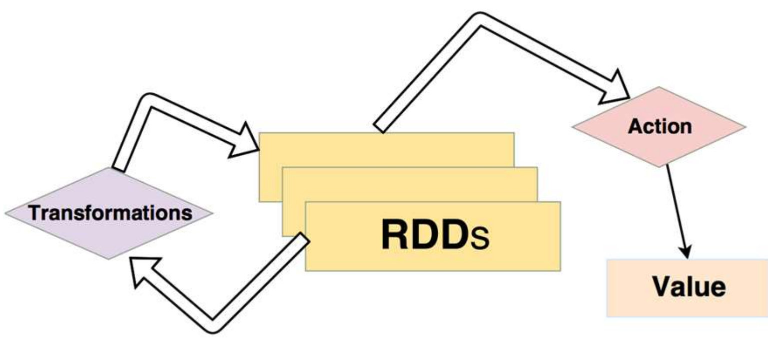


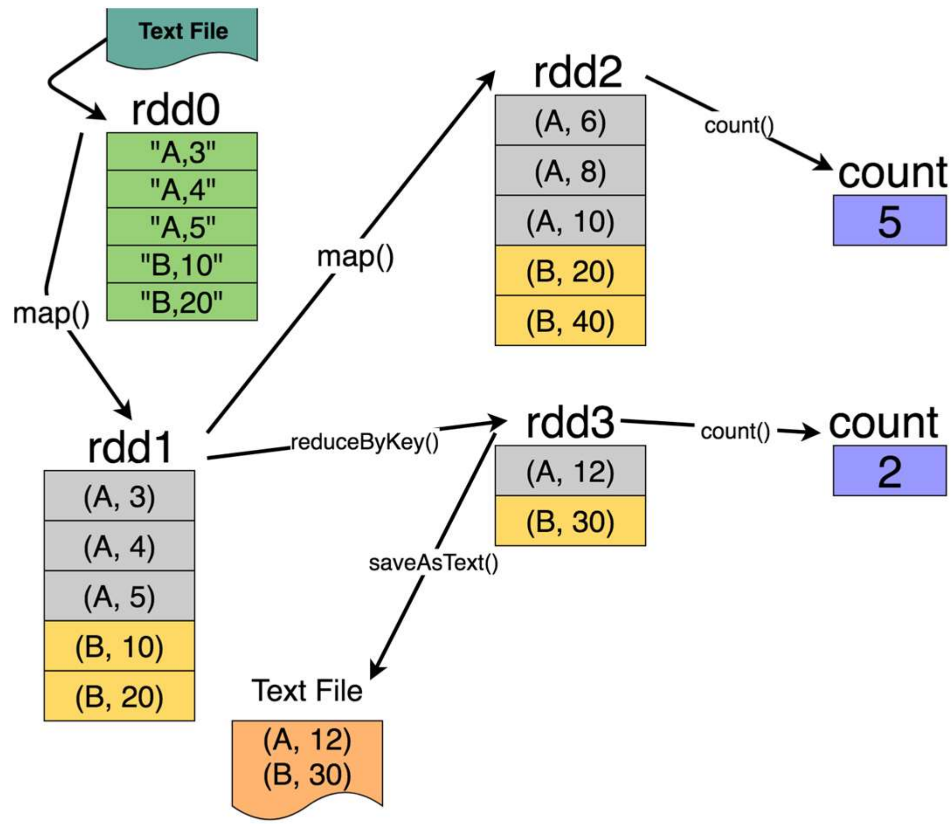
Spark Eco-System (source Databricks)

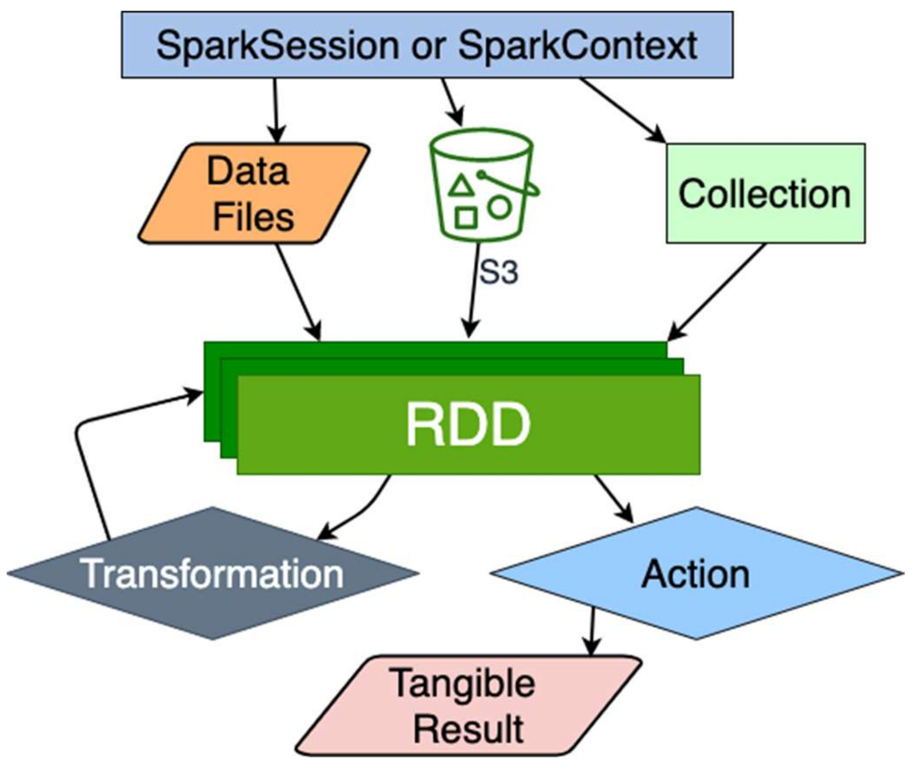
- three main components:
    - **Environments**:
      can run anywhere and integrate well with other environments
    - **Applications**:
      it integrates well with big data platforms and applications
    - **Data Sources**:
      can read/write data from/to many data sources

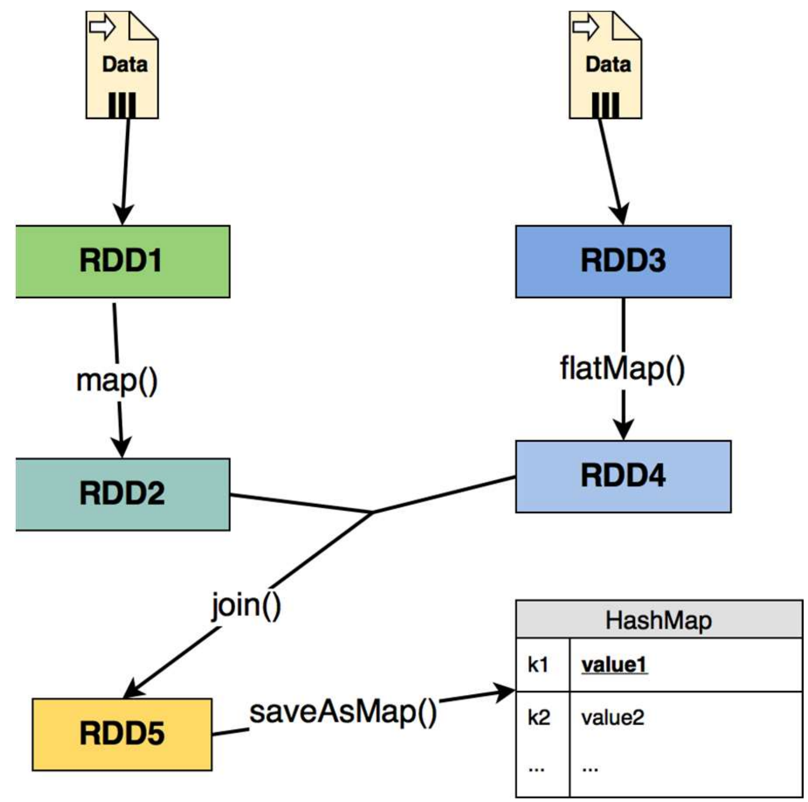


Example operations



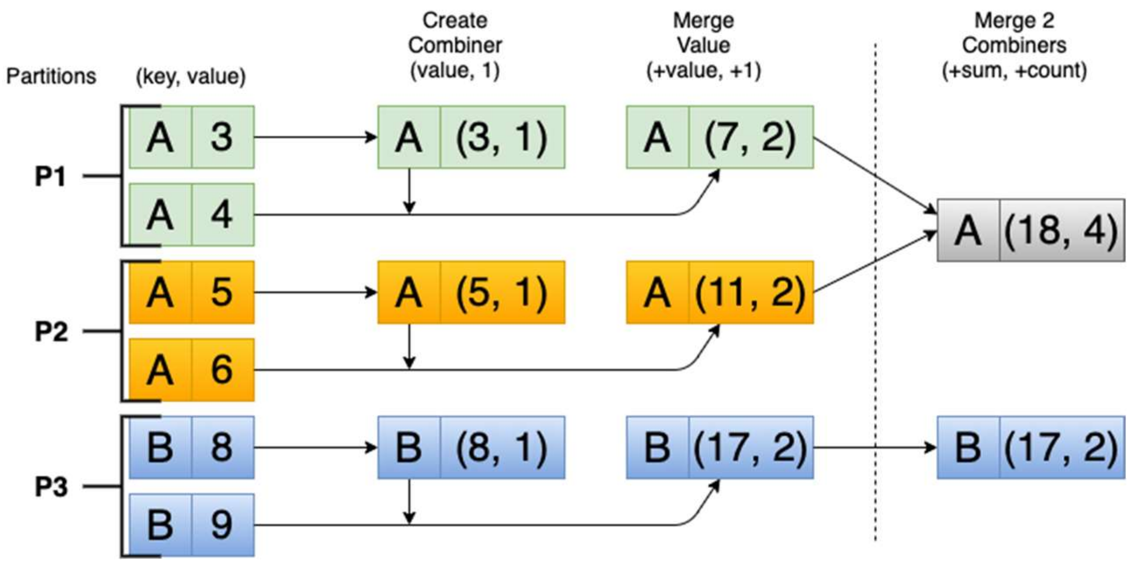


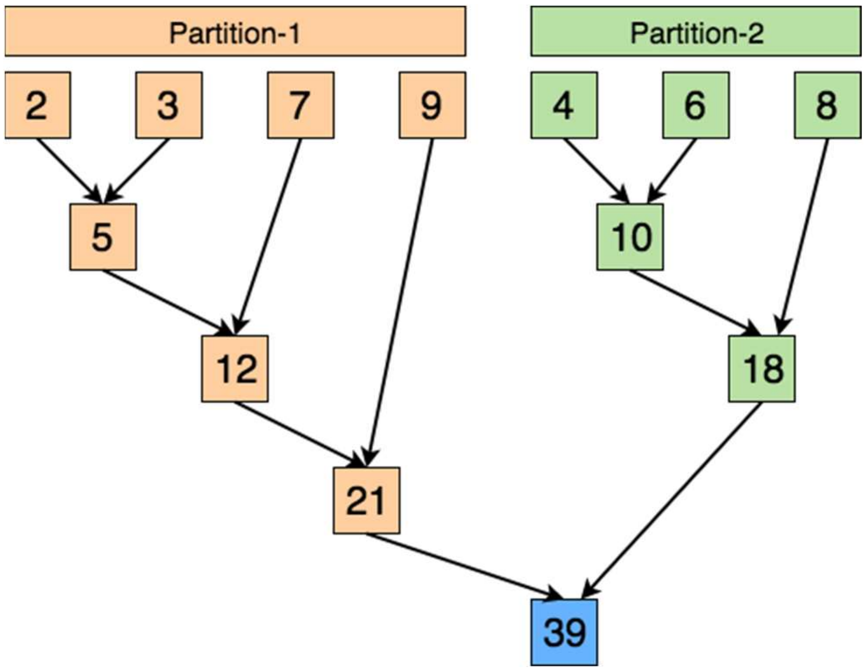

SparkSession or SparkContext
Data Files
S3
Collection
RDD
Transformation
Action
Tangible Result


Data
Data
RDD1
RDD3
map()
flatMap()
RDD2
RDD4
join()
RDD5
saveAsMap()
HashMap
k1
value1
k2
value2
...
...

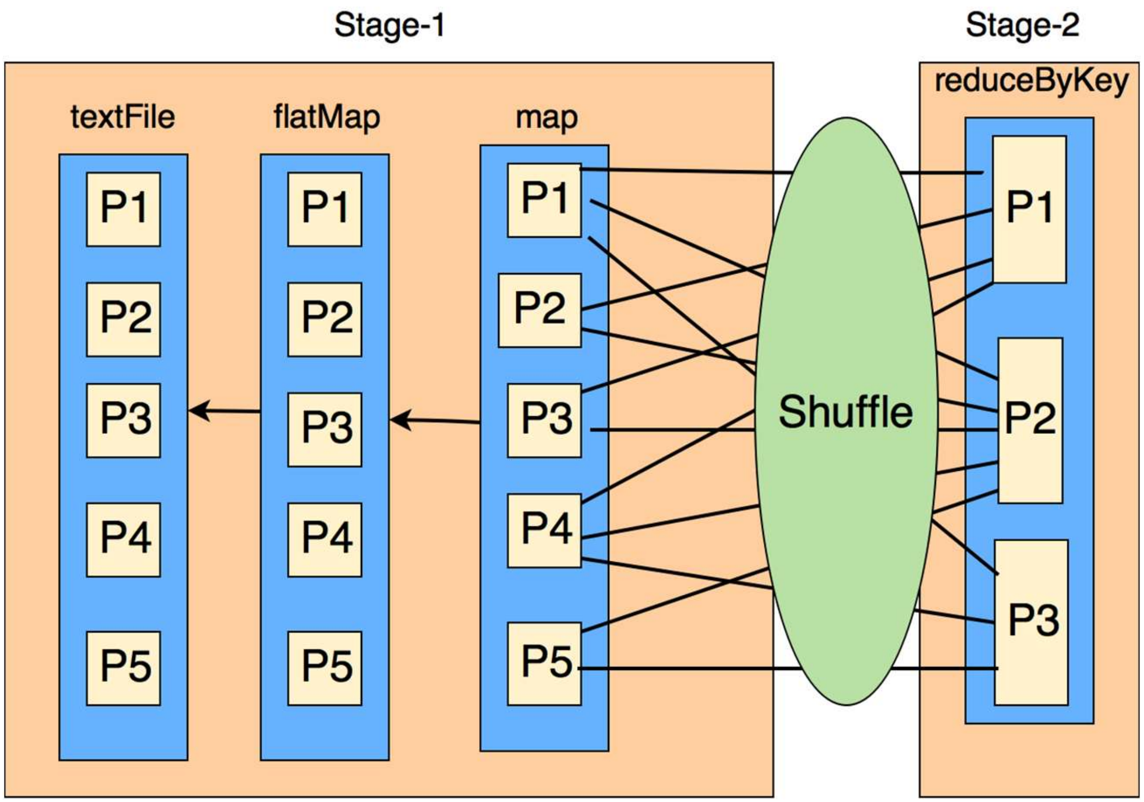
How it works in partitions

- **Reduction Concept**



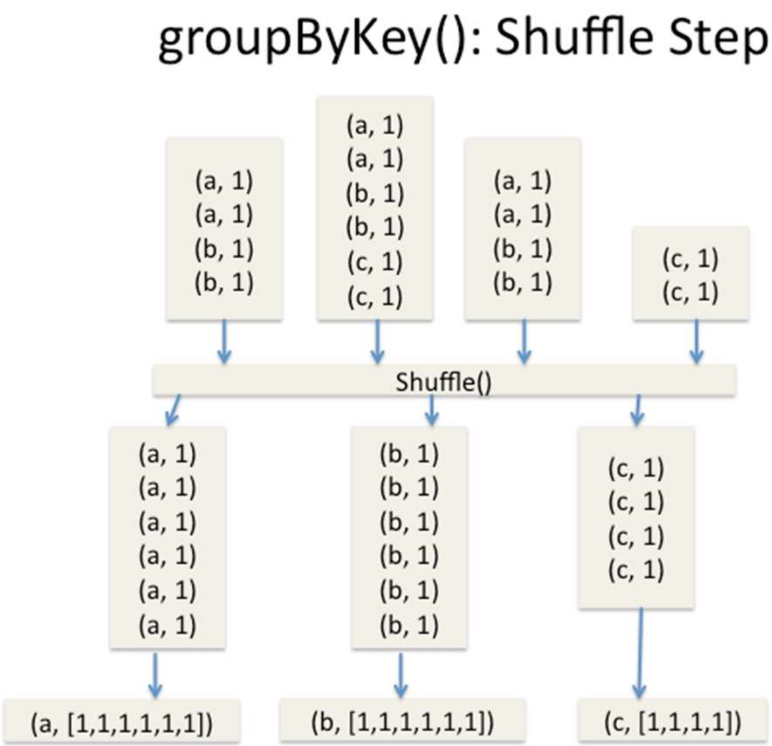


Shuffle

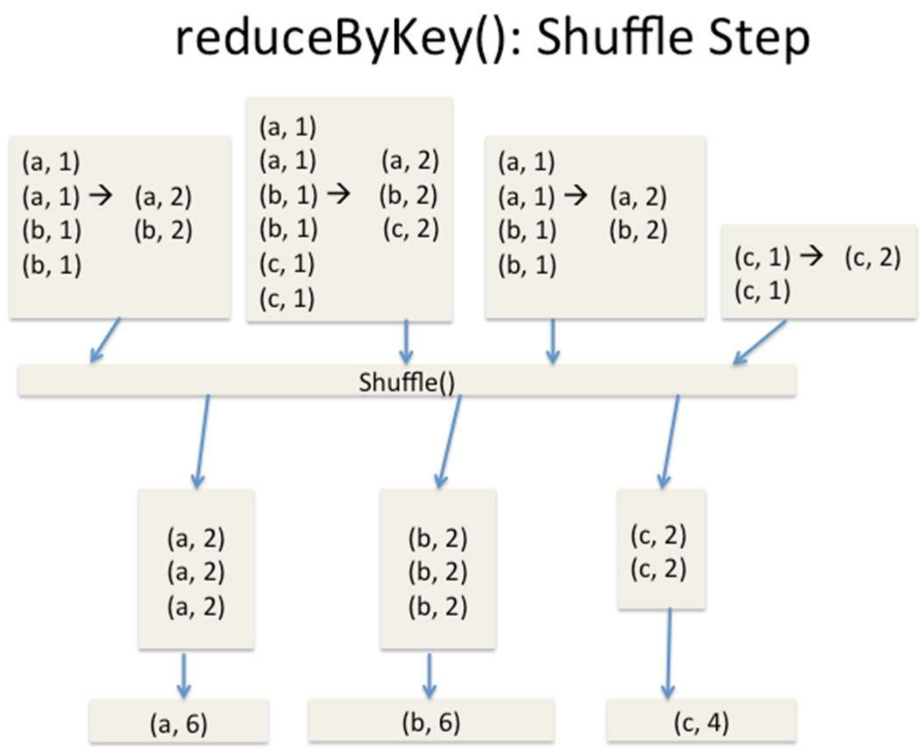


Stage-1
Stage-2
textFile
flatMap
map
reduceByKey
P1
P2
P3
P4
P5
P1
P2
P3
P4
P5
P1
P2
P3
P4
P5
Shuffle
P1
P2
P3

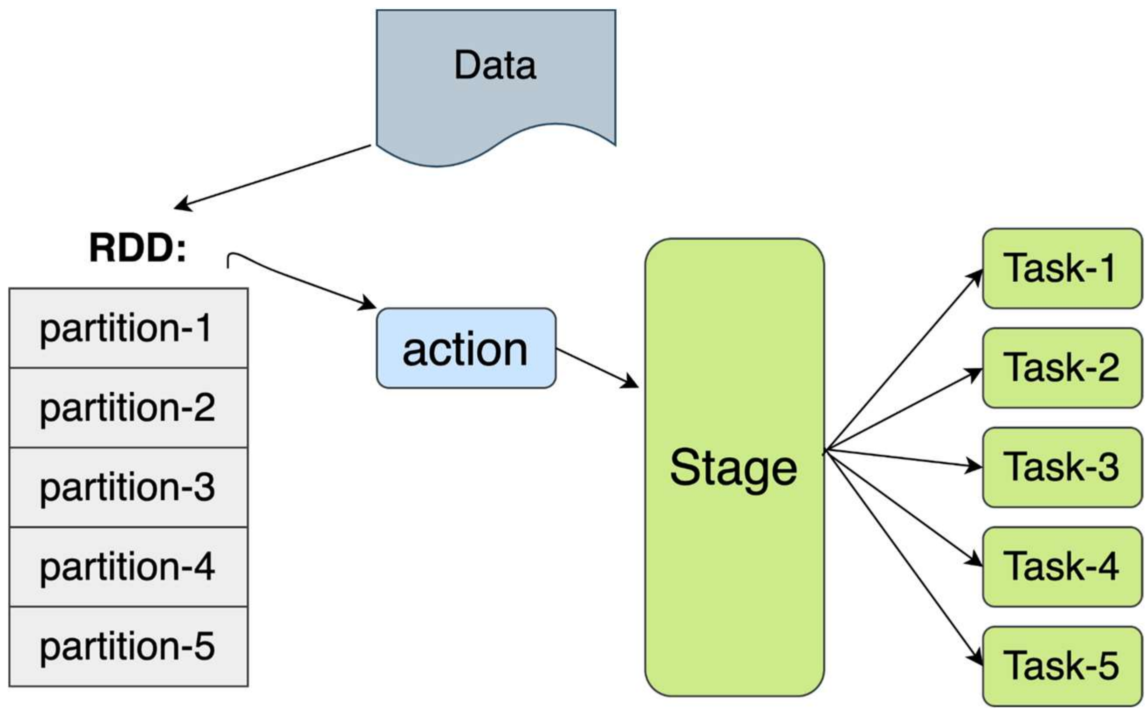
Shuffle


groupByKey(): Shuffle Step
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(c, 1)
(c, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(c, 1)
(c, 1)
Shuffle()
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(c, 1)
(c, 1)
(c, 1)
(c, 1)
(a, [1,1,1,1,1,1])
(b, [1,1,1,1,1,1])
(c, [1,1,1,1])

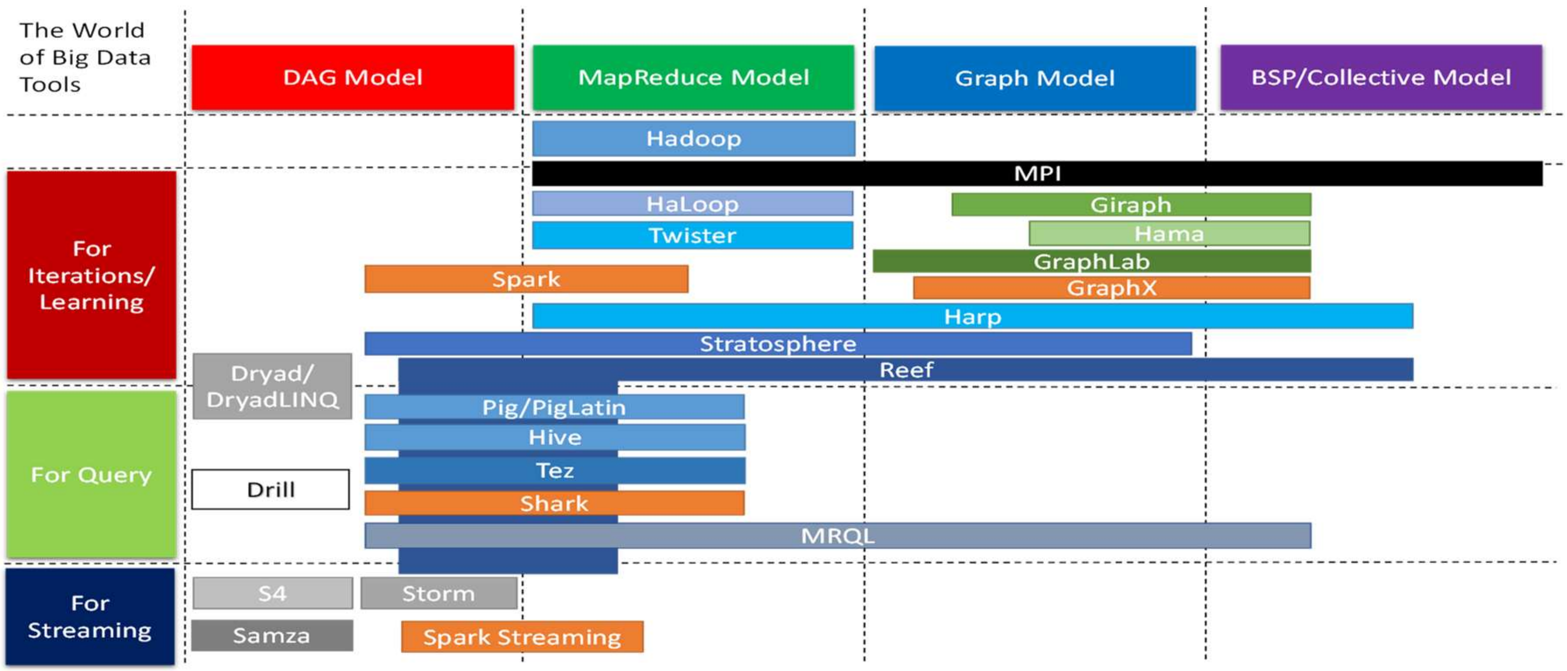
Shuffle



reduceByKey(): Shuffle Step
(a, 1)
(a, 1) → (a, 2)
(b, 1) (b, 2)
(b, 1)
(a, 1)
(a, 1) (a, 2)
(b, 1) → (b, 2)
(b, 1) (c, 2)
(c, 1)
(c, 1)
(a, 1)
(a, 1) → (a, 2)
(b, 1) (b, 2)
(b, 1)
(c, 1) → (c, 2)
(c, 1)
Shuffle()
(a, 2)
(a, 2)
(a, 2)
(b, 2)
(b, 2)
(b, 2)
(c, 2)
(c, 2)
(a, 6)
(b, 6)
(c, 4)


Data
RDD:
partition-1
partition-2
partition-3
partition-4
partition-5
action
Stage
Task-1
Task-2
Task-3
Task-4
Task-5

The Big Data tools Ecosystem



Batch Storage

MongoDB

MongoDB is **Document-oriented NoSQL** tool

Open source NoSQL DB

- In memory access to data
- Native replications toward reliability and high availability (CAP)
- Collection partitioning by using sharding key so to keep the information fast available and also replicated
- Designed in C++

MongoDB

Collection partitioning by using a **shard key**: **Hashed-based** to obtain a (not always) balanced distribution

Distributed architecture:

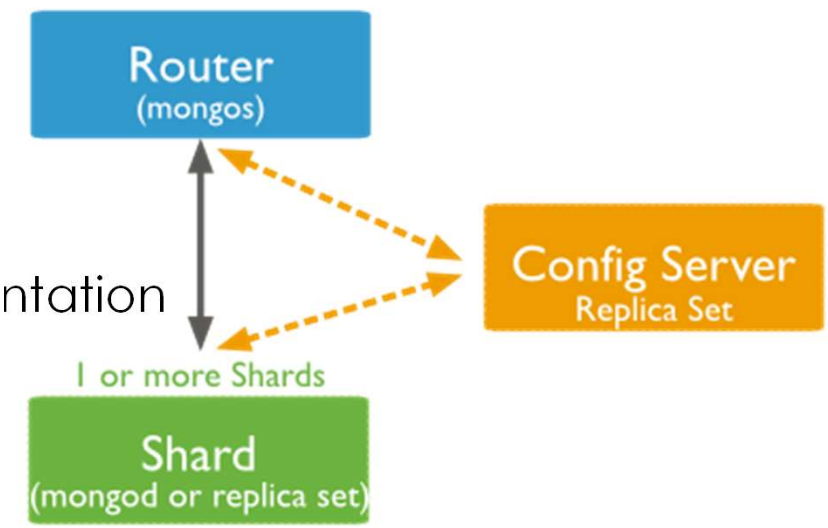
- **Router** to accept and route incoming requests coordinating with **Config Server**
- **Shard** to store data

Sharded cluster:

- Shard: partition containing subset of data
- Mongos: **query router**, interface between presentation

Layer and sharded cluster

- Config servers: config settings & metadata



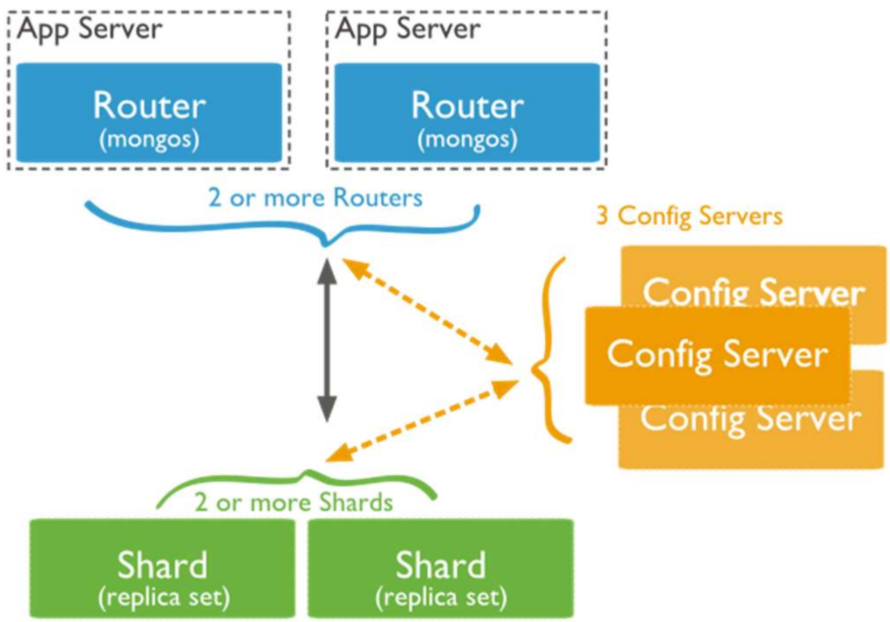
MongoDB in a deployment

The configuration can grant different properties

In a distributed architecture you may employ replication

Distributed architecture:

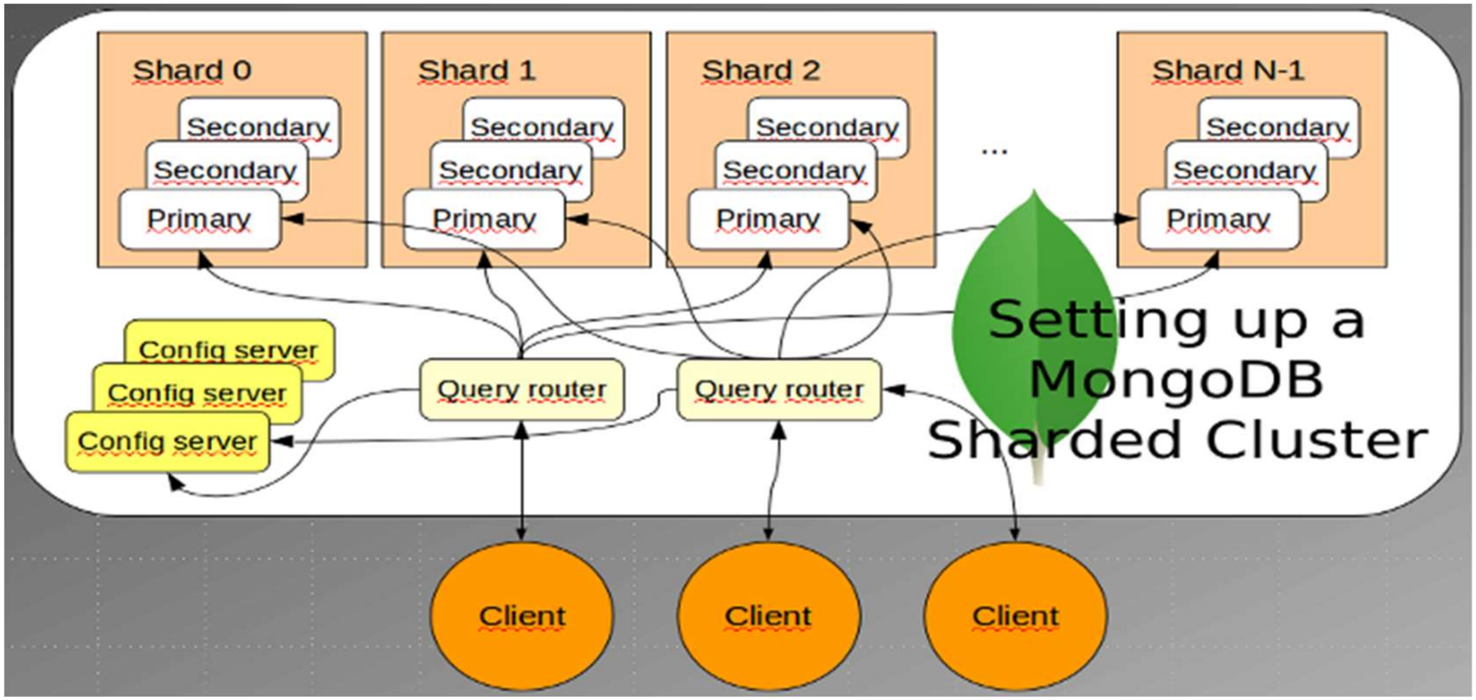
- **Several Routers** to accept incoming requests
- **Config Server to give access to requests**
- **Shards** to store data



The system is capable of supporting dynamic access to documents

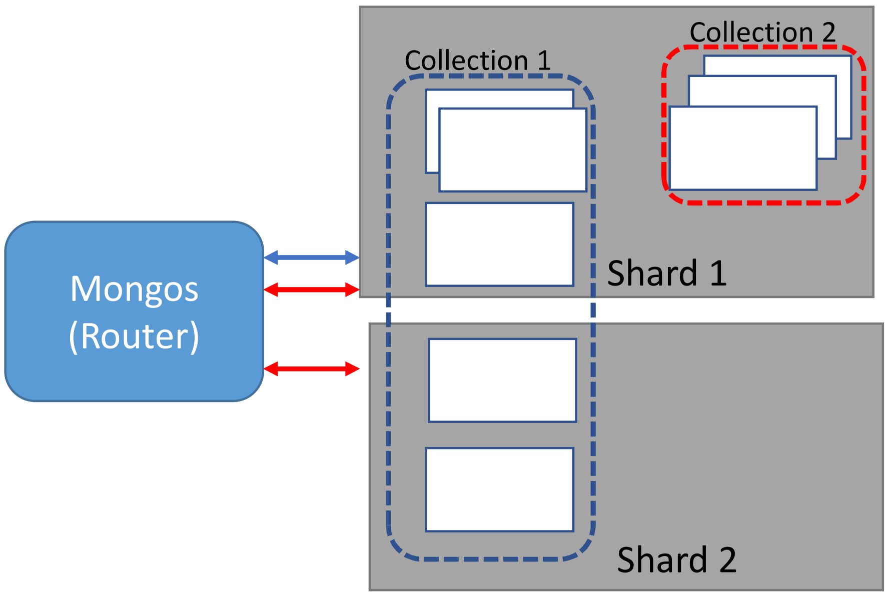
MongoDB

The configuration can grant different properties. In a distributed architecture you may define better



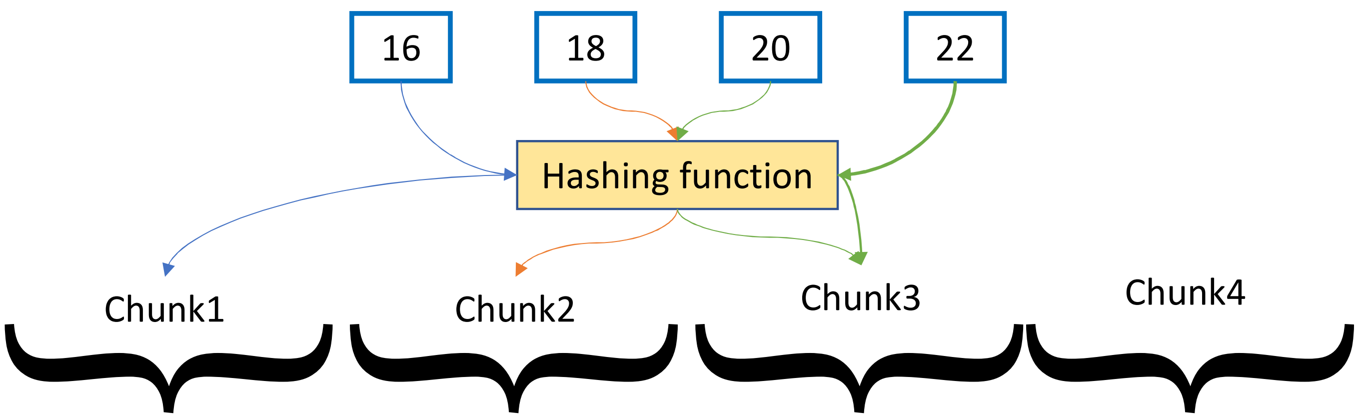
Sharding for high throughput operations

- MongoDB exploits **shard key** to divide a collection documents across multiple shards
- **Shard key** choice has a great impact on the **performance**, **efficiency**, and **scalability** of the cluster
    - A well-built cluster maybe **bottlenecked** by wrong shard key choice of
- Data is sharded into **chunks**
    - **Balancer** migrates chunks across shards to achieve **load balancing**
- Read/write workloads are distributed across shards for **higher throughput**
- Queries that include the shard key allows **targeted scans**, where **mongos** route the query request to specific shards
    - More efficient than **broadcasting (scatter/gather)**
    - Client apps interact with shards through Mongos



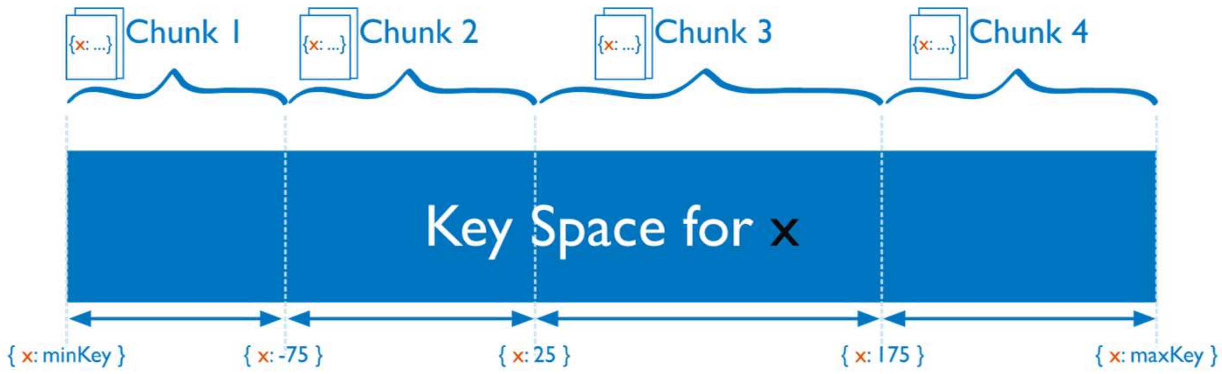
Hashed Sharding

- Computing a hash of the shard key field's value
  - Each chunk is then assigned a **range** based on the hashed shard key values
- Data distribution based on hashed values facilitates more **even data distribution**
  - hashed distribution means that **range-based queries** on the shard key are **less likely** to target a **single shard**, resulting in more cluster wide **broadcast operations**



Ranged Sharding

- **Range-based sharding** is the default sharding methodology
- **Dividing** data into **ranges** based on the shard key values
- Each chunk is then assigned a **range** based on the shard key values
- A range of shard keys whose values **are "close"** are more likely to **reside on the same chunk**
- The efficiency of ranged sharding relies on the shard key selected
    - Poorly selected shard keys cause **uneven distribution** of data, counteracting the benefits of sharding or causing performance degradation



MongoDB Data Model

Based on collections of documents

Stores data in form of **BSON** or Binary JSON

(**Binary JavaScript Object Notation**) ***documents***

```
{
    name: "travis",
    salary: 30000,
    designation: "Computer Scientist",
    teams: [ "front-end", "database" ]
}
```

Group of related *documents* with a shared common index is a ***collection***

- When designing a data model, consider how applications will use your database
  - if your application needs are mainly read operations to a collection, adding indexes to support common queries can improve performance.

Embedded Data

- **Embedded** documents capture relationships between data by storing related data in a single document structure
  - **embed** document structures in a **field** or **array** within a document
  - ***denormalized*** **data models** allow applications to retrieve and manipulate related data in a single database operation
  - **better** performance for **read** operations, as well as the ability to request and **retrieve** related data in a **single** database **operation**.

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

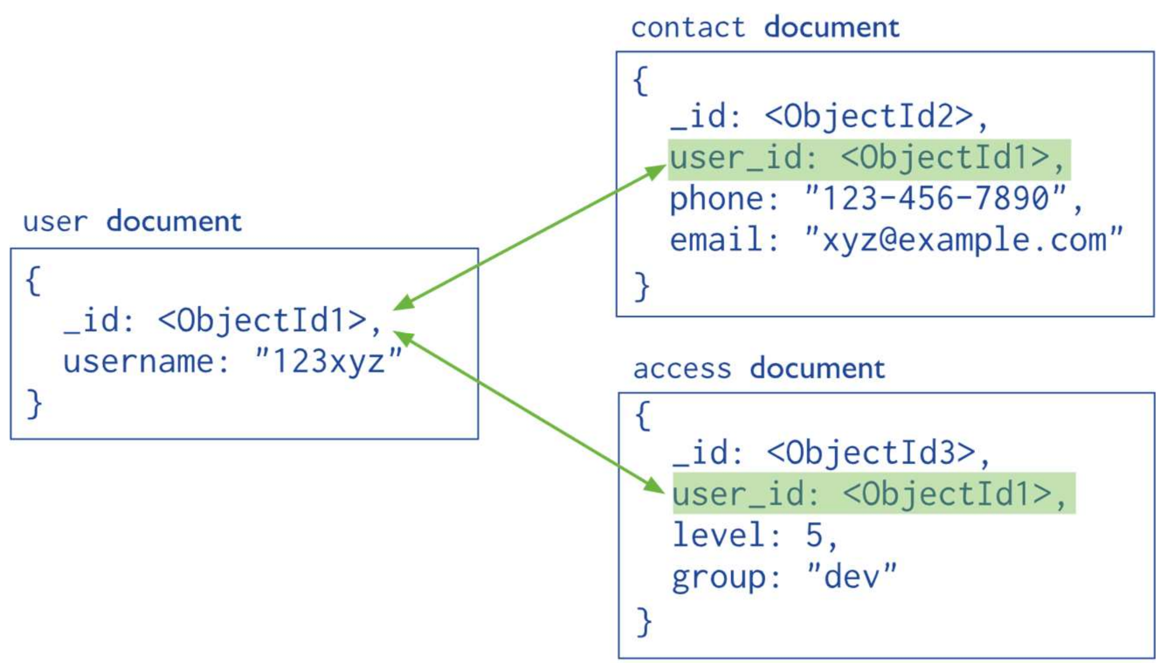
Embedded sub-document

References (normalized data models)

References store the relationships between data by including **links** or *references* from one document to another

Applications can resolve these references to access the related data

To **join** collections, MongoDB provides the **aggregation stages ($lookup)**

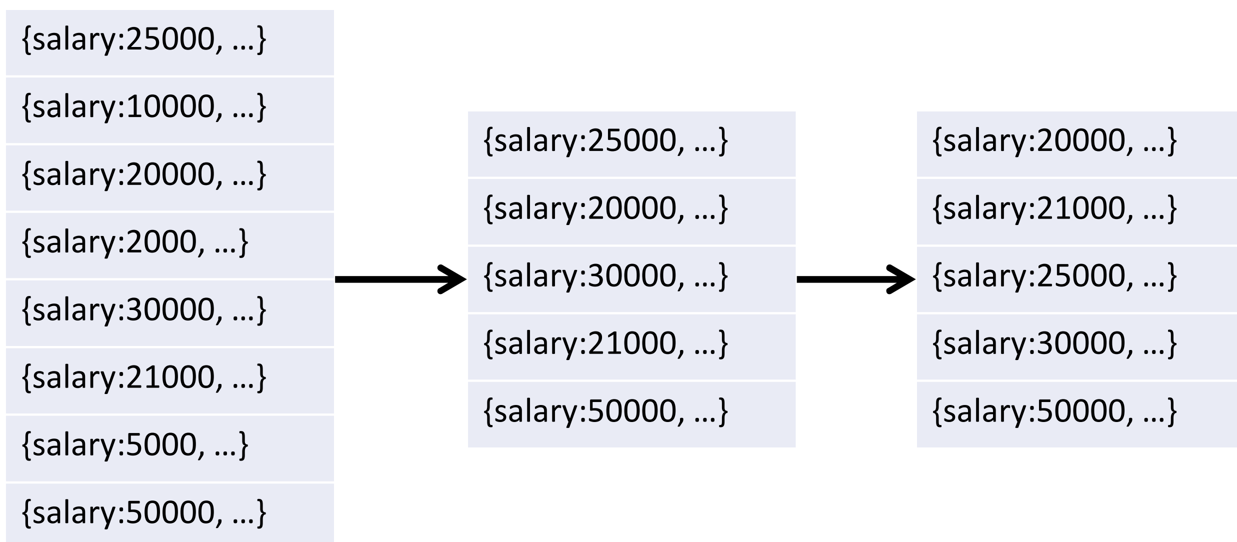


MongoDB: Typical Query

Query all employee names with salary greater than 18000 sorted in ascending order

```
db.employee.find({salary:{$gt:18000}, {name:1}}).sort({salary:1})
```

Collection | Condition | Projection | Modifier



Insert, Update, Remove

Insert: insert a row entry for new employee Sally

```
db.employee.insert({ name: "sally", salary: 15000, designation: "MTS", teams: [ "cluster-management" ] });
```

Update: All employees with salary greater than 18000 get a designation of Manager

```
db.employee.update( {salary:{$gt:18000}}, {$set: {designation: "Manager"}}, {multi: true})
```

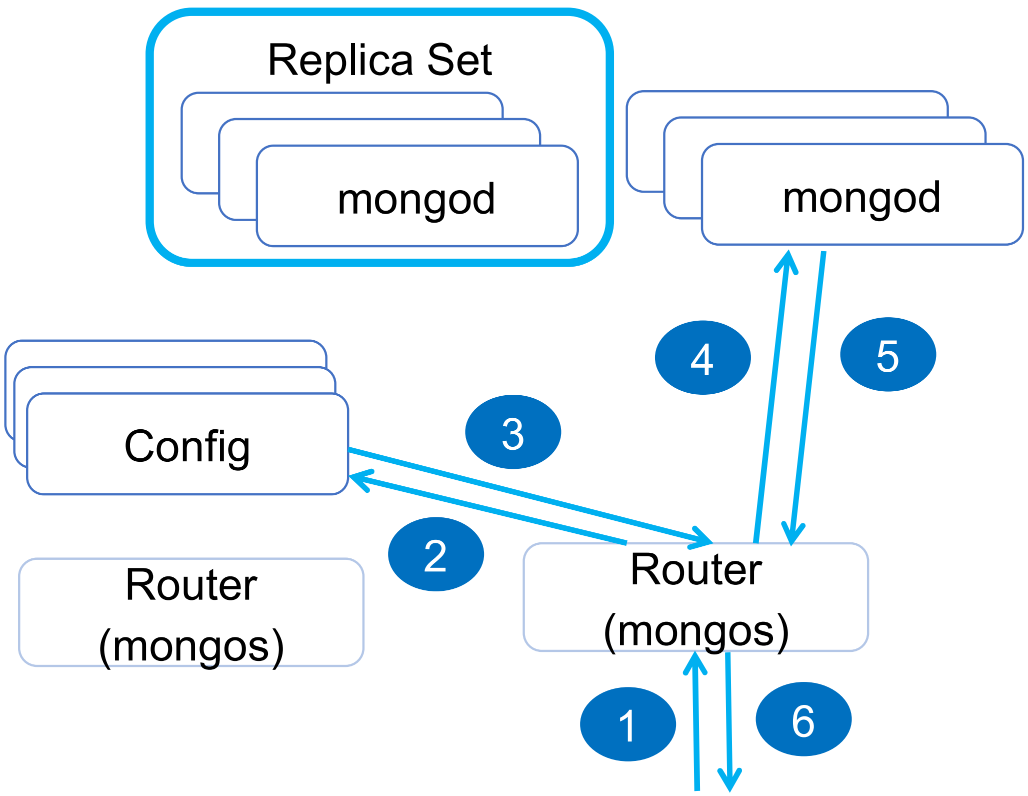
Multi-option allows multiple document update

Remove: remove all employees who earn less than 10000

```
db.employee.remove({salary:{$lt:10000}})
```

Can accept a flag to limit the number of documents removed

Typical MongoDB Deployment



- Data split into **chunks**, based on shard key (~ primary key)
  - Either use hash or range-partitioning
- **Shard**: collection of chunks
- Shard assigned to a replica set
- **Replica set** consists of multiple **mongod** servers (typically 3 mongod's)
- Replica set members are mirrors of each other
  - One is primary
  - Others are secondaries
- **Routers**: **mongos** server receives client queries and routes them to right replica set
- **Config server**: Stores collection level metadata.

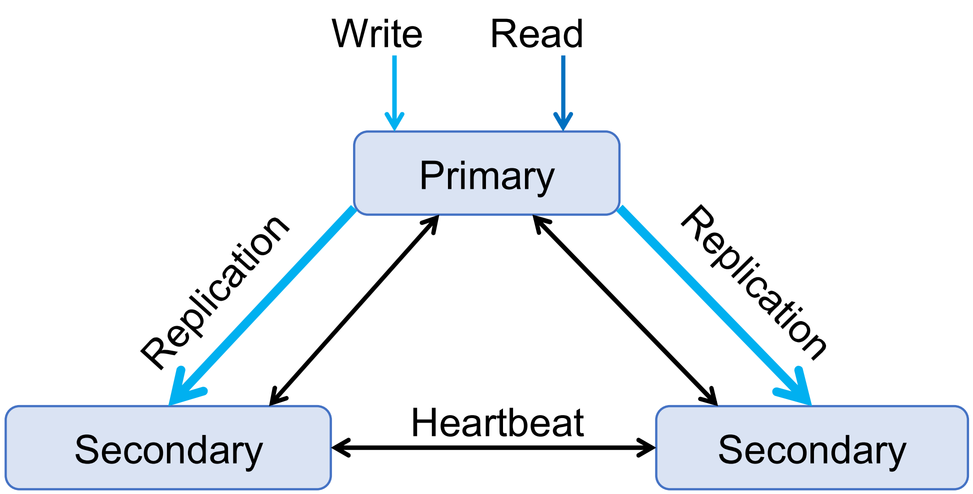
Replication

Uses an **oplog** (**operation lo**g) for data sync up:

- **Oplog** maintained at primary, delta transferred to secondary continuously/every **once** in a while

When needed, leader **Election protocol elects a master**

Some mongod servers do not maintain data but can vote – called as **Arbiters**



Read preferences

Determine where to route read operation.

Default is **primary**

Some other options are

- **Primary-preferred**
- **Secondary**
- **Nearest**

Helps reduce latency, improve throughput
Reads from secondary may fetch stale data

Write concern

Determines the guarantee that MongoDB provides on the success of a write operation

Default is ***acknowledged*** (primary returns answer immediately)

Other options are:

- **journaled** (typically at primary)
- **replica-acknowledged** (quorum with a value of W), etc.

Weaker write concern implies **faster write** time

Write concern

Determines the guarantee that MongoDB provides on the success of a write operation

Default is ***acknowledged*** (primary returns answer immediately)

Other options are:

- **journaled** (typically at primary)
- **replica-acknowledged** (quorum with a value of W), etc.

Weaker write concern implies **faster write** time

Journaling: Write-ahead logging to an on-disk journal for durability

(Journal may be memory-mapped)

Indexing: Every write needs to update every index associated with the collection

Balancing & Consistency

Balancing

Over time, some chunks may get larger than others

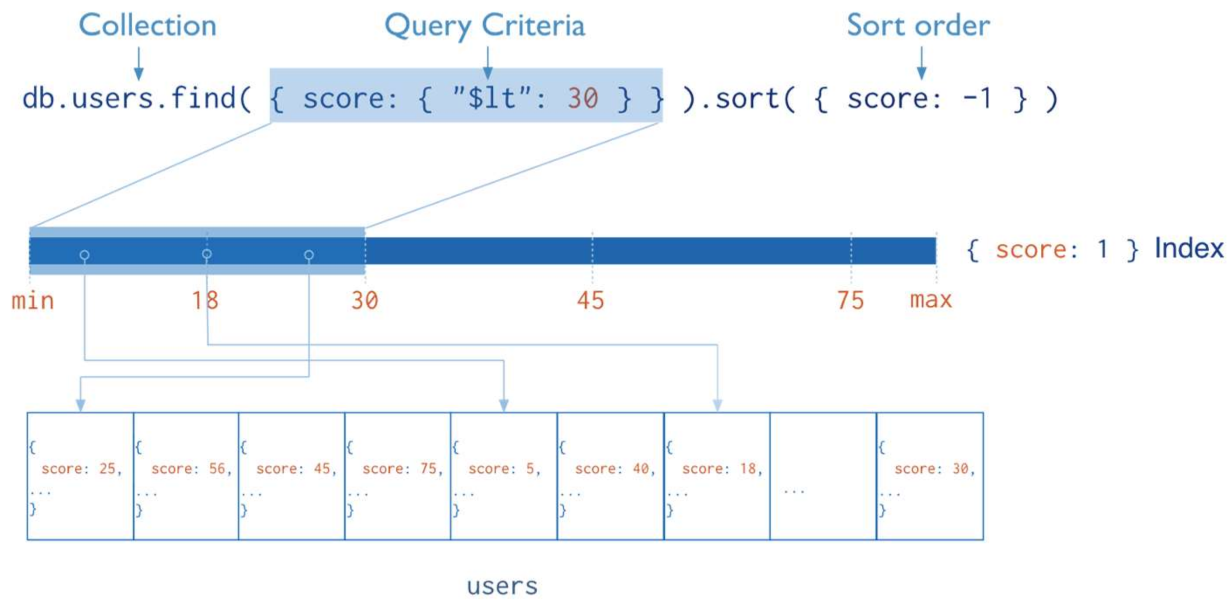
- **Splitting**: Upper bound on chunk size; when hit, chunk is split
- **Balancing**: Migrates chunks among shards if there is an uneven distribution

Consistency

- **Strongly Consistent**: Read Preference is Master. With Strong consistency, under partition, MongoDB becomes write-unavailable thereby ensuring consistency
- **Eventually Consistent**: Read Preference is Slave (Secondary or Tertiary)

Indexing in MongoDB

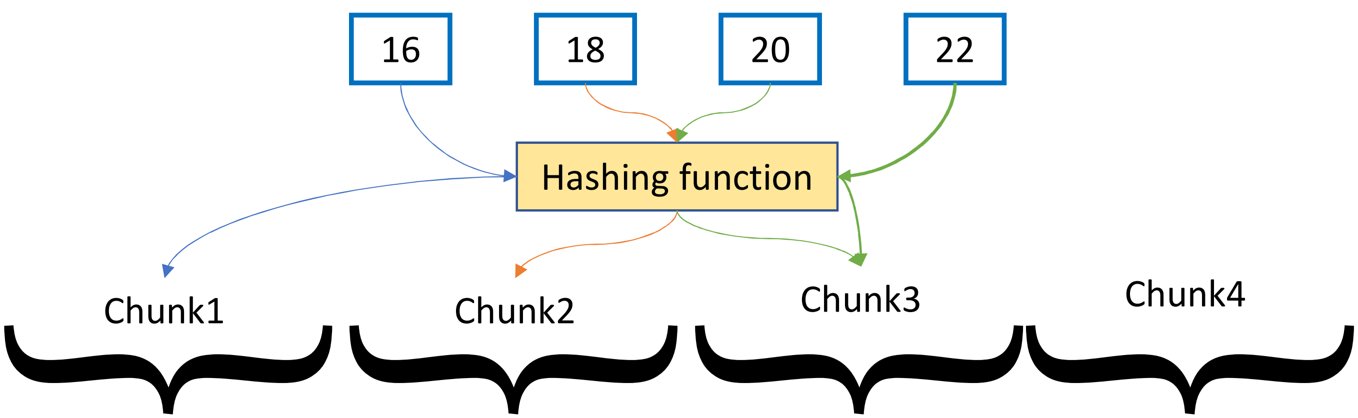
- Without indexes, collection scan (broadcast scan)
- Types
  - Single Field
  - Compound Index
  - Multikey Index
  - **Geospatial Index**
  - **Hashed Indexes**



Hashed Indexes

- Indexes the hash of the value of a field
  - Support **hash based sharding**
- *Only* support **equality matches** and cannot support **range-based** queries

- **Hashing function** is used to calculate the **hash** of the value of the index field



Things to consider when indexing

- you should have a deep **understanding** of your **application's queries**
- When your index **fits in RAM**, the system can avoid reading the index from disk and you get the fastest processing
- Indexes **fill up space** (each index requires 8 kB)
- Indexing can **negatively** impact **write** operations, for workloads with high **write-to-read** ratio
- Indexes are beneficial for workloads with high **read-to-write** ratio

Stream processing models

Stream Processing

There is more and more interest on **stream processing** … so …

Automatize everything – for **dedicate-purpose** behavior

data stream is a potentially **unbounded sequence of events**

monitoring data, sensor measurements, credit card transactions, weather station observations, online user interactions, web searches, etc.

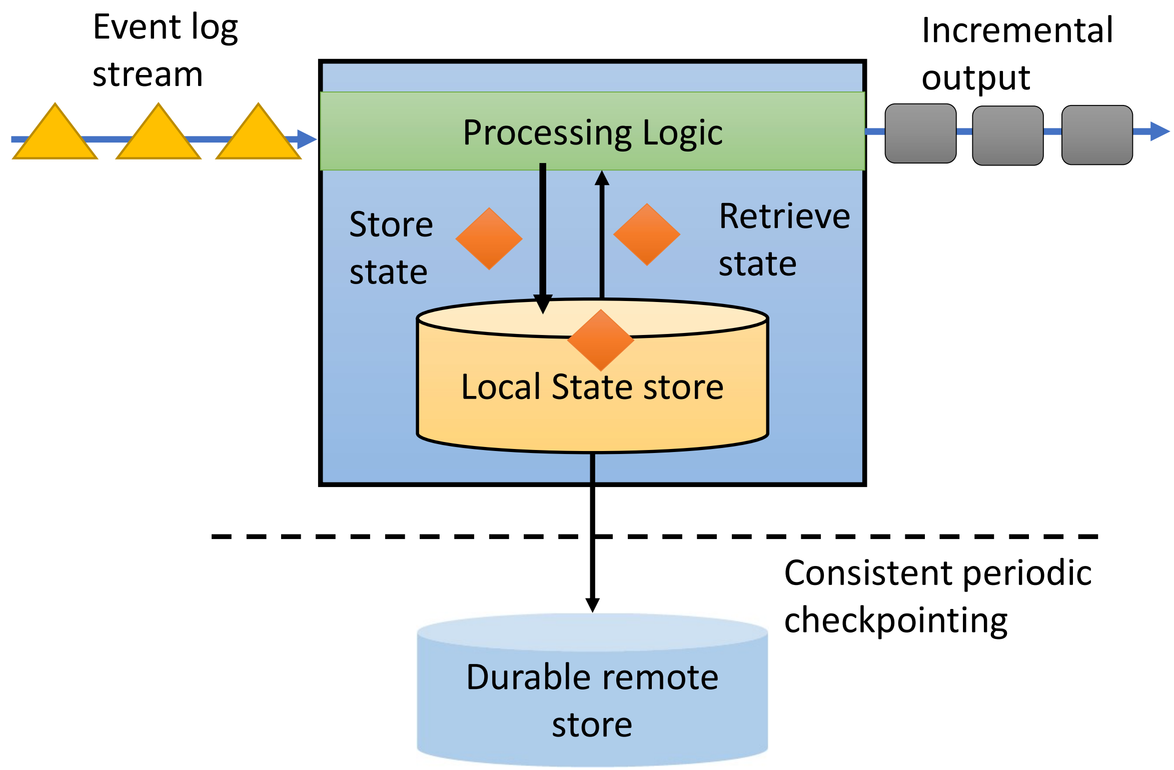
More and more set of tools become available to express and design a **complex streaming architecture** to be immediately deployed

- Apache **Storm**
- Yahoo **S4**
- Spark **Streaming (?)**
- Apache **Flink**

…

A stateful streaming application

- Applications normally process streams of events
  - Not just trivial record-at-a-time transformations
  - Need to be **stateful**
    - Storing and accessing intermediate results
  - Reading/writing data to the state
    - Variables, local files, embedded or external DBs
- Apache Flink
  - Writing **state** locally in-memory or to embedded DB
  - Periodically **consistent checkpointing** to remote and durable storage



Stateful stream processing

- Stateful stream processing applications **ingest** events from an event log
  - Event logs **store** and **distribute** event streams
  - Events are typically stored to a **durable**, **append-only** log, meaning that the order at which events are of written is **unchangeable**
  - **Apache Kafka** is the de facto event log system
- In failure cases, stream processors (e.g., Apache Flink) restores the latest known state from the last checkpoint and resets the read position in the event log
  - **Replaying** events from the event log until the stream tail is reached
- Three kinds of applications typically implemented by exploiting stateful stream processing:

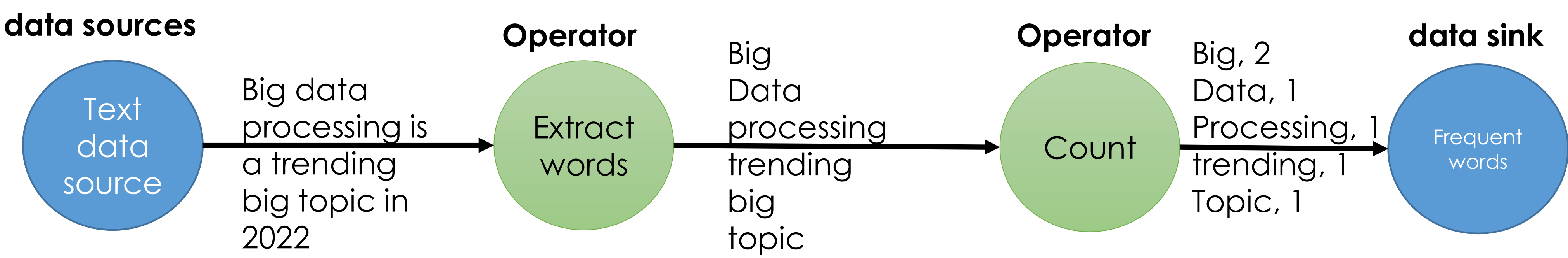
  (1) event-driven applications,

  (2) data pipeline applications, and

  (3) data analytics applications

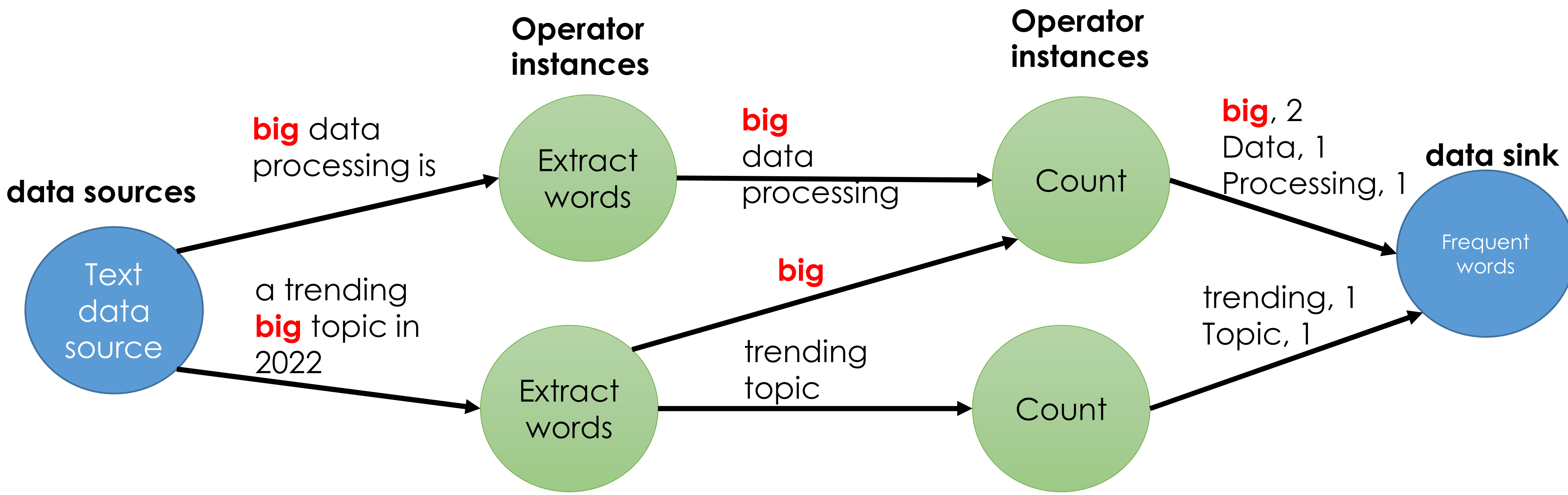
Dataflow programming paradigm

- **Dataflow** graphs specify the way data flows between operations
  - **Directed graphs**,
    - where nodes are known as **operators**, which represent **computations**
    - **edges** represent data dependencies
  - **Logical** graphs as because they present a high-level view of the involved computation logic
  - **Operators** are the primitive functional units
    - Ingest data from sources, perform a computational logic, and produce output data for subsequent stages
    - Operators with no input are known as **data sources**, while operators with no output are known as **data sinks**



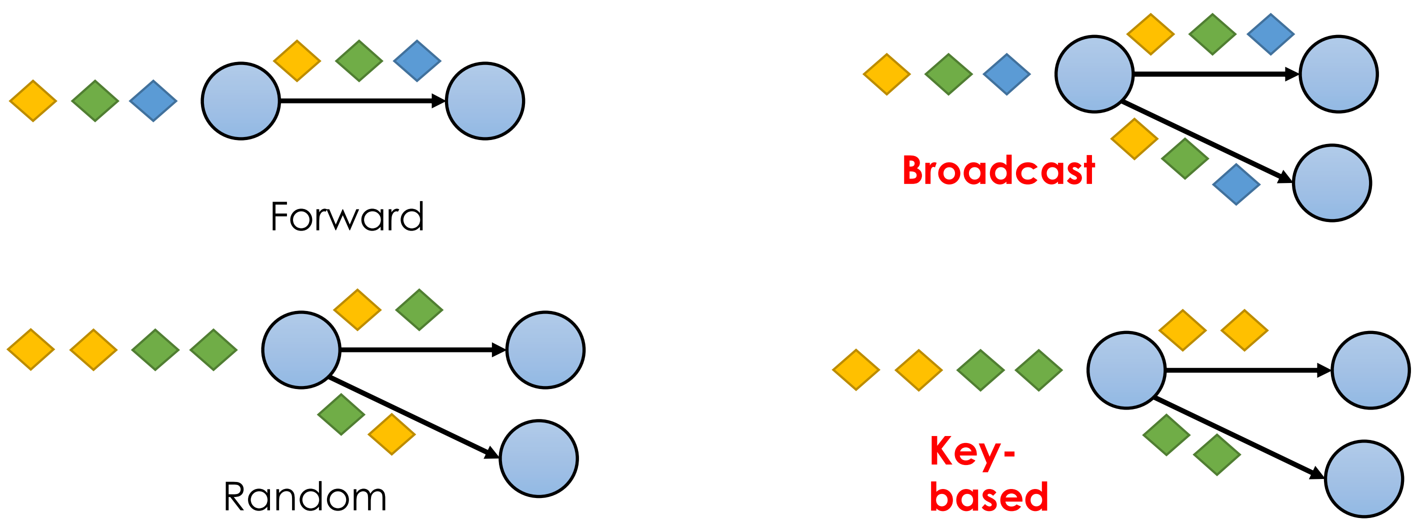
Dataflow programming paradigm (cont.)

- **Logical graph** will be converted to **physical dataflow graph**, which specifies in detail how the program is executed.
- In a distributed processing deployment
  - One operator with multiple parallelly running tasks, working on partitions of data stream.



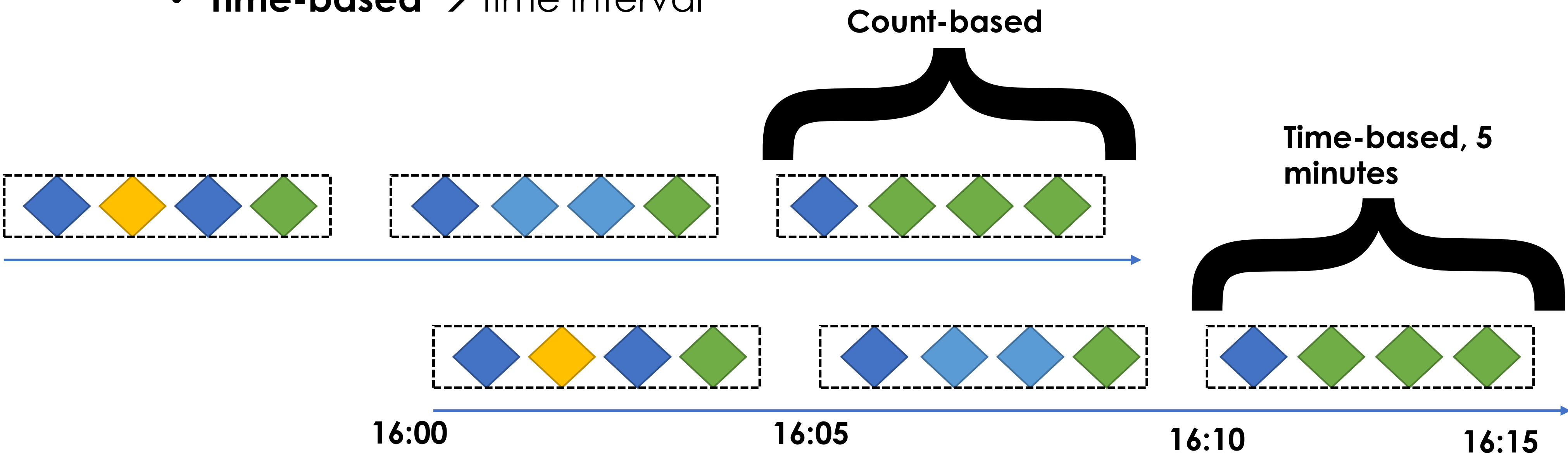
Data Exchange Strategies (online data partitioning)

- Specifies the way by which data tuples are distributed to parallelly connected physical dataflow graph tasks
- Strategies
  - ***Forward*** strategy. Forward data from one task to a subsequent task
  - ***Broadcast*** strategy. Sending the same copy of data to all parallelly connected instances (tasks) of an operator → **expensive**
  - ***Key-based*** strategy. Sends same-key tuples to the same operator instances (tasks)
  - ***Random*** strategy. Randomly assigning roughly equal data loads to parallel operator tasks (instances)



Common window types

- ***Tumbling*** windows
  - assign streaming events to non-overlapping fixed-size buckets (micro batches)
  - Evaluation function is triggered whenever a window border is crossed
    - **Count-based** → how many events before triggering the function
    - **Time-based** → time interval

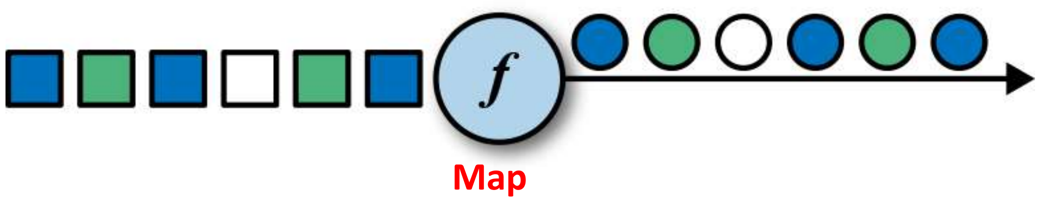


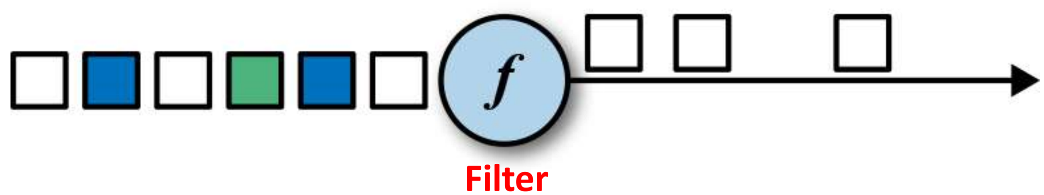
Transformations

- A stream transformation converts an input stream to an output stream
- Common transformations
  - Basic transformations → transformations on individual events
  - Multi-stream transformations → merge/split multiple streams

Basic transformations

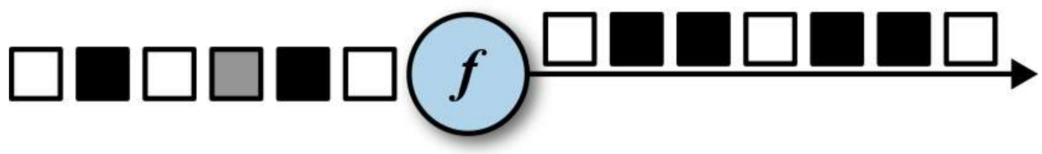
- Processing **single** events (one-record-at-a-time)
    - Single input tuple produce single output tuple
    - **Conversions**, records **filtering** and **splitting**
- **Map** transformation: a user-defined mapper produces an output from an input tuple, possibly with different type
- **Filter** transformation : a Boolean condition decides wether to drop tuples





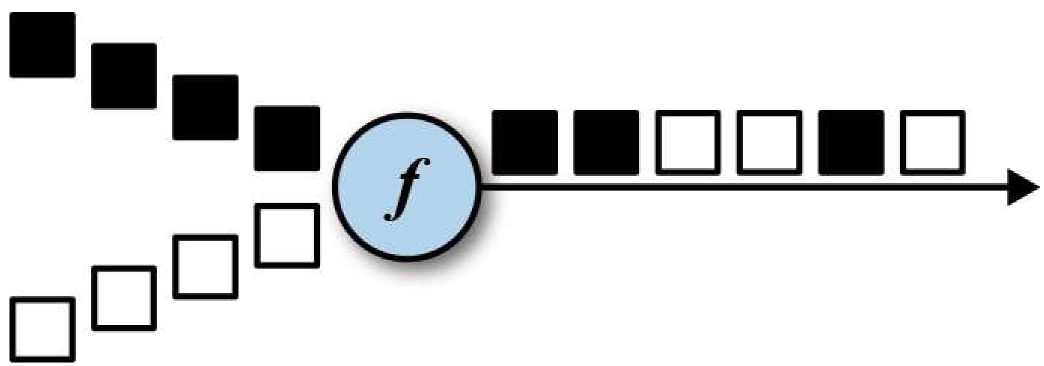
Basic transformations

- **FlatMap**
  - similar to map, but may result in zero, one, or more output tuples for each incoming input tuple



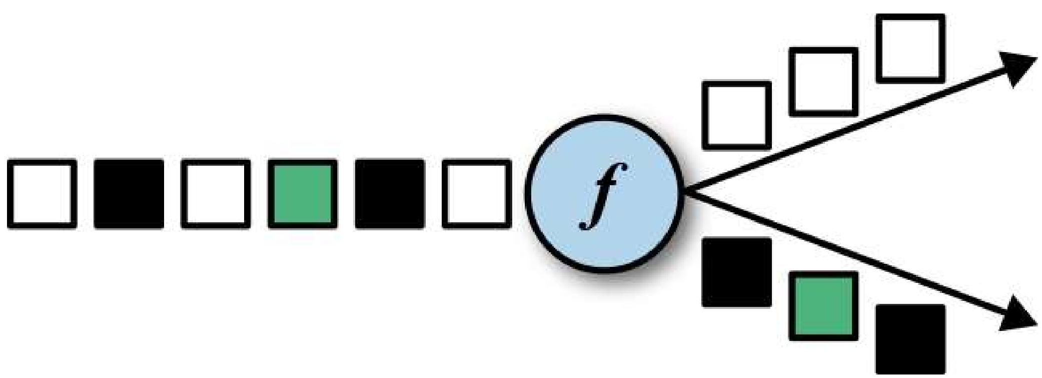
Multi-stream transformations

- Merging multiple streams or split a stream to sub-streams
- **UNION**
    - **merges** two or more streams of the same type and output a new stream having same type
    - Subsequent transformations process the elements of all combined input streams



Multi-stream transformations (cont.)

- **SPLIT**
  - **Splits** an input stream to two or more sub-streams having same type as the input stream
    - Incoming tuples are assigned to zero, one, or more output streams



Stream Processing Challenge

Large amounts of data → Need for **real-time views of data**

- Social network trends, e.g., *Twitter real-time search*
- Website statistics, e.g., *Google Analytics*
- Intrusion detection systems, *e.g., in most datacenters*

Process large amounts of data

- **with latencies of few seconds**
- **with high throughput**

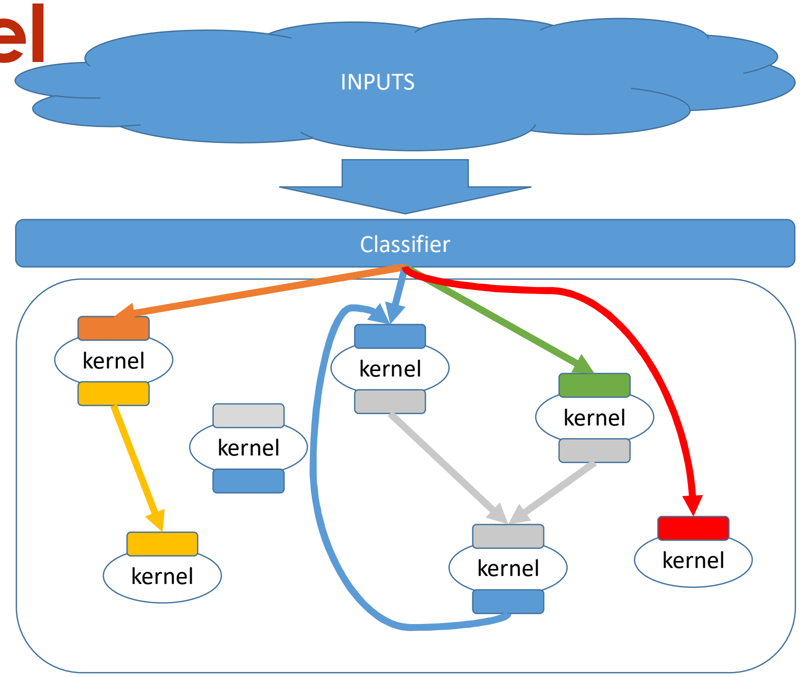
Not MapReduce

The out-of-line workflow is not suitable at all

The typical **Batch Processing → need to wait for entire computation on large dataset before completing**

In general batch approaches are not intended for **long-running stream-processing**

Stream Processing Model



Stream processing manages:

- Allocation
- Synchronization
- Communication

Applications that benefit most of the **streaming model** with requirements:

- **High computation resource intensive**
- **Data parallelization**
- **Data time locality**

Stream processing support functions

We need **available some basic functions** that can help in **mapping the concepts** we need to express

Storm is fast in **processing over a million tuples per second per node**: it is **scalable, fault-tolerant, respecting SLA** over data to be processed

Main functions must support the **stream processing** model:

- **Resource allocation**
- **Data classification**
- **Information routing in flows**
- **Management of execution/processing status**

STORM

Apache Project **http://storm.apache.org/**

Highly active **Java based JVM** project

Multiple languages supported via user API:

- Python, Ruby, etc.

Over 50 companies use it, including:

- **Twitter**: for personalization, search
- **Flipboard**: for generating custom feeds
- Spotify, Groupon, Weather Channel, WebMD, etc.

STORM

Apache Project **http://storm.apache.org/**
Highly active **Java based JVM** project
Multiple languages supported via user API:

- Python, Ruby, etc.

Over 50 companies use it, including:

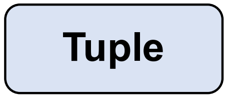
- **Twitter**: for personalization, search;
- **Flipboard**: for generating custom feeds;
- Spotify, Groupon, Weather Channel, WebMD, etc.

Core Components:
Tuples, Streams, Spouts, Bolts, Topologies

Tuple

We have already seen tuple as a set of values according to some attributes

The tuple is an ordered list of elements



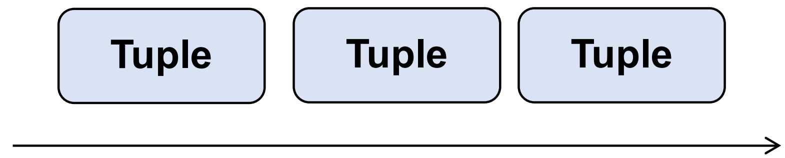
E.g., <tweeter, tweet>

- E.g., <“Miley Cyrus”, “Hey! Here’s my new song!”>
- E.g., <“Justin Bieber”, “Hey! Here’s MY new song!”>

E.g., <URL, clicker-IP, date, time>

- E.g., <coursera.org, 101.102.103.104, 4/4/2014, 10:35:40>
- E.g., <coursera.org, 101.102.103.105, 4/4/2014, 10:35:42>

Stream



Sequence of tuples

Tuples potentially **unbounded in number**

Social network example:

<“Miley Cyrus”, “Hey! Here’s my new song!”>,

<“Justin Bieber”, “Hey! Here’s MY new song!”>,

<“Rolling Stones”, “Hey! Here’s my old song that’s still a super-hit!”>, …

Website example:

<coursera.org, 101.102.103.104, 4/4/2014, 10:35:40>,

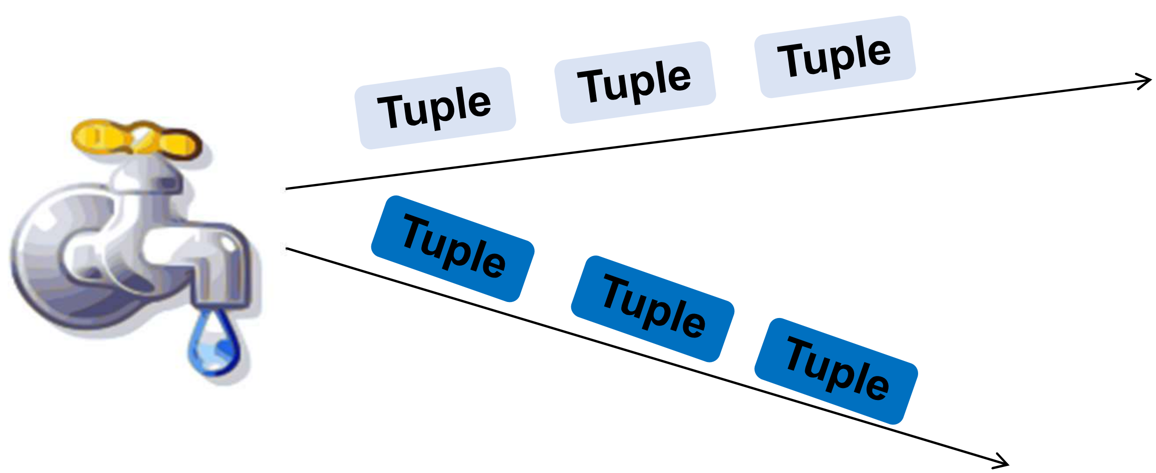
<coursera.org, 101.102.103.105, 4/4/2014, 10:35:42>, …

Spout

One **spout is a Storm entity** (process) that is a **source of streams (set of tuples)**
Often reads from a crawler or DB

Spouts normally read data from an external data source and emit tuples into the topology

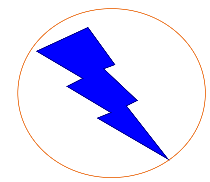
Spouts don't perform any processing; they simply act as a source of streams, reading from a data source and emitting tuples to the next type of node in a topology: the bolt

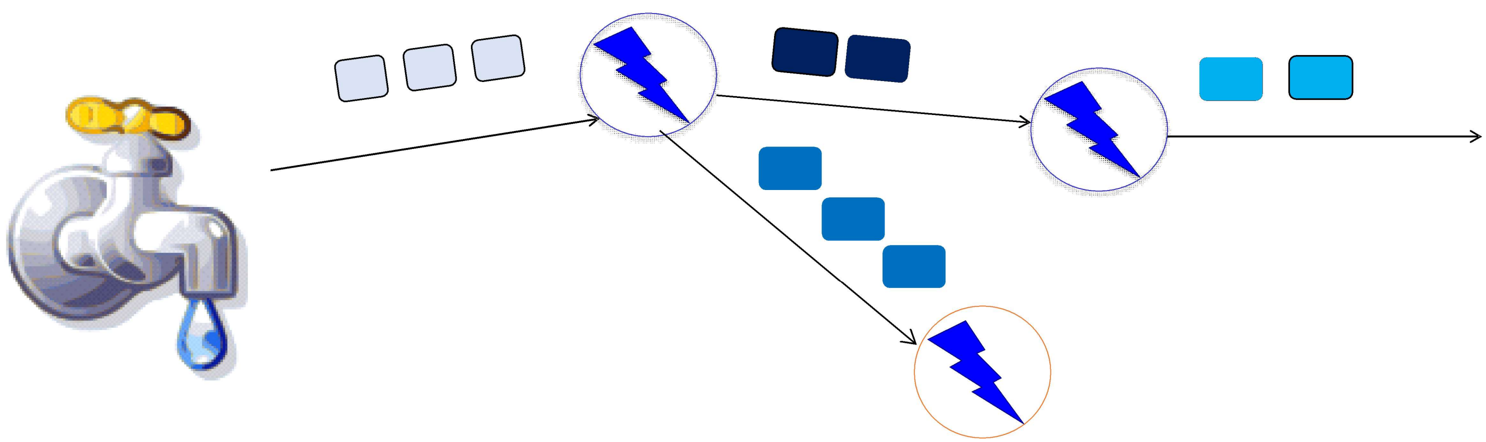


Spout

A **bolt is a Storm entity** (process) that

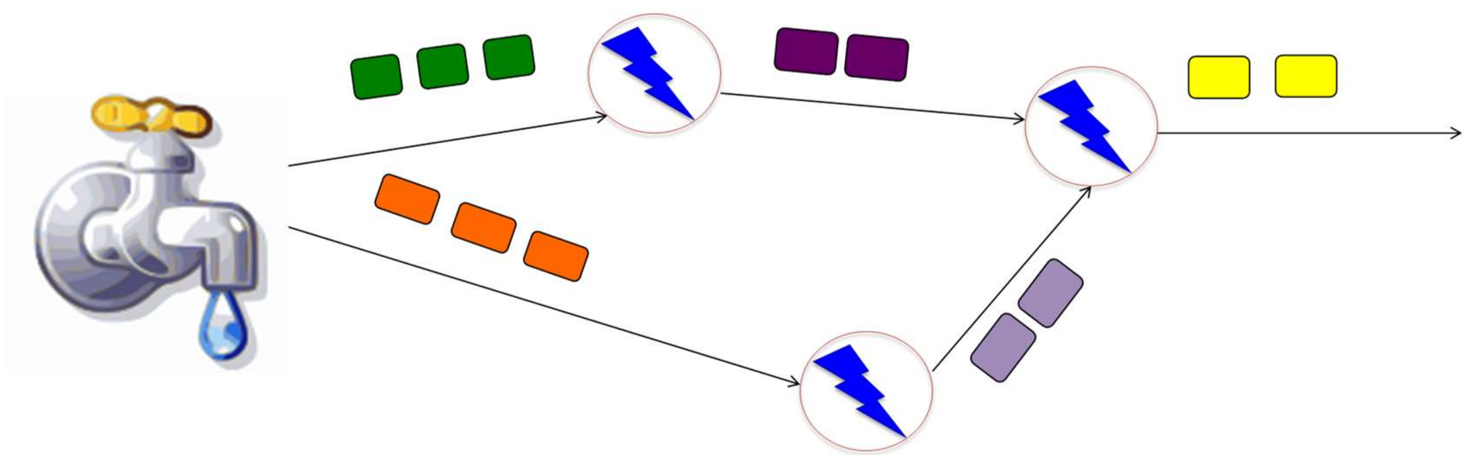
- Processes input streams
- Outputs more streams for other bolts





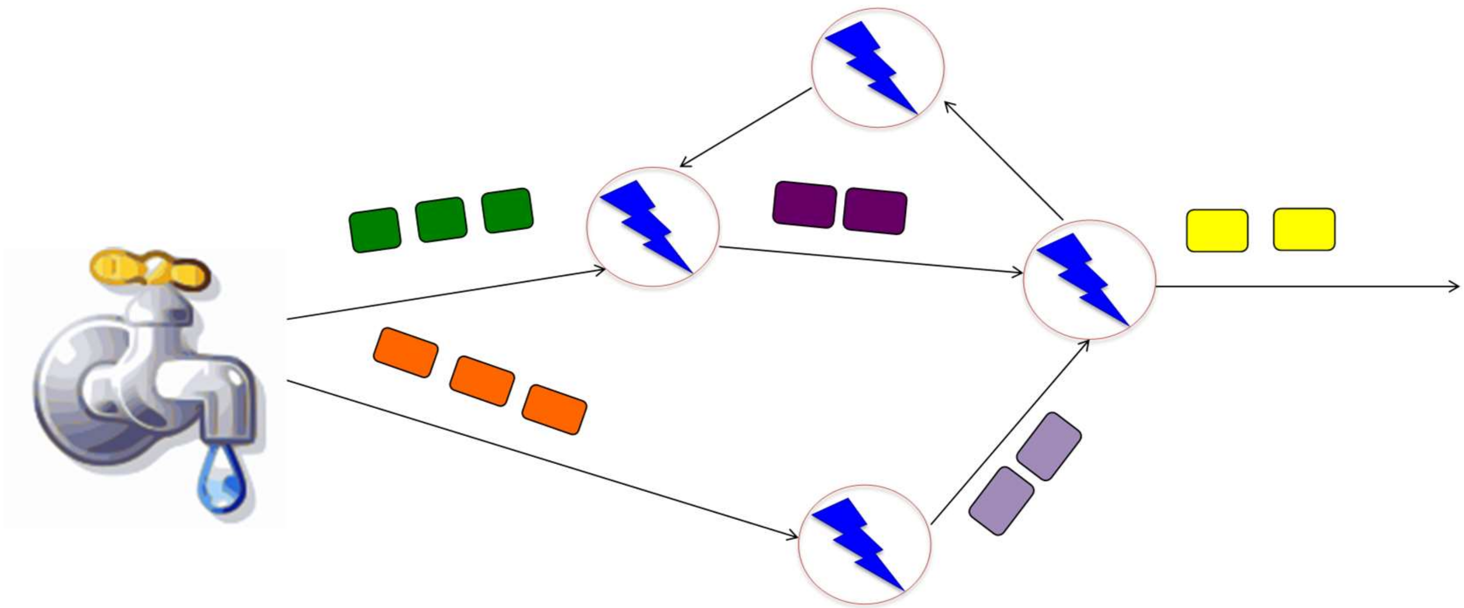
Topology

A **directed graph** of **spouts** and **bolts** (and output bolts)
Corresponds to a Storm "application"



Topology

A **Storm topology** may define an architecture that can also have **cycles** if the application needs them



Bolts come in many Flavors

Operations that can be performed

- **Filter**: forward only tuples which satisfy a condition
- **Joins**: When receiving two streams A and B, output all pairs (A,B) which satisfy a condition
- **Apply/transform**: Modify each tuple according to a function
- …And many others

But bolts need to process a lot of data

- Need to make them fast

Parallelizing Bolts

Storm provides also **multiple processes ("tasks") that can constitute a bolt**

Incoming streams split among the tasks

Typically each **incoming tuple goes to one task** in the bolt

- Decided by "Grouping strategy"

Grouping

Three **types of grouping** are popular

Shuffle Grouping

- Streams are distributed evenly among the bolt tasks
- Round-robin fashion

Fields Grouping

Group a stream by a subset of its fields such as

- all tweets where twitter username starts with [A-M,a-m,0-4] goes to task 1, and
- all tweets starting with [N-Z,n-z,5-9] go to task 2

All Grouping

- All tasks of bolt receive all input tuples
- Useful for joins

Failure behavior

Also **failures can be mapped**

A tuple is considered failed when its topology (graph) of **resulting tuples fails to be fully processed within a specified timeout (time dimension)**

Anchoring: Anchor an output to one or more input tuples

- Failure of one tuple causes one or more tuples to be replayed

API For Fault-Tolerance (OutputCollector)

Emit (tuple, output)

- **Emits an output tuple**, perhaps anchored on an input tuple (first argument)

Ack (tuple)

- Acknowledge that a bolt **finished** processing a tuple

Fail (tuple)

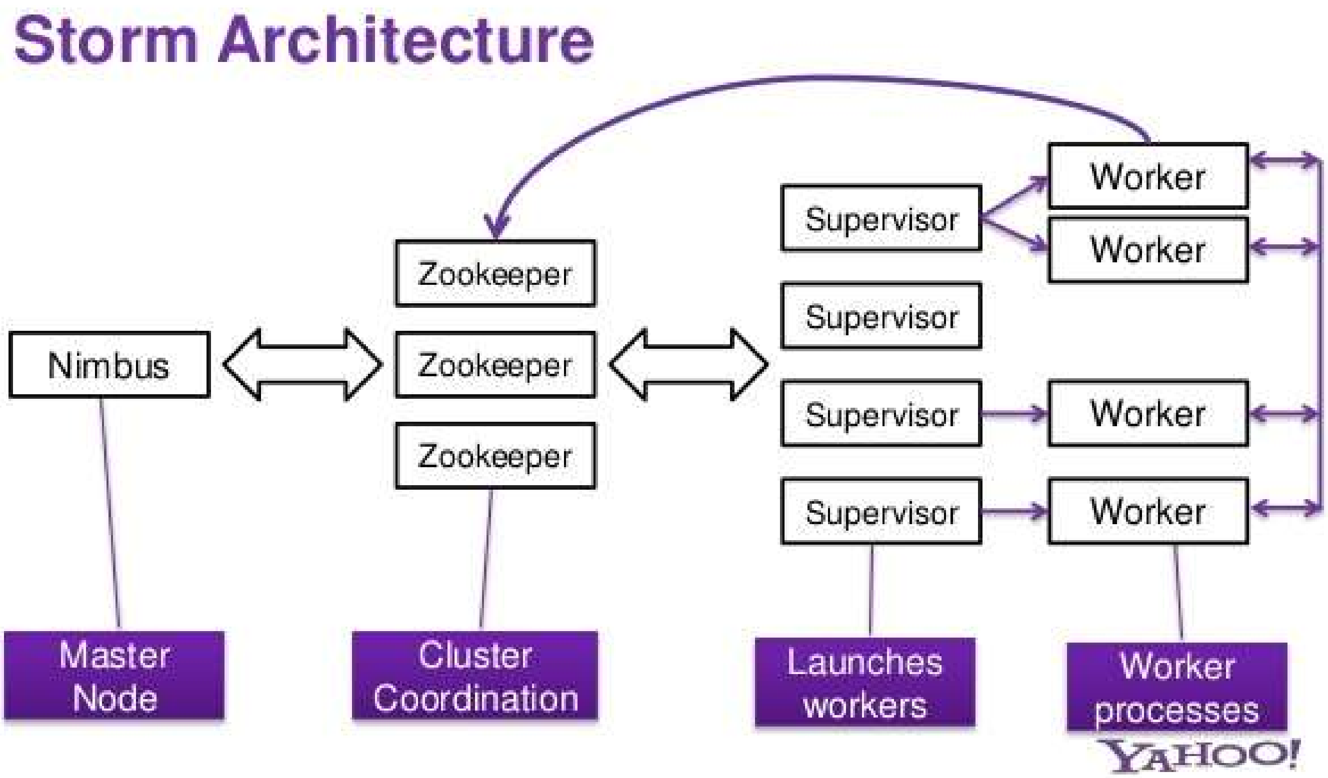
- Immediately fail the spout tuple at the root of tuple topology if there is an exception from the database, etc.

Must Record the **ack/fail of** each tuple

- Each tuple consumes memory. Failure to do so results in memory leaks.

Storm Cluster

Several components in a Cluster



Zookeeper

ZooKeeper is an open-source Distributed Coordination Service for Distributed Applications:

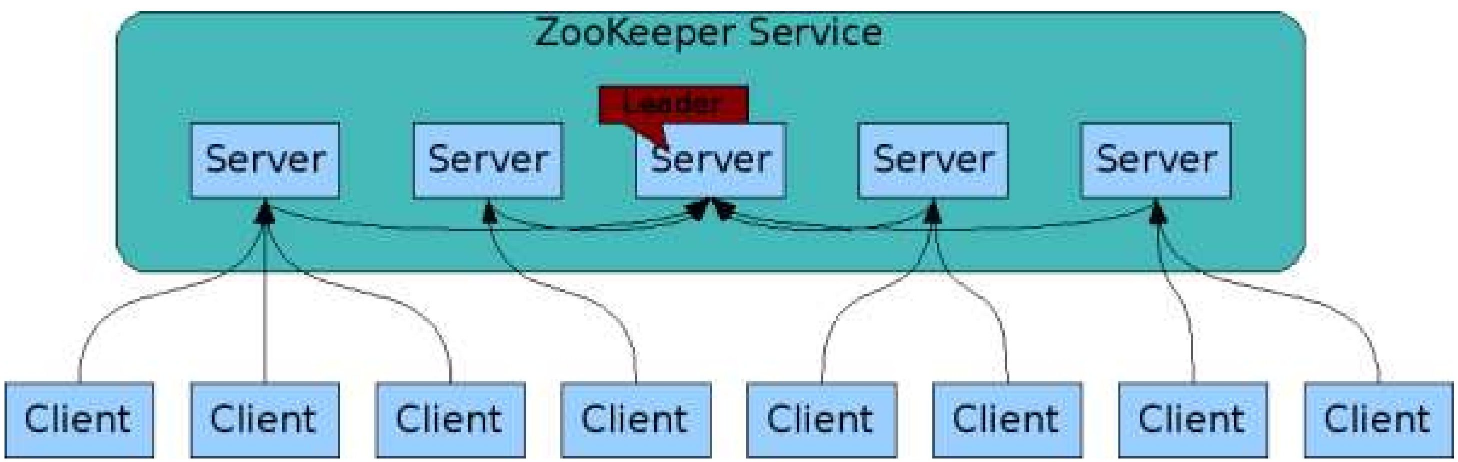
- can propose a unique **memory space with very fast access in reading and writing** with **some quality** (QoS: replication is paramount and dynamicity too)
- relieves distributed applications from **implementing coordination services from scratch**
- exposes a **simple set of primitives** to implement **higher level services for synchronization, configuration maintenance, and groups and naming**

The data model is shaped after the familiar **directory tree structure of file systems** and it runs in Java with bindings for both Java and C

Zookeeper

ZooKeeper is seen as **a unique access space with very fast operations to read and write data with different semantics (FIFO, Atomic, Causal, …)**

Data are **dynamically mapped over several nodes** and their location can be dynamically changed and adjusted without any actions of clients.



Storm Architecture

Storm allows to:

1. First express your need **in streaming via its components** you can easily define and design
2. Secondly, configure your **capacity needs over a real architecture** so to produce a controlled execution
3. Then **operate it over different architectures**

Storm Cluster

Master node

- Runs a daemon called *Nimbus*
- Responsible for
  - ✓ Distributing code around cluster
  - ✓ Assigning tasks to machines
  - ✓ Monitoring for failures of machines

Worker node

- Runs on a machine (server)
- Runs a daemon called *Supervisor*
- Listens for work assigned to its machines
- Runs “Executors”(which contain groups of tasks)

Zookeeper

- Coordinates Nimbus and Supervisors communication
- All state of Supervisor and Nimbus is kept here

Spark Streaming

micro-batch-processing tools

Framework for large scale stream processing

- Scales to 100s of nodes
- Can achieve 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.

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!

SPARK Streaming Requirements

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

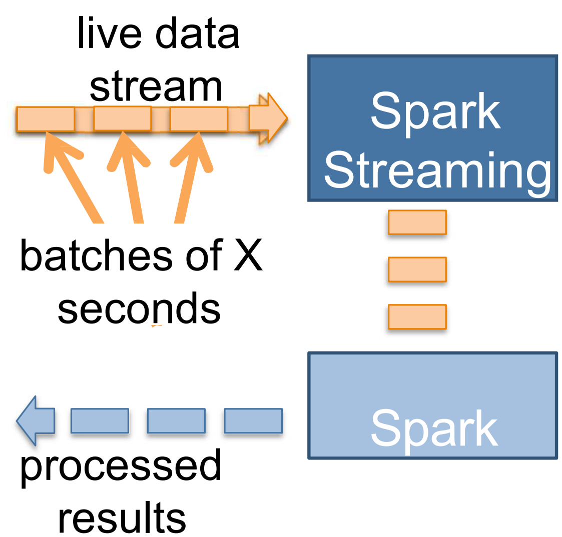
Spark Streaming

Spark Streaming: extension that allows to analyze streaming data

- ➢Ingested and analyzed in micro-batches

Uses a high-level abstraction called Dstream (discretized stream) which represents a continuous stream of data

- Divide live stream into batches of X seconds
- Spark treats each batch of data as RDDs
- Return results in batches, output can be persisted on the storage layer*

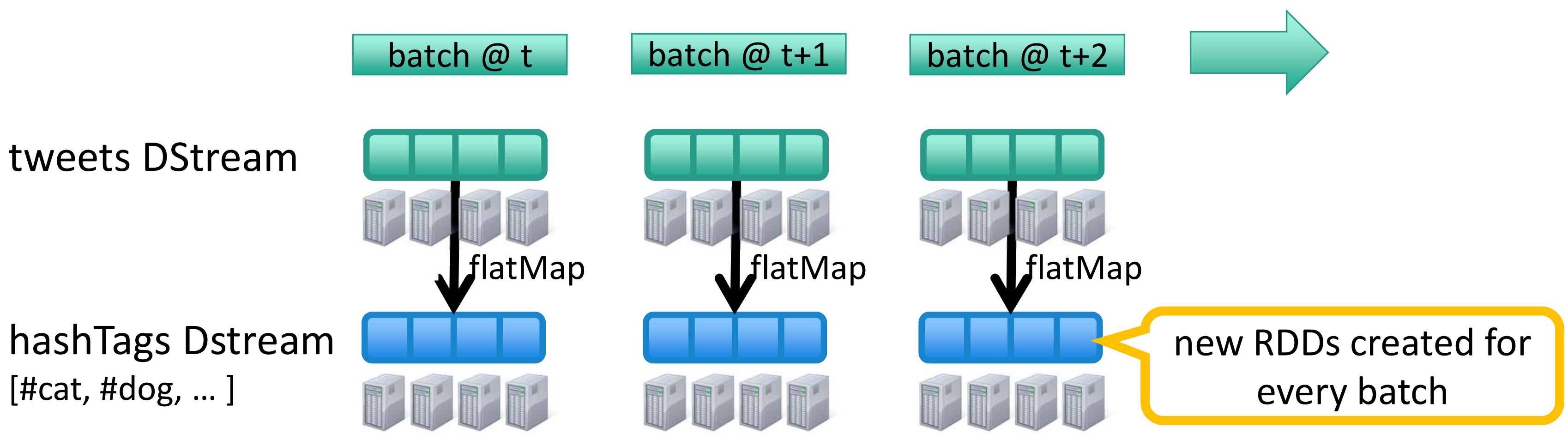


Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)

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

new DStream

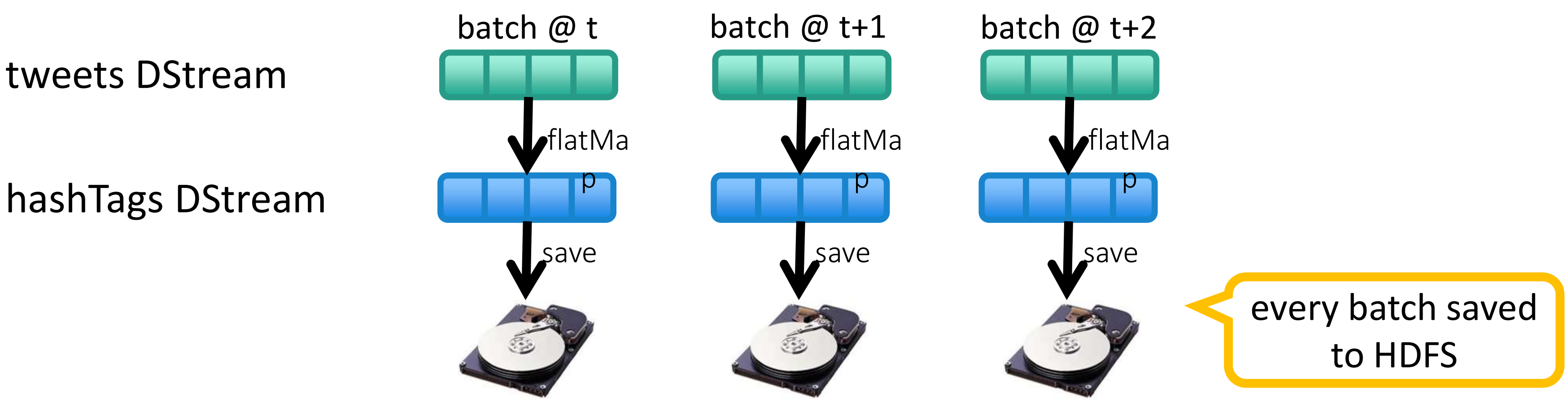
transformation: modify data in one Dstream to create another DStream



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)

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

hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage



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

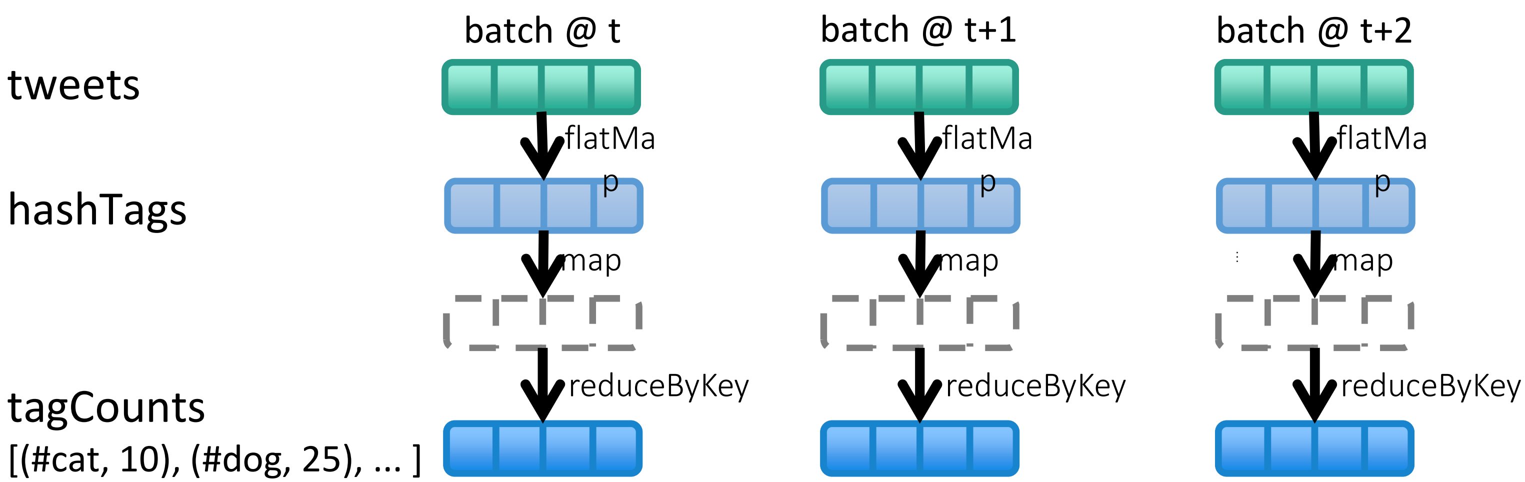
- Standard RDD operations – map, countByValue, reduce, join, …
- Stateful operations – window, countByValueAndWindow, …

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(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

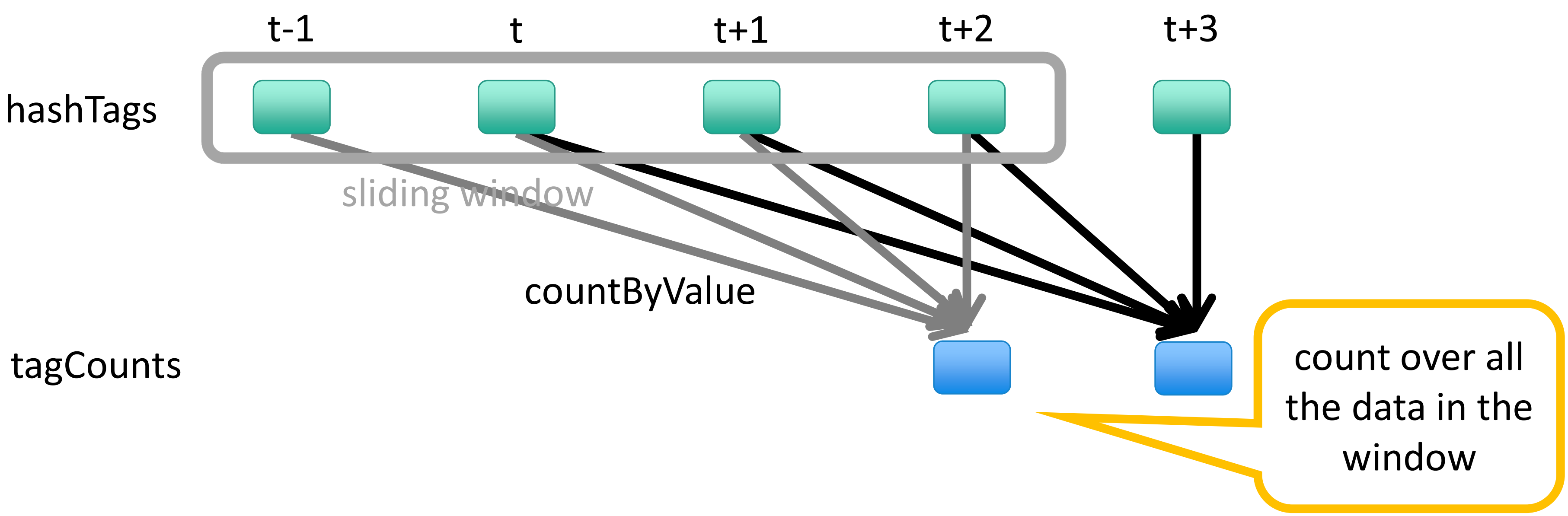
sliding window operation

window length

sliding interval

Example 3 – Count the hashtags over last 10 mins

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



Comparison with Storm and S4

Higher throughput than Storm

- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node