Big Data Storage and Processing
MSc in Data Analytics
CCT College Dublin

MapReduce Design Patterns
Week 3

## Lecturer: Dr. Muhammad Iqbal*
Email: miqbal@cct.ie

Agenda

- Concept of Design Patterns

- MapReduce Programming Model

- MapReduce Design Patterns for Big Data Processing

- Summarisation Patterns

- Filtering: Top Ten Pattern

- Examples for MapReduce using purchases.txt dataset

Design Patterns

Design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

Gang of Four

- Design patterns offer solutions to common application design problems.

- They provide generalized solutions in the form of boilerplates that can be applied to real-life problems.

- We can visualize design patterns using a class diagram, showing the behaviours and relations between classes.

Singleton
– instance: Singleton
– Singleton () + getInstance () : Singleton
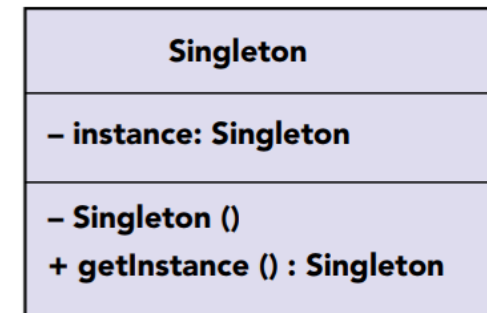
The singleton pattern class diagram

Design Patterns

How Patterns Were Discovered and Why We Need Them?

- Object-oriented programming emerged in the 1980s, and several languages that built on this new idea shortly followed. **For example**, Smalltalk, C++, and Objective C are still prevalent today.

- Design patterns have solved many issues that software engineers have with procedural programming languages like C and COBOL.

- Developers found common problems during the software development and then the best solution is to develop some pattern to solve similar kind of problems.

- When object-oriented programming emerged, it was still a pre-Internet world, and it was hard to share experiences with the masses.

MapReduce Design Pattern

- **MapReduce** is a computing paradigm for processing data specifically that resides on hundreds of computers, which has been popularized recently by Google, Hadoop, and many others.

- The paradigm is extraordinarily powerful, but it does not provide a general solution to what many are calling "big data".

- **What is a MapReduce design pattern?**

- It is a template for solving a common and general data manipulation problem with **MapReduce**.

- **MapReduce** is also called as a programming model that enables large volumes of data to be processed and generated by dividing work into independent tasks and executing the tasks in parallel across a cluster of machines.
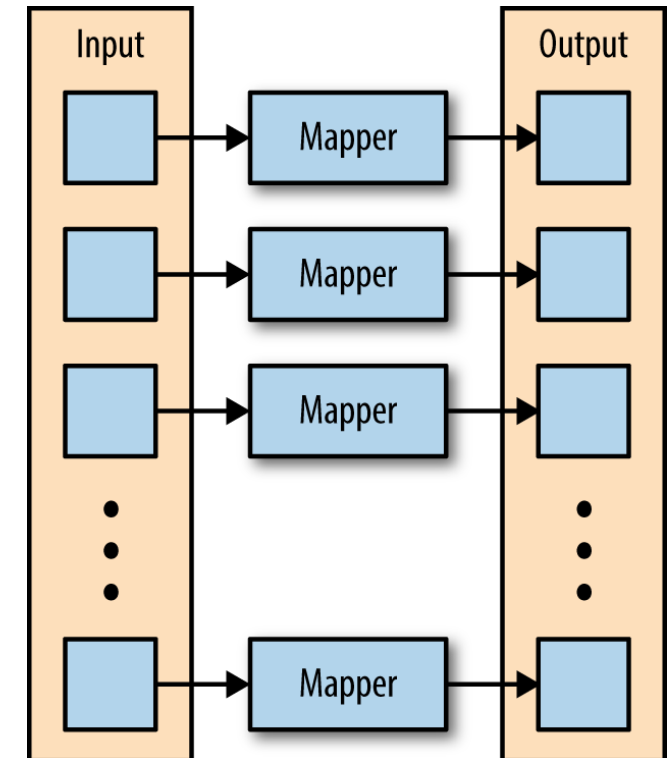
MapReduce with Python
Data Flow

- The **MapReduce** framework is composed of three major phases: **map, shuffle and sort, and reduce**. We describe each phase in detail.

- ## Map

- The first phase of a MapReduce application is the **map** phase. Within the **map** phase, a function (called the **mapper**) processes a series of **key-value pairs (<k, v>)**.

Data → Split → Map → Shuffling
 <k1,v1>   <k2,v2>   <k3,v3>

- The **mapper** sequentially processes each **key-value pair** individually, producing zero or more output key-value pairs.

MapReduce with Python
Data Flow

- Consider a mapper whose purpose is to transform sentences into words as an example.

- The input to this mapper would be strings that contain sentences, and the mapper's function would be to split the sentences into words and output the words.

- The input of the mapper is a string, and the function of the mapper is to split the input on spaces.

- The resulting output is the individual words from the mapper's input.
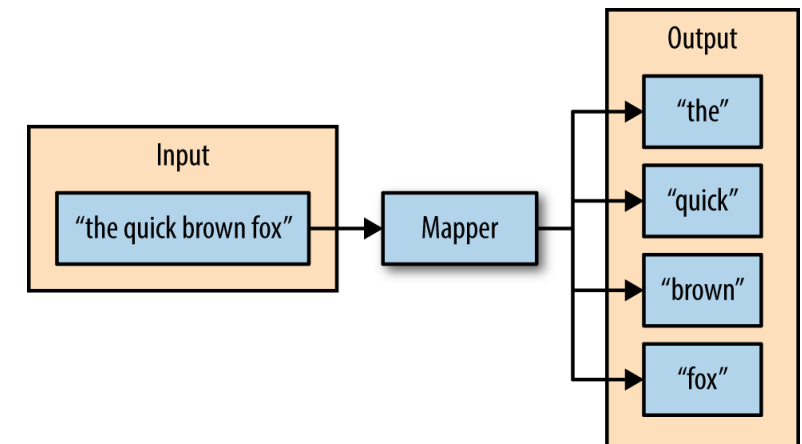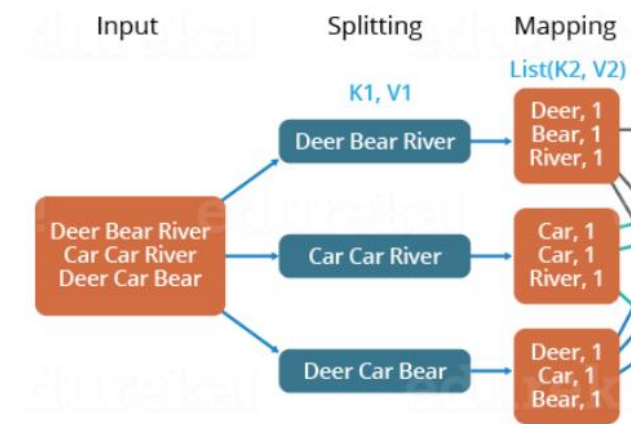


```
hduser@muhammad-VM:~/Desktop/Hadoop$ echo 'jack be nimble jack be quick' | ./mapper.py | sort -t 1 | ./reducer.py
be      2
jack    2
nimble  1
quick   1
hduser@muhammad-VM:~/Desktop/Hadoop$
```

Shuffle and Sort
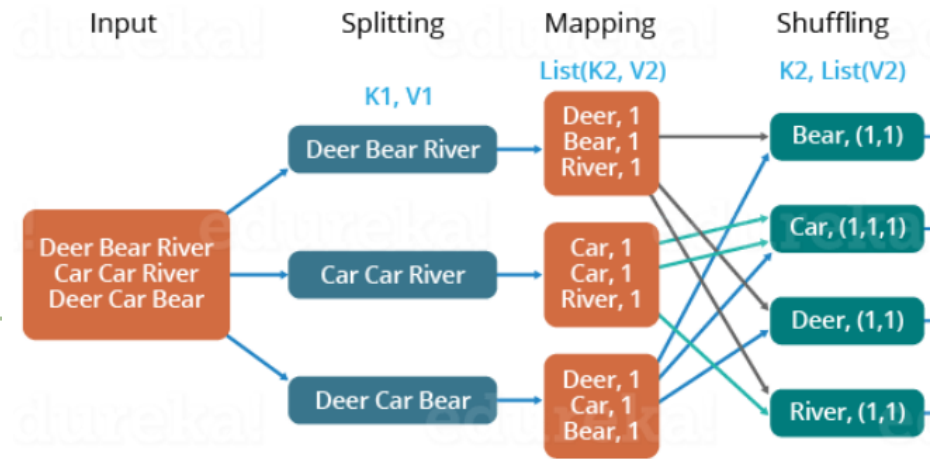
- The second phase of MapReduce is the **shuffle and sort**.

- **As the mappers begin completing, the intermediate outputs from the map phase are moved to the reducers. This process of moving output from the mappers to the reducers is known as shuffling.**

- **Shuffling** is handled by a *partition function*, known as the **partitioner**.

- The **partitioner** is given the mapper's output key and the number of reducers, and returns the index of the intended reducer.

- The final stage before the reducers start processing data is the sorting process. The intermediate keys and values for each partition are sorted by the Hadoop framework before being presented to the reducer.
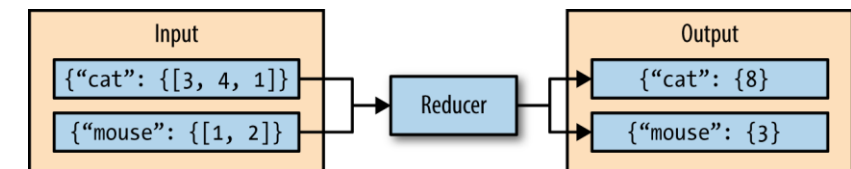
Reduce
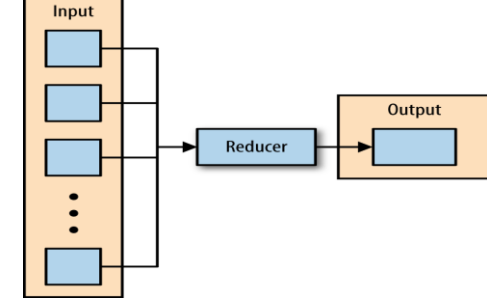
- The third phase of **MapReduce** is the **reduce** phase.

- The **reducer** aggregates the values for each unique key and produces zero or more output key-value pairs.

- Consider a reducer whose purpose is to sum all of the values for a key.

- The input to this reducer is an iterator of all of the values for a key, and the reducer sums all of the values.

- The reducer then outputs a **key-value pair** that contains the input key and the sum of the input key values.
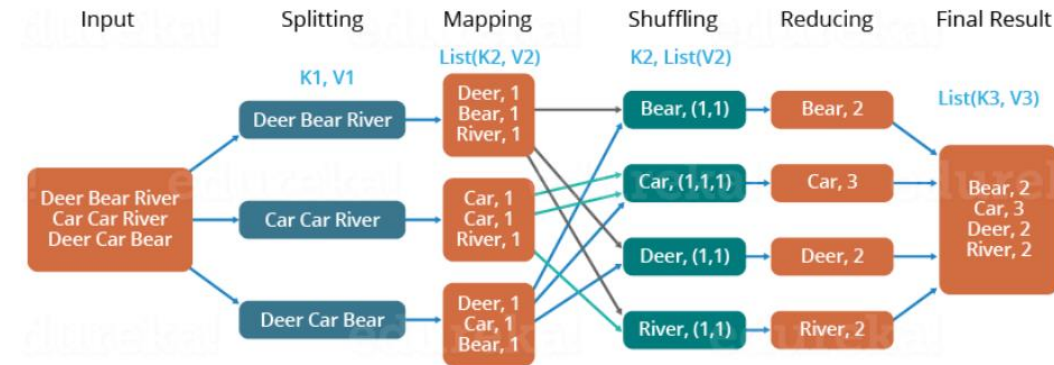
The reducer iterates over the input values, producing an output key-value pair.
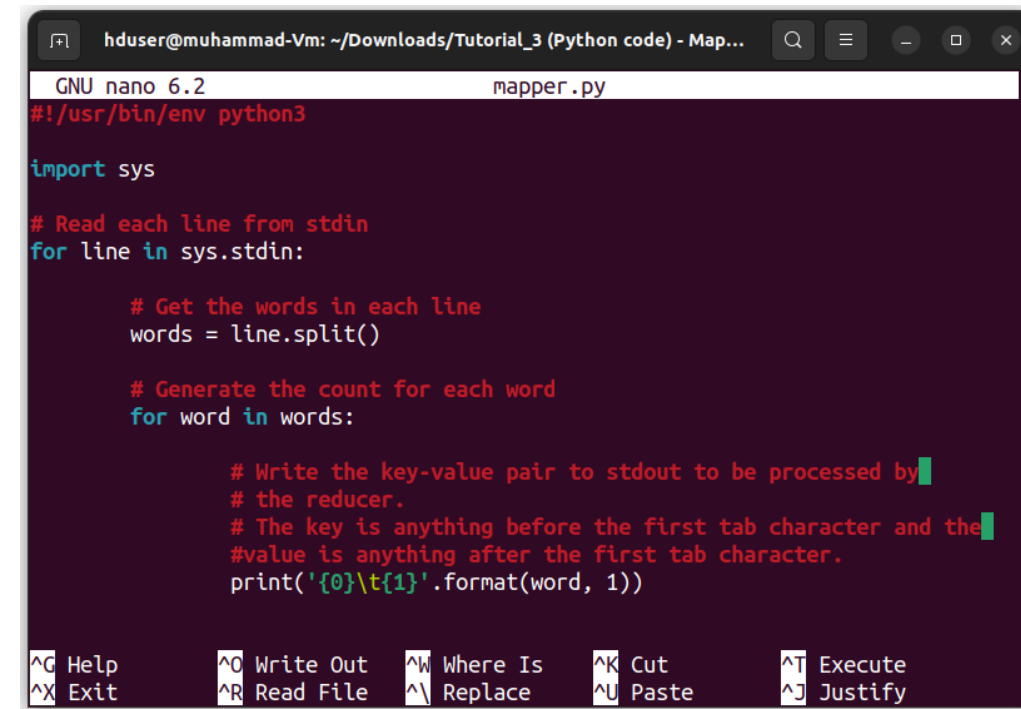

This reducer sums the values for the keys "cat" and "mouse".

Hadoop Streaming

- **Hadoop streaming** is a utility that comes packaged with the Hadoop distribution and allows **MapReduce** jobs to be created with any executable as the **mapper** and/or the **reducer**.

- The **Hadoop streaming** utility enables Python, shell scripts, or any other language to be used as a mapper, reducer, or both.

- ## How It Works

- The mapper and reducer are both executables that read input, line by line, from the standard input (**stdin**), and write output to the standard output (**stdout**).

- The **Hadoop streaming utility** creates a MapReduce job, submits the job to the cluster, and monitors its progress until it is complete.

Hadoop Streaming
A Python Example

- To demonstrate how the **Hadoop streaming utility** can run Python as a MapReduce application on a Hadoop cluster, the **WordCount** application can be implemented as two Python programs as **mapper.py** and **reducer.py**.

- **mapper.py** is the Python program that implements the logic in the **map** phase of **WordCount**.

- It reads data from stdin, splits the lines into words, and outputs each word with its intermediate count to stdout.

- The code in Example 3-1 implements the logic in **mapper.py**.


```
GNU nano 6.2                      mapper.py
#!/usr/bin/env python3

import sys

# Read each line from stdin
for line in sys.stdin:

        # Get the words in each line
        words = line.split()

        # Generate the count for each word
        for word in words:

                # Write the key-value pair to stdout to be processed by
                # the reducer.
                # The key is anything before the first tab character and the
                #value is anything after the first tab character.
                print('{0}\t{1}'.format(word, 1))
```
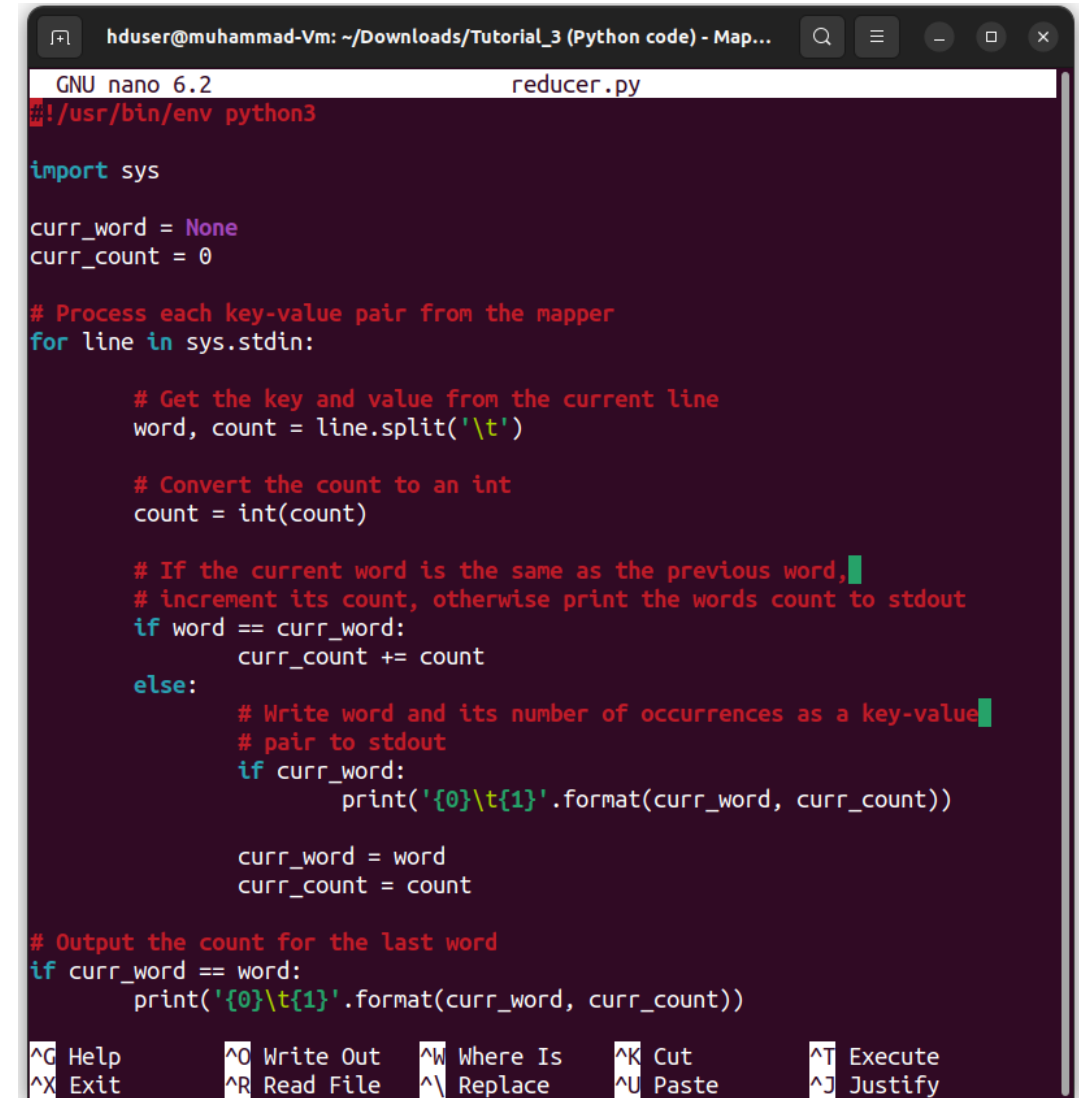
Example 3-1.
python/MapReduce/HadoopStreaming/**mapper.py**

A Python Example

- **reducer.py** is the Python program that implements the logic in the reduce phase of Wordcount.

- It reads the results of **mapper.py** from stdin, sums the occurrences of each word, and writes the result to **stdout**.

- The code in Example 3-2 implements the logic in **reducer.py**.

- Before attempting to execute the code, ensure that the **mapper.py** and **reducer.py** files have execution permission.

- The following command will enable this for both files as executables.

```
$chmod 700 mapper.py
$chmod 700 reducer.py
```



```python
#!/usr/bin/env python3

import sys

curr_word = None
curr_count = 0

# Process each key-value pair from the mapper
for line in sys.stdin:

        # Get the key and value from the current line
        word, count = line.split('\t')

        # Convert the count to an int
        count = int(count)

        # If the current word is the same as the previous word,
        # increment its count, otherwise print the words count to stdout
        if word == curr_word:
                curr_count += count
        else:
                # Write word and its number of occurrences as a key-value
                # pair to stdout
                if curr_word:
                        print('{0}\t{1}'.format(curr_word, curr_count))

                curr_word = word
                curr_count = count

# Output the count for the last word
if curr_word == word:
        print('{0}\t{1}'.format(curr_word, curr_count))
```

MapReduce Design Patterns

- **Categorization of Map-Reduce Patterns**

  – Summarization

  – Filtering

  – Join

  – Data Organization

  – Input/Output

MapReduce Design Patterns

- ## Summarisation Patterns

  - **Produce top-level and summarized view of data**

    - Calculation of aggregate values

      - Total amounts

      - Average amounts

      - Min/ Max values

  - *Specific Summarization Patterns*

    - Numerical Summarisations

    - Inverted Index

    - Counting with Counters

MapReduce Design Patterns
Numerical Summarisation

- **Pattern Description**

  - The numerical summarisations pattern is a general pattern for calculating aggregate statistical values over your data.

  - It is extremely important to use the combiner properly and to understand the calculation you are performing.

- **Intent**

  - Group records together by a key field and calculate a numerical aggregate per group to get a top-level view of the larger data set.

  - Examples for the numerical summarisation function include: minimum, maximum, average, median and standard deviation.

- **Motivation**

  - It is difficult to get the real meaning of the data manually because of the large quantities of data.

  - For example, noticing real usage patterns by reading through terabytes of log files with a text reader.

Summarisation Patterns

- **Word count**

  - **The "Hello World" of MapReduce.**

    - The application outputs each word of a document as the key and "1" as the value, thus grouping by words.

    - The reduce phase adds up the integers and outputs each unique word with the sum.

- **Record count**

  - A very common analytic to get a heartbeat of your data flow rate on a particular interval (weekly, daily, hourly, etc.).

Summarisation Patterns

- **Min/ Max/ Count**

  - An analytic to determine the minimum, maximum and count of a particular event.

    - Such as the first time a user posted, the last time a user posted, and the number of times they posted in between that time period.

    - You don't have to collect all three of these aggregates at the same time, or any of the other use cases listed here if you are only interested in one of them.

- **Average/ Median/ Standard Deviation**

  - Similar to Min/ Max/ Count, but not as straightforward of an implementation because these operations are not associative.

  - A combiner can be used for all three but requires a more complex approach than just reusing the reducer implementation.

Filtering Design Pattern
Top Ten

- The top ten pattern is a bit different than previous ones in that you know how many records you want to get in the end, no matter what the input size. In generic filtering, however, the amount of output depends on the data.

- ### Intent

- Retrieve a relatively small number of top **K records**, according to a ranking scheme in your data set, no matter how large the data.

- ### Motivation

- Finding outliers is an important part of data analysis because these records are typically the most interesting and unique pieces of data in the set.

- The point of this pattern is to find the best records for a specific criterion so that you can take a look at them and perhaps figure out what caused them to be so special.
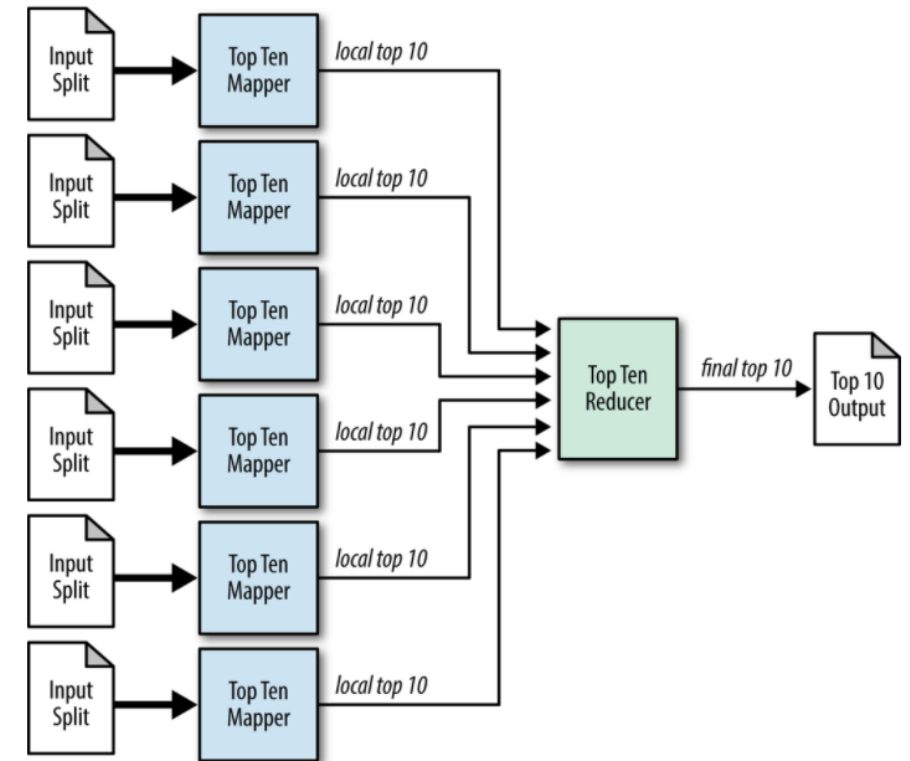
The structure of the top ten pattern

- **Use Cases**

  - Outlier analysis

  - Select interesting data

  - Catchy dashboards

18

Resources/ References

- Tom White, 2012, Hadoop The Definitive Guide, O'Reilly Publishing

- Hadoop with Python, Zach Radtka; Donald Miner, O'Reilly Media, Inc., 2015.

- Lublinsky B., Smith K. T. and Yakubovich A 2013, Professional Hadoop Solutions, Wrox [ISBN: 13:978-11186]

- Holmes A 2012, Hadoop in Practice, Manning Publications [ISBN: 13:978-16172]

- McKinney W. 2012, Python for Data Analysis, O'Reilly Media [ISBN: 13: 978-14493]

- https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

- Some images are used from Google search repository (https://www.google.ie/search) to enhance the level of learning.