# FIT5202 Big Data

**Borhan Kazimipour**

**FIT5202 Big Data**

Borhan Kazimipour

Generated by Alexandria (https://www.alexandriarepository.org) on August 1, 2017 at 12:07 pm AEST

# Contents

# 1
# Introduction to Big Data

Unit Learning Outcomes (1,5,6):

- compare the use of **data streaming** methods such as **sampling**, **sketching** and **hashing**;
- evaluate the suitability of different **distributed technologies** for big data processing;
- explain the workings of a typical **cloud computing** environment.

Module Learning Objectives:

- Demonstrate their high-level understanding of Big Data concepts, skills, and technologies.
- Explain the significant Vs of Big Data, MapReduce, and other fundamental *parallel patterns*.
- Explain the potentials and pitfalls for a business adopting Big Data.
- Interpret and develop basic Scala programs for Spark.

Assessments:

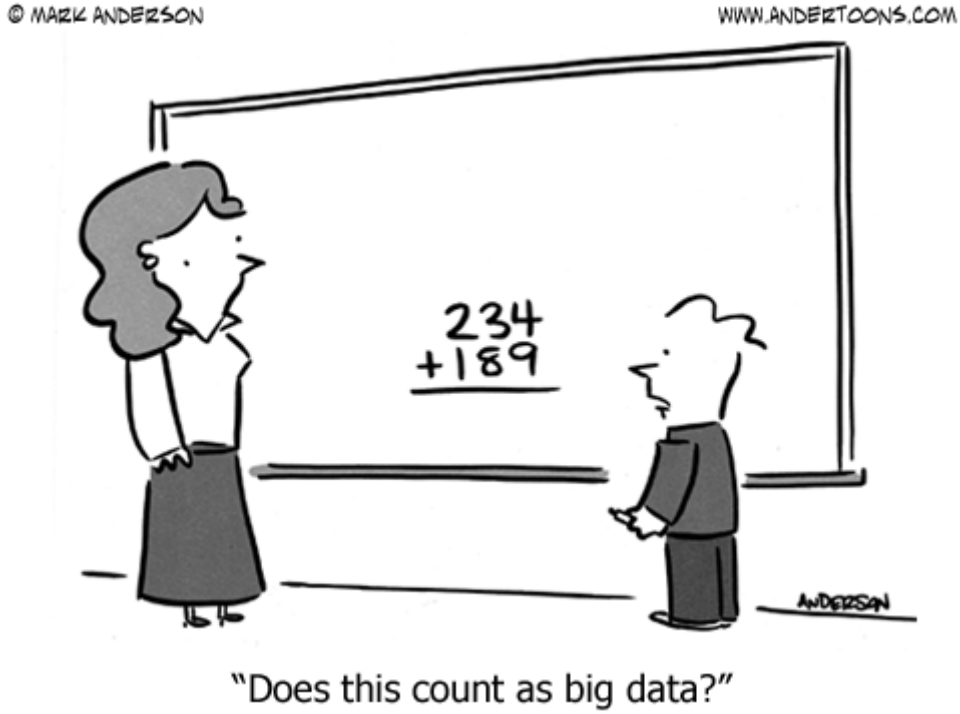
- Essay writing on a topic that covers Big Data concepts, characteristics, and impacts with some practical examples (UO/O 5,6)
- Develop/Complete/Interpret one or more Scala scripts (UO/O 1)

Resources:

- Chapter 1, Pries, Kim H. Big Data Analytics. Auerbach Publications, 20150205, CRC Press, Taylor & Francis 2015.
- Chapter 1, Mastering Scala Machine Learning - Alex Kozlov
- Bigdata teaching VM for labs and assignments

# 1.1
# Big Data

## Concepts, Skills, and Technologies.




*What is Big Data? (source: www.andertoons.com)*

We are definitely living in the golden era (http://www.forbes.com/sites/dorieclark/2013/08/08/four-things-you-need-to-know-in-the-big-data-era/#4e1c3e7f2e86) of Big Data, but, what is Big data (http://www.slideshare.net/BernardMarr/big-data-25-facts/3-Over_90_of_allthe_data)? Generally speaking, Big Data is a broad term that experts may define it in different languages. In the next part, we quote some of the possible definitions of Big Data.

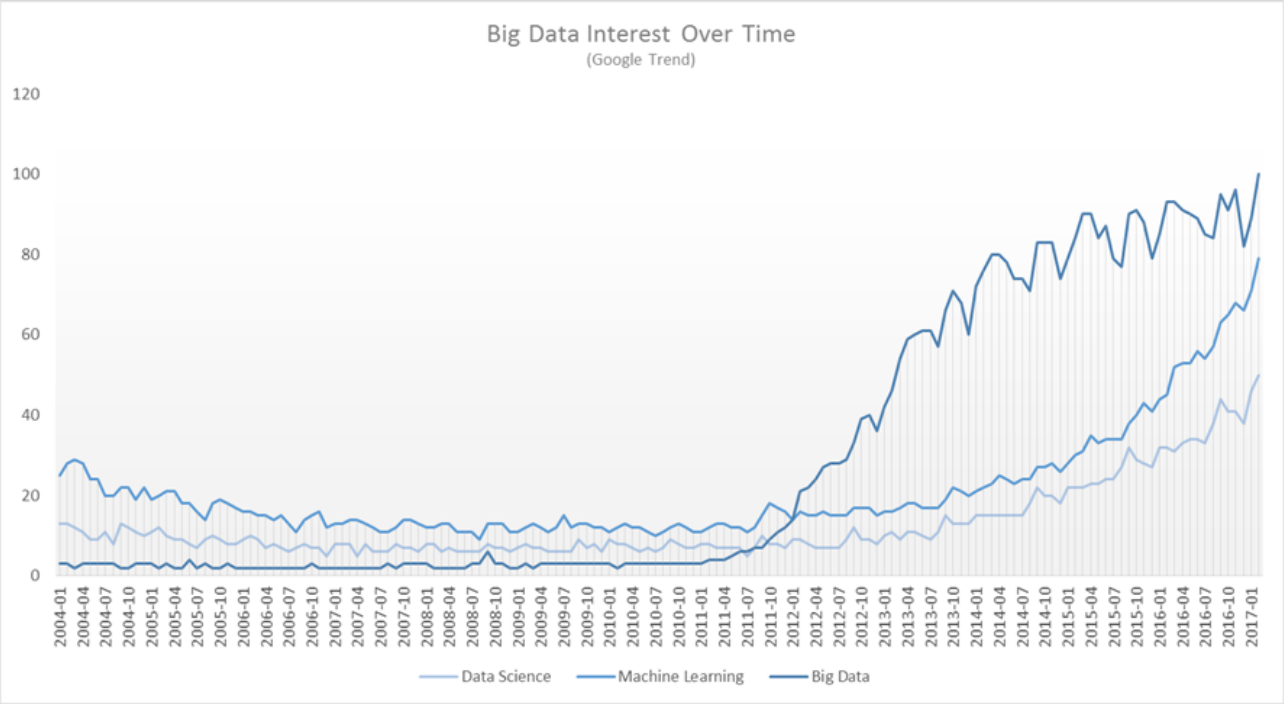
# Definitions

These are some definitions written well-known authors:

- Bernard Marr (http://www.forbes.com/sites/bernardmarr): Big Data can be defined as the digital trace that we are generating in this digital era.
- Gartner (http://What%20Is%20Big%20Data?%20-%20Gartner%20IT%20Glossary%20-%20Big%20Data): Big Data is high-volume, high-velocity, and high variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation.
- Ernst and Young (http://www.ey.com/au/en/home): Big Data refers to the dynamic, large and disparate volumes of data being created by people, tools, and machines. It requires new, innovative, and scalable technology to collect, host and analytically processes the vast amount of data gathered in order to drive real-time business insights that relate to consumers, risk, profit, performance, productivity management and enhanced shareholder value.
- Lisa Arthur (http://www.forbes.com/sites/lisaarthur): Big Data is a collection of data from traditional and digital sources inside and outside a company that represents a source of ongoing discovery and

analysis.

We may simply define Big data as datasets that grow so large that it is difficult to capture, store, manage, share, analyze and visualize with the typical software tools.




*Big Data Interest Over Time (source Google Trend)*

# Characteristics

As these definitions propose, there is no single widely accepted definition of Big Data. However, almost everyone agrees on the main characteristics of Big Data a.k.a. Big Data Vs:

**Velocity:** The speed at which data accumulates. We all know how data is being generated rapidly by processes that seem will never stop. This is the reason that real-time streaming and cloud-based services have been developing to a great extent to facilitate our computational need anytime anywhere quickly. To get a sense of the velocity of data we constantly generate, think about hours of videos that are uploaded to YouTube every minute (an estimate of 65,000 videos per day). We also create more than 10 Terabytes tweets and 15 Terabytes Facebook posts on a daily basis.

**Volume:** The amount of the data that is being generated, stored or processed for particular applications. By increasing the number of sources (e.g., mobile apps, online trading website, smart cars, etc) and improving the resolution of the sensors the quick growth in the data volume is inevitable. Luckily, most of the current infrastructures are scalable to facilitate these enormous of amount of data and data related services. In 2011, for example, it was estimated (http://www.computerworld.com/article/2513110/data-center/scientists-calculate-total-data-stored-to-date--295--exabytes.html) that total data stored to date was around 300 Exabytes (1 Exabyte = 1 billion Gigabytes = 245,146,535 DVDs) and growing with 23% rate every year. As another example, we know that more than 30 billion pieces of content shared on Facebook every month. This huge amount of data needs to be stored securely and efficiently in multiple places and processed in real-time 24/7.

**Variety:** The diversity of the data such as structured data that can be organized as rows and columns

and stored relational databases, and unstructured data such as social media contents, pictures, videos, and machine logs. It is estimated that 80% of the currently stored data is indeed unstructured data which are more difficult to analyze and visualize.

Some experts may add more Vs (https://upside.tdwi.org/articles/2017/02/08/10-vs-of-big-data.aspx) to this list, such as ***Veracity*** (as well as ***Viability*** *(https://www.wired.com/insights/2013/05/the-missing-vs-in-big-data-viability-and-value/)* *and* ***Validity***) which refers to the conformity to the accuracy, consistency, completeness, integrity, and ambiguity of data and data sources, ***Visualization*** of Big Data which is far more challenging (https://www.tableau.com/stories/workbook/get-value-data-tableaus-big-data-bi) than plotting standard two-dimensional bar chart $x$ and $y$ variables, ***Variability*** which refers to data whose statistics (https://en.wikipedia.org/wiki/Concept_drift) and meaning (http://nosql.mypopescu.com/post/6361838342/bigdata-volume-velocity-variability-variety) is constantly changing, ***Vulnerability*** which concerns (http://www.crn.com/slide-shows/security/300081491/the-10-biggest-data-breaches-of-2016-so-far.htm/pgno/0/5) about the Big Data breaches (http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/), ***Volatility*** which tries to determine how long a piece of data needs (and worth) to be kept in the data warehouses while the volume of the storage is increasing by a high velocity, and finally the ***Value*** (http://www.ibmbigdatahub.com/infographic/extracting-business-value-4-vs-big-data) which refers to our ability and demand to turn data into valuable insights, increase customers/employee/stakeholders satisfaction and generate more profit.

*Big Data Word Cloud*

# Applications

So far we discussed the Big Data as a broad concept. We also briefly explored some of the challenges in handling, processing, analyzing and visualizing the Big Data. The question arises here is that "What is the value of Big Data?" Or more precisely, "In exchange of all the challenges involved in the Big Data processing, what values it can bring to the organization businesses and people's lives?". In the following video, Susan Etlinger answers some of these questions.

(https://www.ted.com/talks/susan_etlinger_what_do_we_do_with_all_this_big_data)

The value and applications (https://www.atkearney.com/documents/10192/698536/Big+Data+and+the+Creative+Destruction+of+Todays+Business+Models.pdf/f05aed38-6c26-431d-8500-d75a2c384919) of Big Data are so many and broad that makes it very challenging to have a concise summary. However, let us short list some of the interesting applications (https://www.ap-institute.com/big-data-articles/how-is-big-data-used-in-practice-10-use-cases-everyone-should-read) of Big Data:

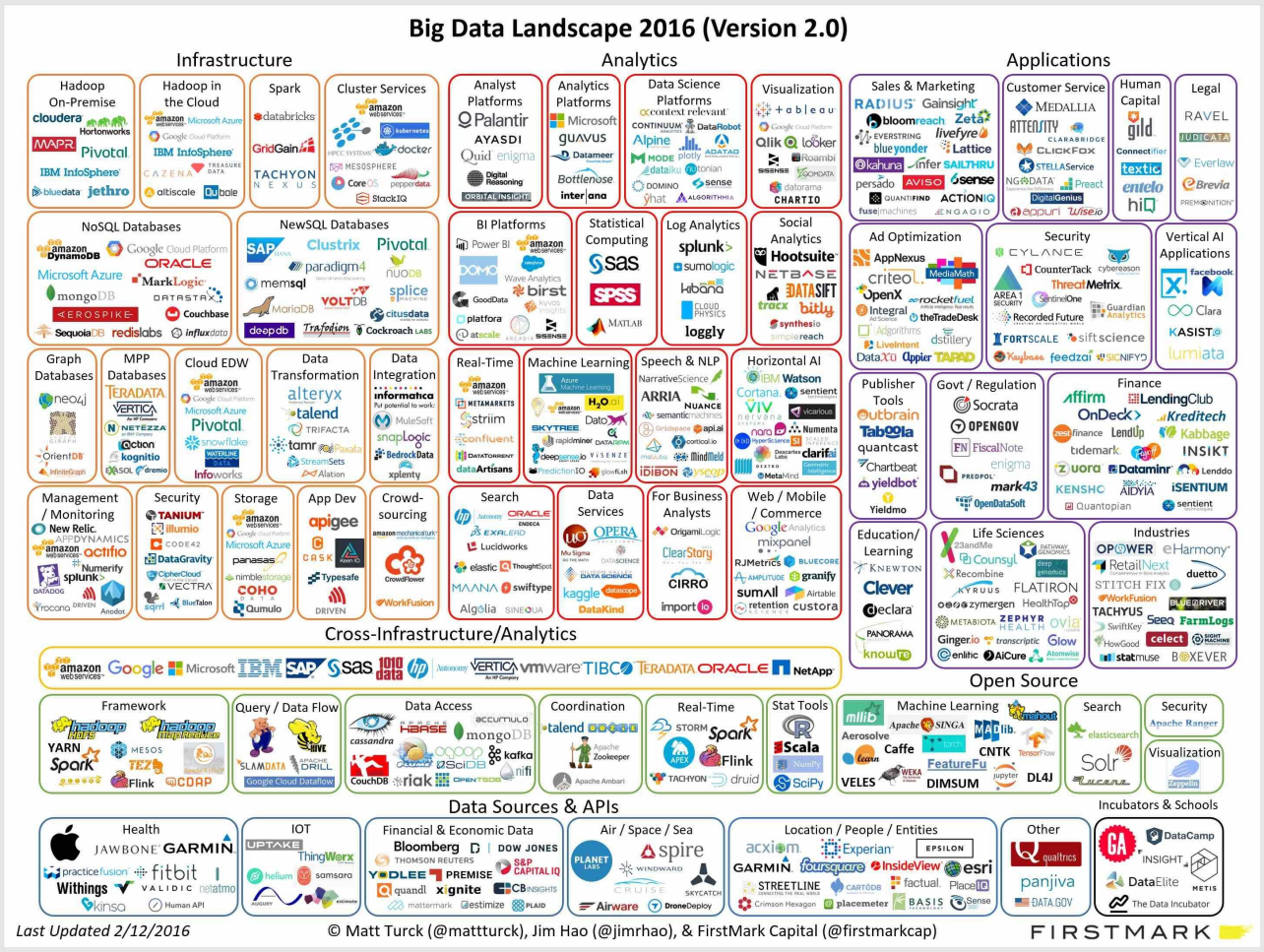
1. Saving lives: Early detection (https://hbr.org/2016/12/how-geisinger-health-system-uses-big-data-to-save-lives) of diseases.
2. Recommender systems/engines (http://www.forbes.com/sites/lutzfinger/2014/09/02/recommendation-engines-the-reason-why-we-love-big-data/#16036f65218e): Amazon, Google Adwords, and Spotify.
3. User (https://www.webtrends.com/blog/2016/05/applying-big-data-to-improve-customer-experience/) behavior analysis: Predicting the successfulness of House of Cards series even before filming by Netflix
4. Question answering and summarization (http://link.springer.com/chapter/10.1007%2F978-1-4939-2092-1_38#page-1): Apple Siri and OK Google!
5. Scientific research (http://insidebigdata.com/2015/07/07/case-studies-big-data-and-scientific-research/): up to 2 billions human genomes could be sequenced by the year 2025!
6. Internet of Things (IoT (https://www.sas.com/en_us/insights/articles/big-data/big-data-and-iot-two-sides-of-the-same-coin.html)): Reducing pollution and traffic congestion using traffic sensors, satellites, camera networks, smartphone GPS etc.
7. Saving wildlife (http://www.techrepublic.com/article/10-big-data-projects-that-could-help-save-the-planet/)

In the next two videos, some other interesting applications of Big Data analytics are explained: analyzing 5+ million books using Google Ngram (https://books.google.com/ngrams) and knowing our body with the data that it constantly generates.

(https://www.ted.com/talks/what_we_learned_from_5_million_books)

(https://www.ted.com/talks/talithia_williams_own_your_body_s_data)

# 1.2 Ecosystem




*Big Data Landscape (credit: Matt Truck)*

In the previous chapter, we broadly discussed the Big Data definitions, characteristics and applications. Now it is time to look a little bit deeper into the Big Data Ecosystem and the big players in this field. Generally speaking, we can categorize these species into four large categories: data sources, infrastructure, analytics, and skills.

## Data Sources

Nowadays we have access to a variety of structured and unstructured data. Besides the organization owned data, there are many publicly available and purchasable datasets and streams that one can utilize to address their specific applications. Here is a short list of many available options:

- Biometrics streams: e.g., Apple Watch, Fitbit.
- Financial and economic data: e.g., Bloomberg, Daw Jones, and Tomson Reuters.
- Social Media: e.g., Twitter, Facebook, Pinterest.
- Geolocation data: e.g., GPS navigation, telecommunication and sonar product (e.g., Garmin, Google Map, Vic Road).
- Others (http://www.datasciencecentral.com/profiles/blogs/20-free-big-data-sources-everyone-should-check-out): Quandle,

Government data, BOM.

> **Quick quiz:** Can you name some on-premise or on the cloud Big Data sources that you have worked (or interested to work) with?

# Infrastructures

## Hadoop

A great portion of Big Data infrastructure created based on Apache Hadoop (http://hadoop.apache.org/) platform. Hadoop (https://en.wikipedia.org/wiki/Apache_Hadoop) is an open-source software framework utilized for distributed storage and processing large amounts of data. The main objective of this framework is to breaking and distributing data into parts and analyzing those parts in parallel. Hadoop is implemented in several specialized modules such as storage (i.e., Hadoop Distributed File System (https://en.wikipedia.org/wiki/Apache_Hadoop#Hadoop_distributed_file_system) or HDFS), resource management (YARN (http://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/)), distributed processing programming models (MapReduce (https://en.wikipedia.org/wiki/MapReduce)), and software libraries. We will discuss Hadoop in details in the next module.

> **Quick quiz:** Hadoop, Spark and many other Big Data platforms are published as open source solutions. In your opinion, what are the advantages and disadvantages of open source technologies?

## NoSQL

Not Only SQL (https://en.wikipedia.org/wiki/NoSQL), or NoSQL for short, is involved in processing huge amount of multi-structured data. Some of NoSQL database management systems such Apache HBase (https://hbase.apache.org/) and Apache Accumulo (http://accumulo.apache.org/) are designed to harness the power of Hadoop. Apache Cassandra (http://cassandra.apache.org/) and MongoDB (https://www.mongodb.com/) are other famous NoSQL databases that widely used by practitioners.

> **Quick quiz:** What are the main shortcomings of the traditional relational databases that NoSQL can address?

## Massively Parallel Processing

Massively Parallel Processing (MPP) architecture may be used by SQL databases such as Microsoft Azure SQL (https://docs.microsoft.com/en-us/azure/sql-data-warehouse/sql-data-warehouse-overview-what-is) Data Warehouse, Teradata

(http://www.teradata.com.au) and Vertica (https://vertica.com/) to segment data across multiple nodes and process them in parallel. Note that the architecture of Hadoop databases and MPP databases can be very different (https://0x0fff.com/hadoop-vs-mpp/). In brief, MPP databases are usually faster than Hadoop databases, however, Hadoop-based databases scale better.

# Analytics

The number of specialized technologies that are designed to address advanced analytical tasks is not only large but also growing. The following is a short list of some categories of Big Data analytical tools:

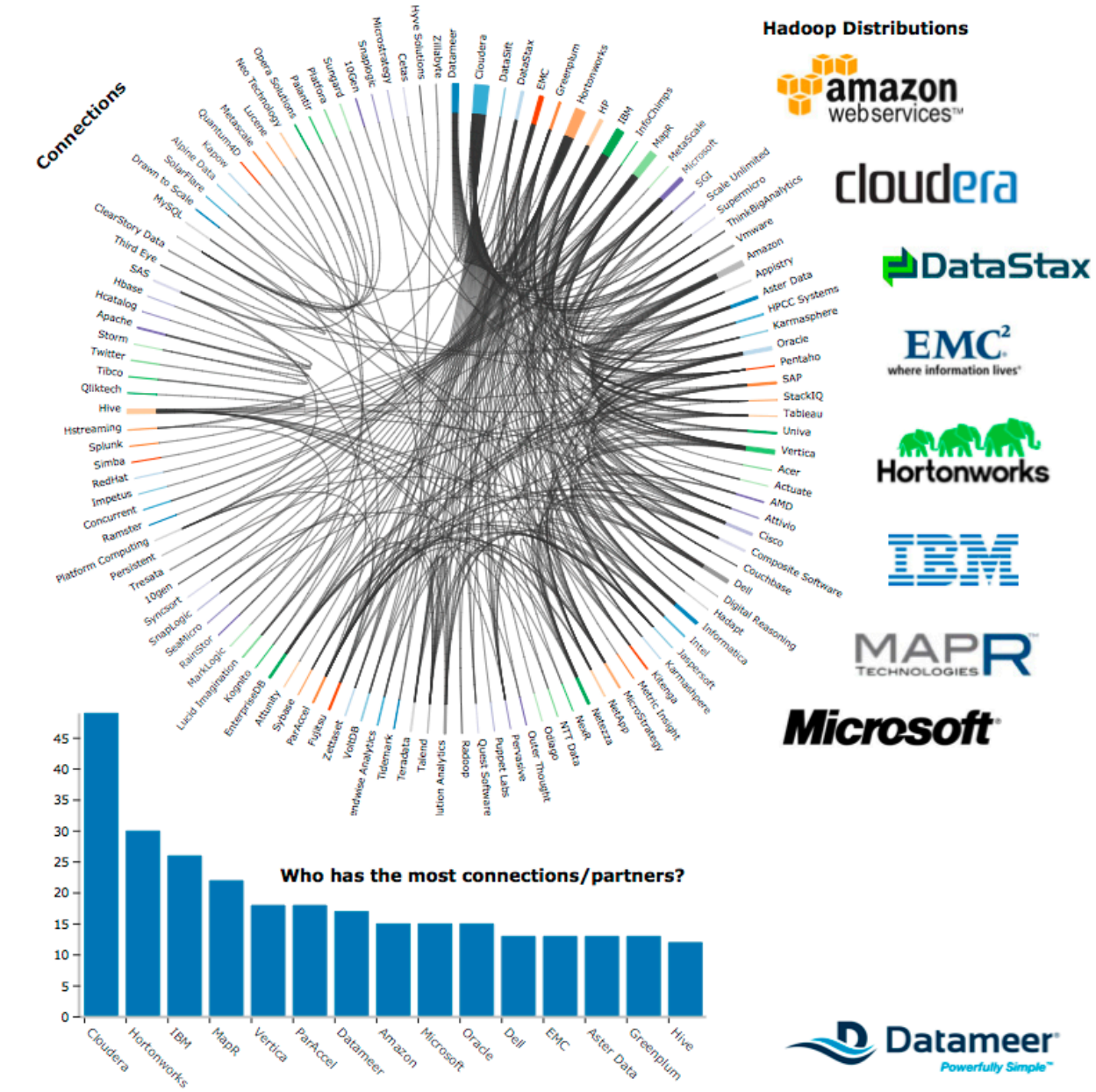
## Analytics

These tools aggregate, process and analyze data to uncover new insights that help organizations make better-informed decisions. The main objective of these platforms is to provide insightful analytics of a large volume of data in the shortest possible time window. IBM BigInsights (https://www.ibm.com/analytics/us/en/technology/biginsights/), HP HAVEn (http://www.havenondemand.com/), and SAP Big Data (http://www.sap.com/australia/solution/big-data.html) solutions are some examples of analytic platforms.

> **Quick quiz:** From your perspective, what are the main characteristics of a practical analytical platform? Can you identify a platform that has all these characteristics?

## Visualizations

The visualization platforms such as Tableau (https://www.tableau.com/) and Qlik (http://www.qlik.com/en-au) are utilized to acquire raw data from different sources and presenting it in multi-dimensional visual formats to illuminate the latent information. Producing interactive visualizations and integrating with other analytical platform is a strong trend for these technologies. For example, the following graphic illustrates some of the famous Hadoop distributions and the connections between them (as 2012).




*Hadoop Distributions (created by Datameer 2.0)*

**Quick quiz:** Name some impressive Big Data visualization you have seen recently. What factors do make them attractive?

## Business Intelligence (BI)

These platforms are used for integrating, analyzing and visualizing data specifically for businesses analytical purposes. The output format of these platforms is usually business reports and interactive dashboards. Microsoft Power BI (https://powerbi.microsoft.com/en-us/) and Sisense (https://www.sisense.com/)Solutions

are two of famous BI platforms.

> **Quick quiz:** What is the main difference between the BI products and other analytic platforms?

As the above figure indicates, there are many cross-platform products and services. Microsoft, Google, IBM, Talend and many others enterprises offer solutions that seamlessly integrate more than one platform.

While each end user organization may develop its own Big Data solution that solves its particular problem(s), there are some Big Data products that specialized to address certain applications. Fields in which applications are used include, but not limited to, marketing, finance, health, education, retail, energy, accommodation, and transportation.

> **Quick quiz:** Provide some examples of the Big Data applications that you are familiar with?

# Skills

A variety of skills is needed to maximize the benefits one can receive from Big Data. Indeed, the set of necessary skills is as wide as the tools and technologies that we are briefly introduced in the previous parts. Therefore, Big Data teams in large corporate usually consist of members from different backgrounds. Big Data engineers, developers, analysts, and scientists are some of the positions that one can take in such teams. Obviously, each of these positions needs a particular set of skills. Some of these skills such as critical thinking and problem-solving are general, whilst some others are very specific and change fast. Mastering Scala programming language or Spark development are a few examples of recently emerging but highly demanding skills.

The following is a brief list of further readings on this topic:

1. Top Ten Technology Jobs (https://www.forbes.com/sites/louiscolumbus/2016/12/03/robert-half-top-ten-technology-jobs-in-2017-data-scientists-big-data-expertise-in-high-demand/#4dce94af7714)
2. Big Data Industry Predictions for 2017 (https://insidebigdata.com/2016/12/21/big-data-industry-predictions-2017/)
3. Must-Have Skills to Land Top Big Data Jobs (https://www.datanami.com/2015/01/07/9-must-skills-land-top-big-data-jobs-2015/)

> **Quick quiz:** Considering the current highly demanding Big Data skills, what is your vision of the future of this industry? Which skills do you think would be needed more in the future and which ones will lose their positions?

# 1.3
# Potentials and Pitfalls

## Potentials

Just like any new technology, Big Data comes with lots of opportunities and challenges. In the previous chapters, we discussed some of the potential applications of Big Data in a variety of industries. The following Ted Talk by Kenneth Cukier provides more examples to emphasize the Big Data potentials and the necessity of using it in every organization.

(https://www.ted.com/talks/kenneth_cukier_big_data_is_better_data)

It is extremely important to understand that storing, processing and analysing Big Data have many potential benefits for an organization but also they are mandatory tasks that any business entity must accomplish in order to survive the modern era. In the following video, Philip Evans explains the importance of bringing Big Data on board for businesses of any kind and size.

(https://www.ted.com/talks/philip_evans_how_data_will_transform_business)

**Quick Quiz:** What other potentials can you name that are not covered here?

# Pitfalls

Managers and data teams should leverage the potentials that Big Data brings while approaching possible problems with caution. Companies may face different challenges working with Big Data, regarding their size, budget, and experience. The following is a short list of possible difficulties (https://blogs.wsj.com/experts/2014/03/26/six-challenges-of-big-data/) a company may need to deal with:

1. **Understanding your data:** Organizations may have trouble identifying the appropriate data (in terms of type, volume, and structure) and/or determining how to best use it. Constructing data-related business plans needs thinking outside of the box and choosing paths very different from the traditional approaches. In the following video, Ben Wellington provides some examples that show just owning some large amount of data does not necessarily result in the optimum solution.


(https://www.ted.com/talks/ben_wellington_how_we_found_the_worst_place_to_park_in_new_york_city_using_big_data)

2. **Lack of experts:** It is not always easy to find the right talent capable of both working with new technologies (such as those we introduced in the previous chapters) and of deciphering the data to find meaningful business insights.
3. **Building the right platform:** Constructing and maintaining proper platforms to aggregate, store, and manage the data across the enterprise is a challenging task. If an ill-conditioned platform is taken in place, not only prevents the organization to meet its business objectives but also waste a great deal of the company's time and budget.
4. **Rapid changes:** The Big Data technology landscape is evolving at extremely fast pace. This enforces the business entities to working with strong, reliable, and innovative technology partners that can assist them in creating the right architecture in an adaptive and efficient manner.
5. **Data ownership:** The ownership of data is fragmented across the organization. For example, each of IT, engineering, business, finance, and management departments may own part of the company's data. Without new ways of collaboration across all of these partial owners, an organization cannot leverage the opportunities Big Data brings in.
6. **Security and privacy:** Maintaining high standards of security and protecting people and organizations' privacy can be more challenging than ever. Processing a big amount of aggregated and integrated data may reveal personal information that assumed to be protected earlier. This can damage many parties, such as the data owners, their businesses, and customers. The following Ted Talk video explains the importance of privacy and how Big Data can breach it.

(https://www.ted.com/talks/glenn_greenwald_why_privacy_matters)

**Quick Quiz:** Are there any other pitfalls that you can discuss? How we can avoid, enhance or manage such disadvantages?

## More Readings

This is a short list of more readings about Big Data challenges in different industries:

- Big Data Challenges (http://www.dbjournal.ro/archive/13/13_4.pdf)
- Big Data Challenges and Pitfalls
  (http://docs.media.bitpipe.com/io_10x/io_107360/item_606277/OMGPHD_sDataManage_IO%23107360_E-Guide_120412.pdf)
- Potential and Pitfalls for Big Data in Health Research
  (http://www.advancesinanesthesia.com/article/S0737-6146(15)00007-6/pdf)
- Potentials and Pitfalls of Integrating Data From Diverse Sources
  (https://msu.edu/~hayesdan/PDF/BILD%20database%20paper%20in%20Fisheries.pdf)

# 1.4
# Big Data Programming Languages

## Why Scala?

Broadly speaking, many languages have been used for developing Big Data applications. Python (https://en.wikipedia.org/wiki/Python_(programming_language)), Java (https://en.wikipedia.org/wiki/Java_(programming_language)), and Scala (https://en.wikipedia.org/wiki/Scala_(programming_language)) are just a few examples to name. Each one of these languages has its own strengths and weaknesses. Among the available options, the winners are those that not only can scale-up, but also can be integrated with more Big Data technologies. For example, compatibility with Spark framework is a must. In the following, we discuss Spark programming language in more details.

> **Quick Quiz:** Can you name a number of Spark packages that are available in Scala but still missing in some other languages?

## Spark Programming Languages

As we briefly discussed, Apache Spark is a fast and scalable cluster computing engine that built on Java Virtual Machine (JVM). Spark supports Apache Hadoop MapReduce (MR) programming approach and integrates with Hadoop Distributed File System (HDFS) very well. Spark JVM can easily be interfaced with Java, Python, and Scala, where the latter is the Spark's native programming language. However, the Spark scope of use for these languages varies.

Python, which is extensively used in data science practices, requires translation by its interpreter and a socket connection to the JVM. This can potentially slow down the processing because of the overheads. Furthermore, not all Spark APIs may get implemented in Python (or at least not as soon as they become available for Scala and Java).

In contrast with Python, Java does not suffer from such overhead. However, Java is famous for being verbose. This means the practitioners may require more code writing effort to implement an application that they could develop in Python with a fraction of the time. Therefore, Java may not be the first choice as fast development is a concern of everyone. Note that fast development is translated to saving the budget for the organizations.

Scala (http://www.scala-lang.org/) which is an acronym for ***Scalable Language*** is specially designed to address the aforementioned limitations. Scala is the native Spark programming languages and ensures full Spark API support. Scala runs on JVM and seamlessly integrates with Java. From the coding effort (https://www.toptal.com/scala/why-should-i-learn-scala) point of view, Scala is less verbose than Java.


*Scala vs. Java (source: www.toptal.com/scala/why-should-i-learn-scala)*

**Quick Quiz:** When it comes to Big Data, which feature of a programming language is more important: fast development, scalability, ease of learning, CPU and memory performance? Or perhaps "*It depends*"?

In the next chapters we learn how to write simple to advance Scala scripts. In addition, there are many valuable documentations (http://docs.scala-lang.org/), tutorials (http://people.cs.ksu.edu/~schmidt/705a/Scala/scala_tutorial.pdf), videos (https://www.youtube.com/watch?v=DzFt0YkZo8M), books (https://www.amazon.com/s/ref=nb_sb_noss?url=search-alias%3Dstripbooks&field-keywords=scala), cheatsheets (http://alvinalexander.com/downloads/scala/Scala-Cheat-Sheet-devdaily.pdf), quick references (http://homepage.cs.uiowa.edu/~tinelli/classes/022/Fall13/Notes/scala-quick-reference.pdf), and free online courses (https://www.coursera.org/courses?languages=en&query=scala) are available for who wants to master this highly demanding programming language.

# 1.5
# Scala Programming

In this chapter, we briefly discuss Scala programming language. This introduction might not be very extensive, but big enough to provide the readers some anchors to start Scala programming. Note that although Scala is a fairly new programming language, many valuable documents and tutorials are freely available that one can use to learn and master it. Particularly, Python programmers may read this (https://wrobstory.gitbooks.io/python-to-scala/content/variables/README.html) or this (https://bugra.github.io/work/notes/2014-10-18/scala-basics-for-python-developers/) quick guides for learning Scala. Similar content is available (http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html) for Java developers.

Please note that the provided Scala programming guidelines are by no means complete. Based on your background and experience, you may find some parts very trivial or very advanced. In either case, there is nothing to worry about. As mentioned earlier, the Big Data skills are very broader than only learning a new programming language. Therefore, look at Scala as a tool that we use to practice a variety of Big Data skills. In other words, Scala programming exercises are providing us a path to achieve the learning objective. Mastering Scala, per se, is not a learning outcome!

# 1.5.1 Introduction to Scala

## Hello Scala!

First things first! Before start programming in Scala, we need to install (https://www.scala-lang.org/download/install.html) Scala on our machine or use a cloud-based Scala environment such as IBM Data Scientist Workbench (https://datascientistworkbench.com/) or Cloud x LAB virtual labs (https://cloudxlab.com/). For a complete environment, you may prefer to use one of the freely available virtual machines (VMs) such as Cloudera QuickStart (https://www.cloudera.com/downloads/quickstart_vms/5-10.html) or MapR Sandbox (https://mapr.com/products/mapr-sandbox-hadoop/). The main advantages of such VMs are that they provide a range of different services (in addition to Spark/Scala) and usually offer some free tutorials as well. The disadvantage, however, is that they usually use Hadoop distributions different than the original Apache's. Therefore, there might be some inconsistency.

For the very first examples, you can also use minimalistic Scala consoles such as Scala Tutorials (http://scalatutorials.com/tour/) or Scala Fiddle (http://scalafiddle.net/console). For your convenience, we developed a VM that you can use it either locally (by VMWare (https://www.vmware.com/au.html) or VirtualBox (https://www.virtualbox.org/wiki/Downloads)) or on the cloud.

# Hello Scala

There are different ways to execute a Scala code. In this section, we show how a simple "Hello World!" example can be run in different ways. For more information, refer to Scala official (http://www.scala-lang.org/documentation/getting-started.html) website.

## 1- Interactive Scala Programming

To run your first Scala line of code in an interactive environment, we should go to the correct path and run Scala Read-Eval-Print Loop (REPL) terminal. In the provided VMs, you can simply call `scala` in your home directory. When everything goes right, we should see a welcome message like:

```
Welcome to Scala version 2.11.6 (OpenJDK 64-Bit Server VM, Java 1.8.0_131).
Type in expressions to have them evaluated.
Type :help for more information.
```

Note that you may see different version numbers depending on the installed version of Scala and Java on your local or remote machine.

Let's do the classic Hello World example in REPL environment as:

```
println("Hello, Scala!");
```

and, obviously, the output is:

```
Hello, Scala!
```

We can continue writing more lines of code, or just `:q` or `:quit` to exit the REPL terminal.

## 2- Shell Script

To have a shell (e.g. bash) script that can execute Scala codes without running REPL explicitly, we can write our Scala script and save it to `.sh` file such as `HelloWolrd.sh`. Note that the first line should contain a proper preamble determining that which software must read and execute the following lines. In our case, Scala is the proper program.

To create and write our lines of code, we need a text editor. There is a variety of text editor with and without GUI. For example, you can choose between nano (https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/) and vi (https://www.cs.colostate.edu/helpdocs/vi.html) to create or edit text files. Here, we simply use `cat` command because we only need to create the `HelloWolrd.sh` file once (we do not edit it now). To do so, we can execute the followings

```
cat > HelloWorld.sh
```

In the above example, `HelloWorld.sh` is the name of the file we want to create and > symbol means we want to create a new file or override an existing one. To see the content of a text file, we should omit > symbol. Note that >> symbol append text to the end of an existing file instead of overriding it. After writing the above command and pressing Enter/Return key, we can write our scripts. Write the following (or copy and paste) and then press `ctrl-d` to exit the writing the environment.

```
#!/usr/bin/env scala
object HelloWorld extends App {
  println("Hello, Scala!")
}
HelloWorld.main(args)
```

Note that the first line explicitly says the codes are written in Scala. To execute the above script, we need to go to the right path, giving the file execution permission and call it. In this example, we created the file in the current directory, so there is no need to change the path. However, we most certainly need to give the file execution permission just like the following:

```
chmod 755 HelloWolrd.sh
```

Now, we can execute the code by:

```
./HelloWorld.sh
```

In this example, `./` simply points to the current directory, where we created the file. The output should be the same as the previous example. If you like, you can modify the code by an arbitrary editor. For example, run the following:

```
nano HelloWorld.sh
```

Then, do the modifications, press `ctrl-x` to exit and run the script once again (no need to change the permission again).

## 3- Compile and Run

In most cases, we compile our Scala programs to achieve the maximum efficiency. Assume the above Hello World example is stored in a `.scala` file such as `HelloWorld.scala` (you can follow the same

instruction as the previous example, but save the file with different postfix). Now, we can compile the code as simple as:

```
scalac HelloWorld.scala
```

The above command creates a file named as HelloWorld.class. To run the compiled codes, do the following:

```
scala HelloWorld
```

Note that we use `scalac` to compile the code and `scala` to run the compiled file. This short article (https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Scala.html) helps you master Hello World example!

# 1.5.2
# Data Types

## Basic Data Types

Scala supports various data types which are categorized into two main groups of built-in data types and developer's custom-built data types. Built-in data types are used to define primitive data such as Byte, Short, Int, Long for discrete numbering and Float or Double for values. It also supports Boolean, Char, String as well as special types such as Null and Anything (a complete list is available here (https://www.tutorialspoint.com/scala/scala_data_types.htm)). Note that all types start with capital letters and appear after the `:` symbol. It is possible to create and use variables without explicitly declaring their type. The following examples perform the same, although explicit type declaration is more preferable.

```
var x = 3.14
var y: Double = 3.14
```

In Scala, we can have both mutable (can be changed) and immutable (cannot change their values) objects. Can you guess which one of the following statement throws an error message?

```
var str1: String = "Hi!"
val str2: String = "Bye!"

str1 = "Hello!"
str2 = "Good Bye!"
```

You can also make your own custom-made data types in Scala with `class` keyword. In the following example, we learn more about this topic.

## Extended Data Types

### Point Example

The following code makes a class for defining a point data type with x and y as coordinates:

```
class point(val x:Int, val y:Int)
{
  def move(dx: Int, dy: Int): point = {
    return new point(x + dx, y + dy)
  }
  override def toString(): String = "(" + x.toString() + "," + y.toString() +
")"
}
```

The above class has two properties to store x and y values. The method `move` is used to move the current point to a new position, and the `toString` method is the String representative for the class. Note that in Scala, all classes inherit from the base class `Object`, which has few methods including `toString`. To

modify the base class method, the keyword `override` must be used.

Scala language evaluates the last line of a function as the return statement of that function, which makes the `return` keyword optional if the last line of the function holds the return value. The data type of the function could implicitly be inferred from the last line. Parenthesis for functions without an argument could be omitted, for instance, in the above class `x.toString()` can be safely replaced with `x.toString`.

The above class is an ***immutable*** class, which means that the value of the class cannot be changed when the class gets instantiated. Replacing `val` keywords with `var` keyword converts the above class to be ***mutable***. However, in Scala using the mutable data types are discouraged, since they are harder to be used in parallel computing for fast executions.

Let us update the `point` class considering the above notes to make it more concise:

```
class point(val x:Int, val y:Int){
  def move(dx: Int, dy: Int) = new point(x+dx,y+dy)
  override def toString= "(" + x.toString + "," + y.toString + ")"
}
```

> **Quick Quiz:** Can you spot the differences between the above examples?

Let's put the point class into action. For example, we can simply define a point e.g., `(1,2)` , then move it to `(3,4)` and finally print them using `println()` function.

```
// define point class
class point(val x:Int, val y:Int){
  def move(dx: Int, dy: Int) = new point(x+dx,y+dy)
  override def toString= "(" + x.toString + "," + y.toString + ")"
}
val pnt1 = new point(1,2) // create a point at x=1 and y=2
println(pnt1) //print the point
val pnt2 = pnt1.move(2,2) // move pnt1 2 units up and right
println(pnt2) // print the second point
println(pnt1,pnt2) // print both points
```

The output would be something like:

```
scala> class point(val x:Int, val y:Int){
     |   def move(dx: Int, dy: Int) = new point(x+dx,y+dy)
     |   override def toString= "(" + x.toString + "," + y.toString + ")"
     | }
defined class point

scala> val pnt1 = new point(1,2) // create a point at x=1 and y=2
pnt1: point = (1,2)

scala> println(pnt1) //print the point
(1,2)
```

```
scala> val pnt2 = pnt1.move(2,2) // move pnt1 2 units up and right
pnt2: point = (3,4)

scala> println(pnt2) // print the second point
(3,4)

scala> println(pnt1,pnt2) // print both points
((1,2),(3,4))
```

> **Quick Quiz:** Can you explain why `(1,2),(3,4)` in the last line wrapped in extra parenthesis?

## Vector Example

Let us expand the above data type examples using a `vector` class. Vectors, by default, start from `(0,0)` and end at an arbitrary point. Our vector data type supports two operators of `dot` and `cross` products. Let us first implement the `vector` class.

```
class vector(end_point:point)
{
  override def toString: String = "< (0,0)" + "," + end_point.toString + ">"
}
```

Note that the input argument of the `vector` class is `point` that we already defined it in the previous example.

Assume $V_1$ and $V_2$ are two 2-Dimensional vectors. The dot product and cross products for the vector are defined as follows:

$$\text{Dot product: } V_1 . V_2 = (V_{1.x} V_{2.x}) + (V_{1.y} V_{2.y})$$
$$\text{1-D Cross product: } V_1 \times V_2 = (V_{1.x} V_{2.y}) - (V_{1.y} V_{2.x})$$
$$\text{2-D Cross product: } V_1 \times V_2 = ((V_{1.x} V_{2.y}),\quad (V_{1.y} V_{2.x}))$$

Note that cross product is only well-defined in 3-Dimensional spaces. In the above equations, we defined two analogs variations of cross product in 1 and 2-Dimenions. Now try adding three methods to implement these operators.

```
class vector(val end_point:point) {
  override def toString: String = "< (0,0)" + "," + end_point.toString + ">"
  def dot(that: vector): Int = this.end_point.x * that.end_point.x +
this.end_point.y * that.end_point.y
  def cross1(that: vector): Int = this.end_point.x * that.end_point.y -
this.end_point.y * that.end_point.x
  def cross2(that: vector): vector = new vector(new point(this.end_point.x *
that.end_point.y, this.end_point.y * that.end_point.x))
```

```
}
```

> **Quick Quiz:** Can you expand the vector class definition and its methods to implement a complete cross product in 3-Dimensional space?

Action time! Let's create two points, then two vectors and finally calculate their dot and cross products:

```
// define point class
class point(val x:Int, val y:Int){
  def move(dx: Int, dy: Int) = new point(x+dx,y+dy)
  override def toString= "(" + x.toString + "," + y.toString + ")"
}
//define vector class
class vector(val end_point:point) {
  override def toString: String = "< (0,0)" + "," + end_point.toString + ">"
  def dot(that: vector): Int = this.end_point.x * that.end_point.x +
this.end_point.y * that.end_point.y
  def cross1(that: vector): Int = this.end_point.x * that.end_point.y -
this.end_point.y * that.end_point.x
  def cross2(that: vector): vector = new vector(new point(this.end_point.x *
that.end_point.y, this.end_point.y * that.end_point.x))
}
// create point (3,4), then move it 2 units down and left
val pnt1 = new point(3,4)
val pnt2 = pnt1.move(-2,-2)
// print them:
println(pnt1,pnt2)
// create two vectors
val vec1 = new vector(pnt1)
val vec2 = new vector(pnt2)
// calculate their dot product
println(vec1.dot(vec2))
// calculate their 1-D cross product
println(vec1.cross1(vec2))
// calculate their 2-D cross product
println(vec1.cross2(vec2))
```

The output is:

```
scala> // define point class

scala> class point(val x:Int, val y:Int){
     |   def move(dx: Int, dy: Int) = new point(x+dx,y+dy)
     |   override def toString= "(" + x.toString + "," + y.toString + ")"
     | }
defined class point

scala> //define vector class

scala> class vector(val end_point:point) {
     |   override def toString: String = "< (0,0)" + "," + end_point.toString +
```

```
">"
     |   def dot(that: vector): Int = this.end_point.x * that.end_point.x +
this.end_point.y * that.end_point.y
     |   def cross1(that: vector): Int = this.end_point.x * that.end_point.y -
this.end_point.y * that.end_point.x
     |   def cross2(that: vector): vector = new vector(new
point(this.end_point.x * that.end_point.y, this.end_point.y * that.end_point.x))
     | }
defined class vector

scala> // create point (3,4), then move it 2 units down and left

scala> val pnt1 = new point(3,4)
pnt1: point = (3,4)

scala> val pnt2 = pnt1.move(-2,-2)
pnt2: point = (1,2)

scala> // print them:

scala> println(pnt1,pnt2)
((3,4),(1,2))

scala> // create two vectors

scala> val vec1 = new vector(pnt1)
vec1: vector = < (0,0),(3,4)>

scala> val vec2 = new vector(pnt2)
vec2: vector = < (0,0),(1,2)>

scala> // calculate their dot product

scala> println(vec1.dot(vec2))
11

scala> // calculate their 1-D cross product

scala> println(vec1.cross1(vec2))
2

scala> // calculate their 2-D cross product

scala> println(vec1.cross2(vec2))
< (0,0),(6,4)>

scala>
```

**Quick Quiz:** What would be the output of the dot and cross products if we change the order of `vec1` and `vec2` to in the last three statements? Verify your answer!

# 1.5.3
# Data Structure

In this part, we practice some of the basic data structures that are available in Scala. Using these structures, one can create more advanced data types that suit its needs. Let's start with the collections (https://en.wikipedia.org/wiki/Collection_(abstract_data_type)).

## Scala Collections

Scala supports a variety of data collections such as arrays, sets, and lists.

### Array

An array (http://docs.scala-lang.org/overviews/collections/arrays.html) in Scala is a collection of values under one name. The order of the array elements is preserved by Scala. Like most other programming languages, Scala arrays can contain duplicates. Also, note that arrays are mutable, so we can easily change their elements. Arrays' indices start from zero, and all elements must have the same type. We can access an element by mentioning its index in parenthesis. Declaring the type of an array's elements is optional. Let's create a few arrays:

```
// Create and array of strings with length 10
var anArray : Array[String] = new Array[String](10)
// Does the same thing:
var anotherArray  = new Array[String](10)

// Create an array of Int by initialization:
var someNumbers = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

// And now, a two dimensional array of size 10X5:
import Array._
val nRow = 10
val nCol = 5
var myMatrix = ofDim[Int](nRow,nCol)
```

There are many built-in functions we can use to create or modify arrays. Some of these methods (http://www.scala-lang.org/api/2.12.1/scala/Array%24.html) are: `concat()`, `empty()`, `fill()`, `range()` and `repeat()`.

### List

List (http://docs.scala-lang.org/overviews/collections/concrete-immutable-collection-classes.html#lists) and arrays are very similar in Scala. The first difference between these two data structures is that lists, as opposed to arrays, are immutable. Therefore, we cannot change their elements. Another difference between lists and arrays in Scala is that list are implemented as ***linked lists*** (https://en.wikipedia.org/wiki/Linked_list) whilst arrays are ***flat arrays*** (read more (http://freefeast.info/difference-between/difference-between-array-and-linked-list-array-vs-linked-list/)). Because of their special implementation, we can do some operations on lists that are not possible (technically

possible but not as efficient as list) to perform. For example, we can access the first element of a list using `head()` and all other elements using `tail()`. Note that tail() output itself is a list. Finally, isEmpty() method returns true if the input argument is an empty list (with zero length). With the help of these methods, we can easily implement recursive functions on any Scala list.

A simple way to create a list is to substitute the `Array` keyword in the above example with `List`:

```
// Create an empty String List
var aList : List[String] = List[String]()
// Does the same thing:
var anotherList  = List[String]()

// Create an list of Int by initialization:
var someNumbers = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

However, because of lists' special implementation in Scala, we can create them in a recursive way. Before that, we should recall that lists in Scala, pretty much like many other languages that support functional programming, are made of two parts: head and tail that are connected together using `cons` or `::`. Since an empty list in Scala is called ***Nil***, we can generate the above `someNumbers` list as:

```
// Recursively create a list of numbers:
var someNumbers = 1 :: (2 :: (3 :: (4 :: (5 :: (1 :: (2 :: (3 :: (4 :: (5 ::
Nil)))))))))
```

> **Quick quiz:** Can you guess how we can recursively create a two-dimensional list?

Scala lists support many built-in operators and [methods](https://www.scala-lang.org/api/current/scala/collection/immutable/List.html) (https://www.scala-lang.org/api/current/scala/collection/immutable/List.html) such as: `+`, `::`, `:::`, `contains()`, `drop()`, `filter()`, `foreach()`, `last`, `min`, `max`, and `reverse`.

## Set

Unlike arrays and lists, Scala [sets](http://docs.scala-lang.org/overviews/collections/sets) (http://docs.scala-lang.org/overviews/collections/sets) do not preserve the order of their elements. Also, sets cannot have duplicate elements (recall in the above examples, `someNumbers` list and array contain repeated values). The methods head(), tail(), and isEmpty can be applied on sets as well. In addition, there are a collection of operators that are specially designed to work with sets. For example, + adds an element to the set, - excludes an element from set, ++ concatenates two sets, & returns the intersection of two sets (common elements), and &~ returns the differences of two sets (uncommon elements).

> **Quick quiz:** What are the values of the set3?
>
> ```
> val set1 = Set(5,10,15,20,25,30)
> val set2 = Set(0,10,20,30,40)
> val set3 = set1 ++ set2
> ```

## Tuple

A Tuple (http://docs.scala-lang.org/overviews/quasiquotes/expression-details.html#tuple) is nothing but a simple collection of elements with different types. It is a special type of Scala Collections that can contain various values with same or different types together. It differs from other kinds of Collections such as Arrays and Lists in that it can hold objects with different types. Those objects are also immutable. You may think of a Tuple as a bag in which you can put different items with different shapes and sizes but as a whole container at the end. For example:

```
// Create a tuple
val tuple1 = new Tuple2("port", 80)
```

Or its equivalent shorter version:

```
// Does the same thing
val tuple1 = ("port", 80)
```

In the above example, `tuple1` contains two elements where the first one is a `String` and the second one is an `Int`. The number 2 in `Tuple2` indicates the length of the tuple that we want to create. So, if you wanted to create a `Tuple` of three elements (with one extra String parameter at the end), you would use `new Tuple3("port", 80, "message")` instead. To access the $i^{th}$ element of a tuple, we can use `_i` method. For example, `tuple1._1` and `tuple1._2` in the above example will return `"port"` and `80`, respectively. Note that the tuple indices start from one.

## Map

A map (http://docs.scala-lang.org/overviews/collections/maps.html) data structure in Scala is a collection of key-value pairs (very similar to Python dictionary). Therefore, a value can be retrieved based on its key, not its index in the collection. For example:

```
// Create a color map
val colorMap = Map("red" -> "FF0000", "green" -> "00FF00", "blue" -> "0000FF")
```

In this example, the color names are the keys and the hexadecimal RGB strings are values. We could create the same map using the following statements:

```
// Another way to create a color map
var colorMap:Map[String,String] = Map()

colorMap += ("red" -> "FF0000")
colorMap += ("green" -> "00FF00")
colorMap += ("blue" -> "0000FF")
```

For more on related topics, please refer to the Scala documentation about options (https://www.scala-lang.org/api/current/scala/Option.html) and iterators (http://www.scala-lang.org/api/2.12.0/scala/collection/Iterator.html).

# 1.5.4 Functional Combinators

Scala provides a special feature that allows us to apply a function to all elements of a collection without an explicit loop. These functions are called combinators and can be applied on lists and maps. Let's take a quick look at some of the most widely used combinators.

## map

The `map` (as a combinator, not a collection) evaluates a function over each element of a list and returns a list (the results) with the same number of elements. Note that `Map` is the data collection and `map` is the combinator.

```
// Define a function that returns square of a number
def sqr(x: Int): Int = x * x

// Create a list of Int numbers
val numList = List.range(1, 10)

// Apply the function on all list members
numList.map(sqr)
```

## foreach

The `foreach` is very similar to `map` but returns no values! For example, the following returns nothing:

```
// Apply the function on all list members
numList.foreach(sqr)
```

Indeed, `foreach` is intended for side-effects only.

## filter

The `filter`, as its name implies, removes any elements where the passed function returns a `false`. Therefore, the result of filtering can be expected to have a smaller size. For example:

```
// Define a function
def isPositive(x: Int): Boolean = x > 0

// Create a list of Int numbers
val numList = List.range(-10, 10)

// Apply the function on all list members
numList.filter(isPositive)
```

## zip

THe `zip` is used to aggregate the contents of two input lists into a single output list of pairs (similar to `zip` method in Python).

```
List("A", "B", "C", "D").zip(List.range(1,5))
```

## partition

The `partition` splits a list based on where it falls with respect to a ***predicate*** function. A predicate function is an ordinary function that returns Boolean values (e.g. `isPositive` in one of the above examples). The following example, creates two lists, one for positive numbers and the other one for nonpositive values (negative and zero):

```
numList.partition(isPositive)
```

## find

The `find` returns the first element of a collection (e.g. List or Map) that matches a predicate function. For example, the following code returns 1:

```
numList.find(isPositive)
```

## drop and dropWhile

Both of these functions drop some elements. The `drop(i)` drops the first *i* elements while `dropWhile(condition)` drops the elements that match the condition. Note that, `dropWhile` is sequentially applied to each element, from the very first element. As soon as the condition is not satisfied, `dropWhile` stops checking the remaining elements. Therefore, if some other elements satisfy the condition, they will be kept and not dropped.

The following is an example of `drop(i)`:

```
// A list of 10 consecutive integers starting from 1 and ending at 10.
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
numbers: List[Int] = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Dropping the first 5 elements of the "numbers" list:
scala> numbers.drop(5)
res0: List[Int] = List(6, 7, 8, 9, 10)
```

In the above code, first, we created a list of 10 consecutive integer values from 1 to 10. Next, by issuing the `drop(5)`, the first 5 elements (integer values) of the `numbers` list were dropped!

Now, let's see an example of the `dropWhile(condition)`:

```
// A list of 10 consecutive integers starting from 1 and ending at 10.
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
// Dropping the "odd" elements of the "numbers" list:
scala> numbers.dropWhile(_ % 2 != 0)
res1: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)
```

In the above code, we try to apply the `dropWhile(condition)` on the created list of `10` integers to `dropWhile` odd numbers (`%` is the modulo operator). As you see in the output, `1` was dropped, but `3` was not! The reason for that is `3` is now shielded by `2` and the condition ends. So, we will have a list of `9` integers from `2` to `10` as shown in the result.

## reduceLeft and reduceRight

These functions also apply a given function to all elements of a list, but in pairs. For example, the following code adds the first two elements of the list, saves the result somewhere called accumulator, and then add the third number to the accumulator and stores the result into the accumulator. It continues adding all other numbers to the accumulator and updates it. Finally, it returns the accumulator value:

```
// Define a function
def sum (x:Int, y:Int): Int = x + y

// Apply the reduce function
numList.reduceLeft(sum)
```

In this particular example, reduceLeft and reduceRight provide the same output, although reduceRight starts from the last element and moves backward. Unlike this example, the direction of the reduction technique may change the final outcome in many cases.

## foldLeft and foldRight

These functions perform very similar to `reduceLeft` and `reduceRight`, unless they accept another input and use it as the initial value of the accumulator. Therefore, the following code returns a value which is `100` larger than the outcome of the previous example:

```
// Apply the fold function
numList.foldLeft(100)(sum)
```

The relationship of `foldLeft` to `foldRight` is the same as `reduceLeft` to `reduceRight`.

## flatten

The `flatten` collapses one level of nested structure. Therefore, it converts a N-order nested structure to (N-1)-order structure. The following shows a simple example of this:

```
// apply "flatten" on a List of Lists
scala> List(List(1, 2), List(3, 4)).flatten
// the result is just one outer List as the nested (internal) Lists have been
flattened.
res2: List[Int] = List(1, 2, 3, 4)
```

In the above code, the nested Lists of `List(1, 2)` and `List(3, 4)` are flatten into `1, 2, 3, 4` and then embodied into the outer `List` which results in `List(1, 2, 3, 4)`. As another example, please try the following and explain how it works:

```
List(numList, numList.map(sqr), List(numList.reduceRight(sum))).flatten
```

## flatMap

The `flatMap` is a combination of `flatten` and `map` combinator functions (https://twitter.github.io/scala_school/collections.html).

# 1.5.5
# Functional Programming

## Scala Functional Programming

Scala is a language that supports both object-oriented (https://en.wikipedia.org/wiki/Object-oriented_programming) and also functional (https://en.wikipedia.org/wiki/Functional_programming) style programming. In functional programming, the computation of software is implemented in a mathematical function where the execution of the function cannot change the state of data (data type in functional programming is constant or immutable). On the other hand, Object Oriented programming focuses more on the data rather than function. In this paradigm of programming, software is made of objects which hold data and also functions to transform the data.

Before going further, let us elaborate anonymous functions (https://en.wikipedia.org/wiki/Anonymous_function) which we will use extensively in the following parts of this activity.

### Anonymous Functions

Generally speaking, anonymous functions are functions that are defined without having a name. Recall that in Python anonymous functions are defined using the `lambda` keyword (and, anonymous functions are also called lambda functions). For example, we can have `lambda x: x * 2` to define an anonymous function that doubles the value of x.

We have anonymous functions in Scala as well. For example, we can rewrite the above lambda function in Scala as:

```
(x: Int) => (x * 2): Int
```

which takes an integer (x), multiplies it by 2 and returns another integer (determined by `:Int`). The above code can be written in a very shorter way:

```
x => x * 2
```

### A Simple Example

```
scala> val list = List.range(1,10) // create a list
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> list // print it
res6: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> val anotherList = list.map(x=>x*2) // double it!
anotherList: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18)

scala> anotherList // print the new list
res7: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

> **Quick Quiz:** What do you expect to get if you remove the `map` method and write `foreach` instead? Why?

## Set Example

Let us implement a set (https://en.wikipedia.org/wiki/Set_(mathematics)), which can hold only integers from zero to 100, and do some set operations all using the functional style of programming. An individual element of this set is made of function which retrieves an integer member as input and returns a function that returns *true* if the integer exists in the set. We also want to implement an auxiliary function named `contains` to check if the set contains a specific number. Our implementation is as follows:

```
// Set function type
type Set = Int => Boolean
// Set with only one member
def singleton_set(member: Int): Set = (x: Int) => x == member
// check if set contain an element
def contains(s: Set, elem: Int): Boolean = s(elem)
```

There are two concepts that make Scala collections interesting: type parameterization and higher-order functions. Type parameterization (see `type` in the above code) allows us to create types that take another type as a parameter (such as `Set` type that takes `Int` type). Higher-order functions help us create functions that take other functions as parameters. In this example, `Set` is a function which accepts an integer and returns `Boolean` which is defined as `Int => Boolean`. The `singleton_set` that we defined here represents a set of length one. It accepts an `Integer` and returns a `Boolean` (because we expect a Set to perform similarly). In the above example, `(x: Int) => x == member` is nothing but an anonymous function that returns `true` value only if `x` and `member` store the same value. Note that none of the defined functions has an explicit `return` statement, because the last line of each of them will return the expected values.

Now let us test our singleton set:

```
// A set with only one member
val number = 1
val s1 = singleton_set(number)

// check set if contain added number
if (contains(s1, number)){
 println("set contains the number "+ number.toString)
} else {
 println("set doesn't contain the number")}
```

Currently, our set only can hold one member. Let us remove this limitation by defining the union function:

```
def union(s: Set, t: Set): Set = (x: Int) => s(x) || t(x)
```

Now we try to expand our set with two functions for intersection and difference:

```
// return intersection of two set
def intersect(s: Set, t: Set): Set = (x: Int) => s(x) && t(x)

// return different of two set
def diff(s: Set, t: Set): Set = (x: Int) => s(x) && !t(x)
```

Before testing our extended set, let us implement a function that transforms our set to a String. To implement this function, we used Scala lazy collection to generate the list of numbers from zero to 100. You can complete the following function with filter and mkString (http://alvinalexander.com/scala/how-to-convert-scala-collections-to-string-mkstring) function to return string representation for the set:

```
def toStr(s: Set): String = {
  val all_possible_members = List.range(0,100)
  val list_of_members = all_possible_members.filter(member => s(member))
  list_of_members.mkString("{", ",", "}")
}
```

To make printing a set even simpler, we can define the following wrapper:

```
def printSet(s: Set) {
    println(toStr(s))
}
```

## 1, 2, 3, Action!

It is the time for testing our set with all defined operators. Try to define two sets to hold the following members:

```
first_set = {1,2,3,4,5}
second_set = {4,5,6,7,8}
```

After defining the sets, try to calculate and print the results of the following logical operations:

intersection_of_sets = first_set ∩ second_set
union_of_sets = first_set ∪ second_set
difference_set =first_set - second_set

**Hint:** to define two sets, first_set, and second_set, you can use lazy list collection and map function. You can put all members of a list to one set with the union function and reduceLeft function.

**Answer:**

```
// defining the first set
val first_set = List.range(1,6).map(member =>
singleton_set(member)).reduceLeft((a,b) => union(a,b))
// defining the second set which use shorthand syntax
val second_set = List range(4,9) map(singleton_set) reduceLeft(union)

// union of two sets
val union_of_sets = union(first_set, second_set)
// printing the union of two  set
printSet(union_of_sets)

// intersection of two sets
val intersection_of_sets = intersect(first_set, second_set)
// printing the intersection of two set
printSet(intersection_of_sets)

// difference of two set
val difference_set = diff(first_set, second_set)
// printing the difference of two set
printSet(difference_set)
```

The last function to extend our set is defining a map function, which accepts a set and a function from int to int. We use the recursive function to implement the map. In functional programming a recursive function is popular and programmers use them instead of loop keywords. Let us test them with a function that multiplies each member of the set by 2. We apply this function to the first set to make the third set:

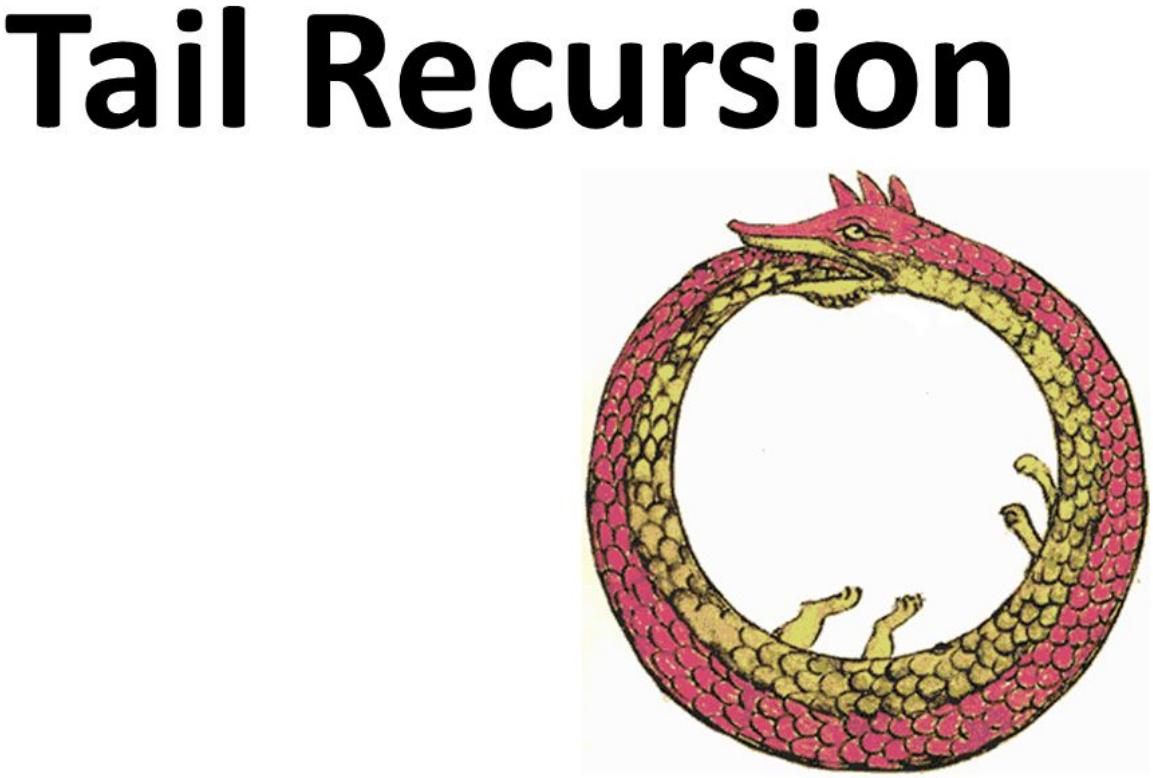
```
// implementation of recursive map function
def map(s: Set, f: Int => Int): Set = {
  def iter(a: Int): Set = {
    if (a > 100)
      return ((x:Int) => false)
    else if (contains(s,a))
      union(iter(a+1), singleton_set(f(a)))
    else iter(a+1)
  }
  iter(0)
}

// function for multiply each member by 2
def multiply_by_2(num:Int):Int = num * 2

val third_set = map(first_set,multiply_by_2)

 printSet(third_set)
```

# 1.5.6
# Tail Recursion




*Tail Recursion (source: slideplayer.com/slide/5923079/)*

The recursive map function that we implemented in the last chapter is not memory-efficient since after calling the inner function in the second `if`, it needs to keep `singleton_set(f(a))` and waiting to calculate recursive function. For just a few iterations, this extra memory is negligible. However, if the number of iterations grows suddenly, they could occupy the entire computer memory. The efficient way of implementing this type of mapping is to use a tail recursive (https://en.wikipedia.org/wiki/Tail_call) function. In a tail recursive function, the inner function call does not need to keep track of any extra data. Scala optimizes tail recursive functions to be executed as memory efficient as loop keywords. The tail recursive implementation of map function is as follows:

```
// tail recursive map function
def map_tail_recursive(s: Set, f: Int => Int): Set = {
  def iter(a: Int, mapped_set: Set): Set = {
    if (a == 100) mapped_set
    else if (contains(s, a)) iter(a+1,union(mapped_set,singleton_set(f(a))))
    else iter(a + 1,mapped_set)
  }
  iter(0,(x: Int) => false)
```

```
}
```

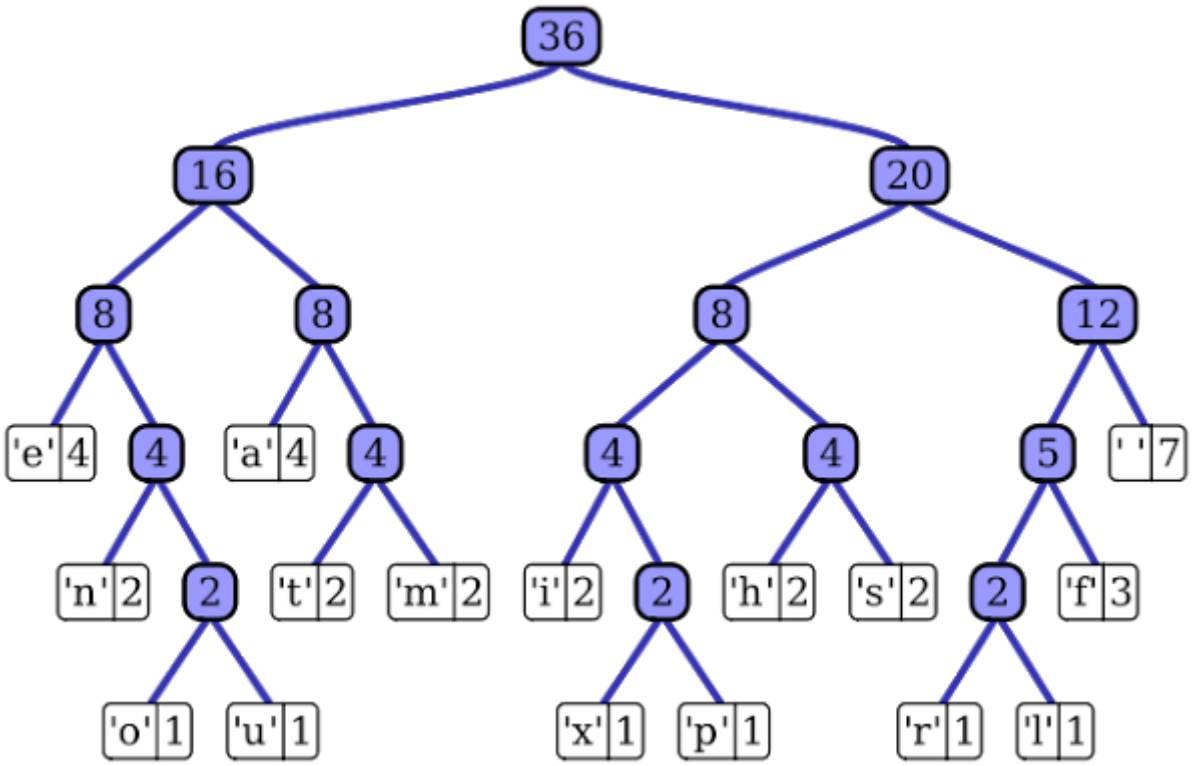
# 1.5.7
# An Advanced Example

## Huffman Code

> **Note:** This is an advanced programming example using Scala functional programming facilities. This challenge can help you with improving your Scala skills, however, it is not mandatory!

## Huffman Coding

In this section, we implement Huffman Coding (https://en.wikipedia.org/wiki/Huffman_coding) which is used for lossless data compression. Huffman coding is nothing but a binary tree (as depicted below) which stores the characters (of the input string that we want to compress) and the corresponding weights in its leaves. The branches of the tree hold the weights of leaves that connected to them.




*Huffman Tree (source: Wikipedia)*

In our implementation, the encoding of a character is the number of times that character appear in the given text. As demonstrated in the above diagram, leaves near the root of the tree have larger weights. Huffman Coding encodes leaves to a binary representation. The binary representation of the tree of leaf characters are calculated with traversing the tree from root to the leaf which traversing left and right branches encode to `0` and `1`, respectively.

Let's implement the base data class to hold the leafs and branches of the data. We used case class to

implement our data class. Case class is almost similar to the Scala class but it provides decomposition mechanism with pattern matching.

```
abstract class Huffman_tree
case class Branch(left: Huffman_tree, right: Huffman_tree, weight: Int) extends
Huffman_tree
case class Leaf(char: Char, weight: Int) extends Huffman_tree
```

In the next step, a function is needed to calculate the weight of each character in a text. Try to implement a function that converts a String to a list of tuple characters and their corresponding weights. For more information about Scala's list please check here (http://alvinalexander.com/scala/different-ways-create-populate-list-scala-cookbook-range-nil-cons).

```
def number_times(text: String): List[(Char, Int)] = {
  val chars = text.toList
  def timesIteration(chars: List[Char], counter: List[(Char, Int)]): List[(Char,
Int)] = {
    if (chars.isEmpty)
      counter
    else {
      val head = chars.head
      val count_head = chars.count(_ == head)
      val filter_chars = chars.filter(_ != head)
      timesIteration(filter_chars, counter.::(head, count_head))
    }
  }
  timesIteration(chars, Nil)
}
```

Now let's test our code for "*This text will be encoded with Huffman tree*":

```
val t = "This text will be encode to very small Huffman tree"
println(number_times(t))
```

Now, in the next step, we construct the Huffman Code. The following algorithm is used to implement the Huffman Code:

1. Convert text to list of characters and their corresponding weights.
2. Convert the above list to a priority queue of leaves based on their weight.
3. While there are more than two leaves in the priority queue:
    1. Remove two members of the queue with the lowest weight.
    2. Combine them with a branch.
    3. Put branch back to queue.

Let's implement a few auxiliary functions to convert characters and weights of the list to leaves, finding the weight tree and combining the leaves to branches before implementing the above algorithm.

Now try to implement and test a function convert list chars weight to list of leaves. You can use the map to convert list chars and theirs weight to list of leaves:

```
def make_leaves(chars_weight: List[(Char, Int)]): List[Huffman_tree] =
chars_weight.map(cw => Leaf(cw._1, cw._2))
```

```
val chars_weight = number_times(t)
val list_leaves = make_leaves(chars_weight)
```

Now try to implement and test two functions for the weight of the tree and combine tree. You can use pattern matching to implement both functions. You can use the pattern matching to implement weight function. For more information about pattern matching please check here (http://alvinalexander.com/scala/how-to-use-case-classes-in-scala-match-expressions).

```
def weight(tree: Huffman_tree): Int = tree match {
  case Leaf(c, w) => w
  case Branch(l, r, w) => w
}

def combine(left_branch: Huffman_tree, right_branch: Huffman_tree) =
  Branch(left_branch, right_branch, weight(left_branch) + weight(right_branch))
```

Now, It is time to implement the Huffman code and convert the above text stored in the t variable to HuffmanCode. You can use pattern matching on the list which more information about it can find here to implement this function.

```
def make_huffman_tree(trees: List[Huffman_tree]): Huffman_tree = {
  val ordered_trees = trees.sortBy(weight)
  ordered_trees match {
    case t :: Nil => t
    case t :: ts =>
make_huffman_tree((combine(t,ts.head)::ts.tail).sortBy(weight))
  }
}
```

Finally, try to write a code to decode the secret code stored in the below variable secret:

```
val ht = make_huffman_tree(list_leaves) // Create the Huffman Tree
val my_secret  = List(0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,
1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1)

def decode_(tree: Huffman_tree, bits: List[Int]): List[Char] = {
  def decodeRe(innerTree: Huffman_tree, innerBits: List[Int], res: List[Char]):
List[Char] = {
    innerTree match {
      case Leaf(c, w) if innerBits.isEmpty => (c :: res).reverse
      case Leaf(c, w) => decodeRe(tree, innerBits, c :: res)
      case Branch(l, r, w) if innerBits.head == 0 => decodeRe(l, innerBits.tail,
res)
      case Branch(l, r, w) if innerBits.head == 1 => decodeRe(r, innerBits.tail,
res)
    }
  }
  decodeRe(tree, bits, Nil)
}

val re = decode_(ht,my_secret) //decode the secret!
println(re.foldLeft("")(_+_))
```

# 2
# Apache Hadoop

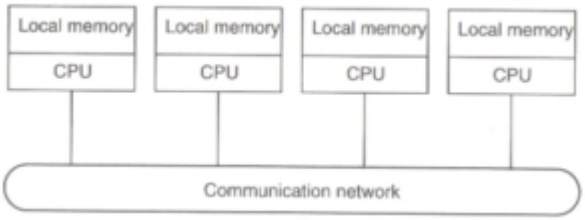
## Ecosystem and Applications


*Hadoop (source: timoelliott.com)*

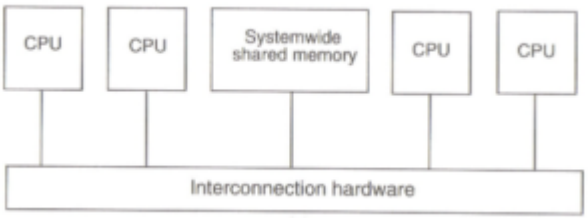
To be able to process Big Data, having some requirements such as rapid processors (in terms of CPU clock) and fast access to data (e.g., disk speed and CPU high bandwidth) is essential. In practice, however, there are some natural barriers such as empirical upper bounds on the CPU speeds and the tight budget. Since early stages of the digital computer era, practitioners followed ***parallelism*** pathway to bypass these obstacles.

Using parallel processing tools, we can simultaneously perform many computational operations on several datasets. The main challenge here is to identify the right parallel structure, optimal number of CPU/s, peripherals, and communications and matching application program design. Essentially, if we can distribute the process and data to separate devices, we can enhance the performance of data processing without necessarily using very fast CPU/s. This leads us to ***distributed system*** designs.

A distributed system is a system in which components located at networked computers communicate and coordinate their actions only by passing messages. Regardless of the size of a distributed system, it appears to its users as a single coherent system. From the computer architecture point of view, distributed systems can broadly be categorized into the ***loosely*** and ***tightly*** coupled systems. In a typical tightly coupled distributed system all devices share a common primary memory, whilst in loosely coupled systems the processors do not share their local memory with the other CPU/s.




(https://www.alexandriarepository.org/media/loosly-coupled1)




(https://www.alexandriarepository.org/media/tightly-coupled1)

*Distributed Systems: loosely coupled vs tightly coupled system*

There are several models for accessing distributed resources and executing distributed applications, such as file model, function call model, and distributed object model. Each of these models can be assessed from different aspects such as performance (e.g., speed), scalability, reliability (e.g., resource availability and fault tolerance), and utilization.

In the rest of this chapter, we discuss Apache Hadoop as one the most famous distributed systems that practitioners use extensively in Big Data processing.

# 2.1 Introduction to Hadoop

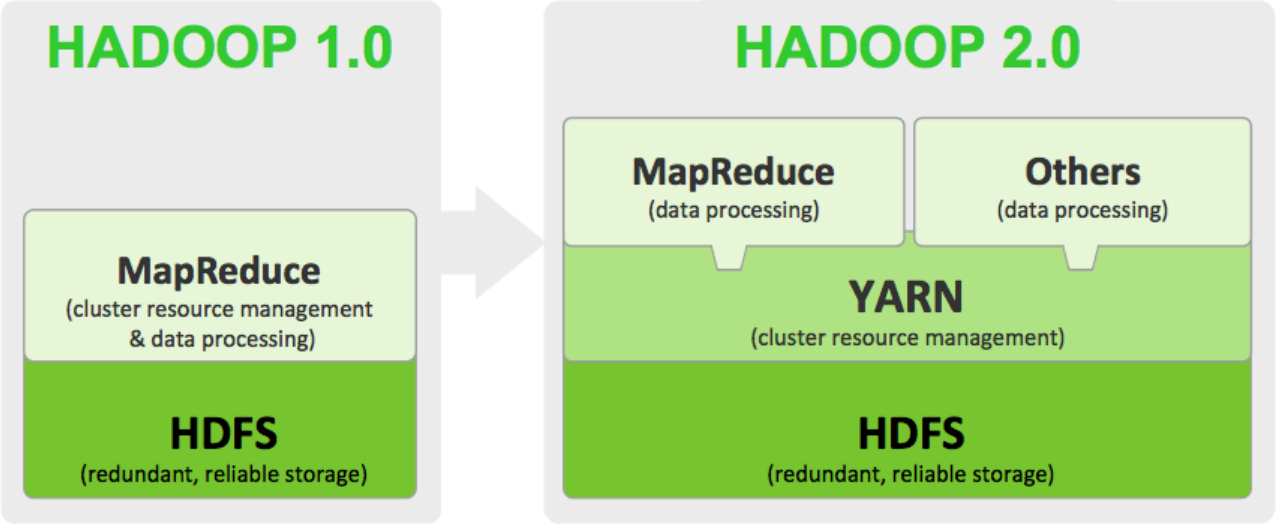
Apache Hadoop is an open source software framework (under Apache license) that implements a significantly scalable and highly fault-tolerant distributed system for data storage and processing. Hadoop consists of two main components namely ***Hadoop Distributed File System*** (HDFS) and ***MapReduce***, which both were inspired by a Google paper that was published in 2003. The HDFS is a self-healing high bandwidth cluster storage solution and MapReduce is a fault-tolerant distributed processing framework. Nowadays, several distributions of Hadoop are released by different organizations (http://hadoopilluminated.com/hadoop_illuminated/Distributions.html) such as Apache (http://hadoop.apache.org), Cloudera (http://www.cloudera.com), MapR (http://www.mapr.com), and Horton Works (http://www.hortonworks.com).

The key values of using Hadoop in Big Data applications are:

- **Flexibility:** It can store almost any type of data with or without a predefined schema.
- **Affordability:** It costs a fraction of the traditional solutions.
- **Generality:** It has a large and actively growing ecosystem
- **Scalability:** Its scalability is practically proven in many different scenarios when dealing dozens of petabyte data sets.

## Architecture

So far, three major versions of Hadoop have been released. The original version, Hadoop 1.0, extensively uses MapReduce programming for batch jobs. However, this implementation can be I/O intensive which is not ideal for interactive analysis such as graph processing, machine learning and on other memory intensive algorithms. In Hadoop 2.0 architecture, this limitation is lifted to a great extent. The two major changes in Hadoop 2.0 are the introduction of ***HDFS Federation*** and the cluster resource manager ***YARN***. The new Hadoop 3.0 implementation does not change the overall architecture of Hadoop framework as much as Hadoop 2.0 did. However, Hadoop 3.0 restructures (https://blog.cloudera.com/blog/2016/09/getting-to-know-the-apache-hadoop-3-alpha/) YARN, improves its cloud compatibility and user experience.




*Hadoop 1.0 vs Hadoop 2.0 (source: dataconomy.com)*

# Hadoop Distributed File System

## HDFS Evolution

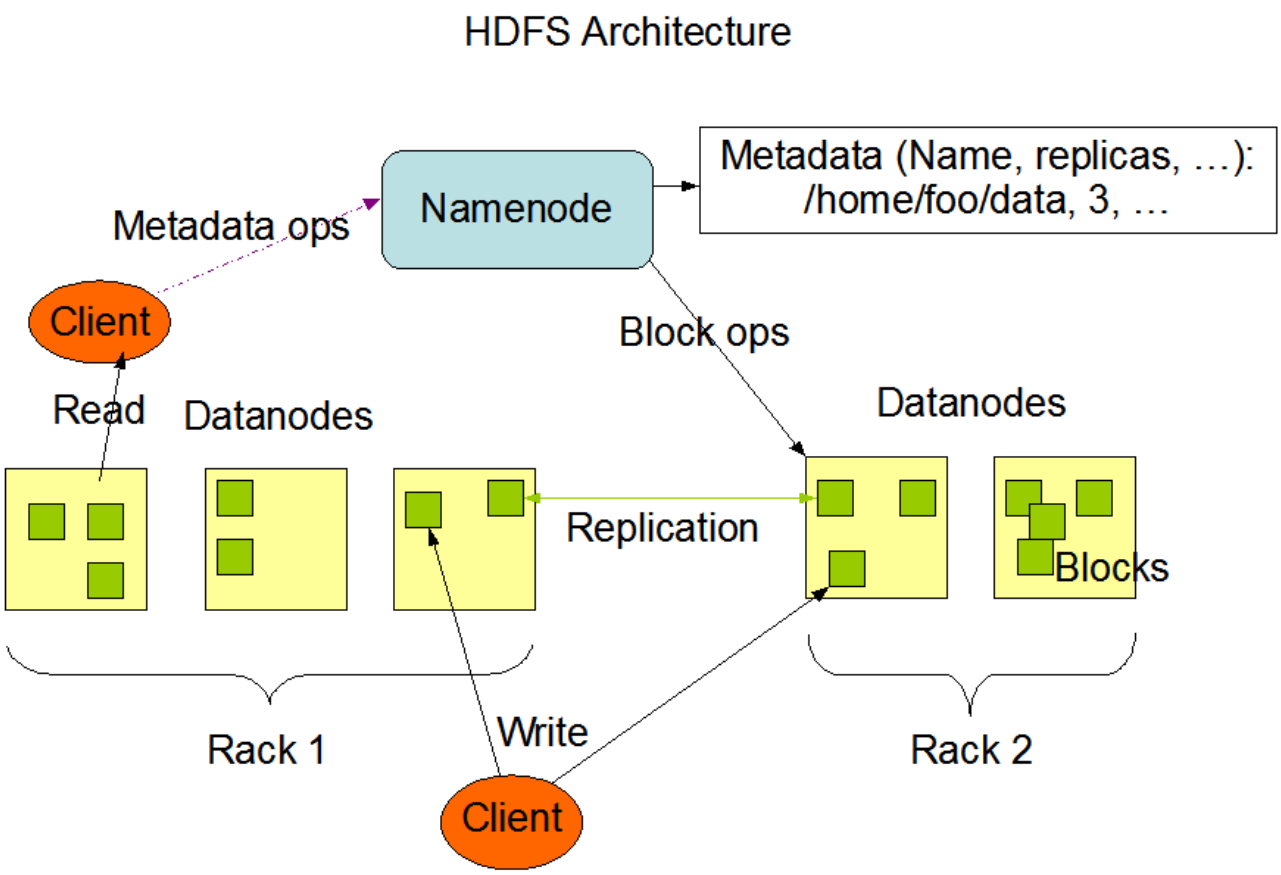
HDFS, as mentioned earlier, is the core part of Hadoop file system which comprises two major components: ***namespaces*** and ***block storage*** service. The namespace service manages operations on files and directories (e.g., creating new directories and modifying the existing files). The block storage service implements data node cluster management, block operations, and replication.

A single **N*ameNode*** in Hadoop 1.0 used to manage the entire namespace for a Hadoop cluster. Using the HDFS Federation introduced by Hadoop 2.0, multiple NameNode servers manage the namespaces. This advances Hadoop capabilities for multiple namespaces, horizontal scaling, and performance improvements. Hadoop 3.0 enhances the storage overhead from up to 300% (in Hadoop 2.0) to only 50%.

## HDFS Architecture

Every HDFS cluster consists of one single NameNode. A NameNode is typically a master server which is responsible for namespace management (i.e., traditional hierarchical file organization) and access control. Furthermore, every node of a cluster consists of one or more ***DataNodes***. A DataNode is a slave server that manages storage attached to its parent nodes.

HDFS allows users to store their data in files. Internally, each file is split into one or more ***blocks***. These blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations such as opening, closing, and renaming files/directories. The NameNode also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the client side. They also perform block creation, deletion, and replication upon instruction from their master NameNode server.



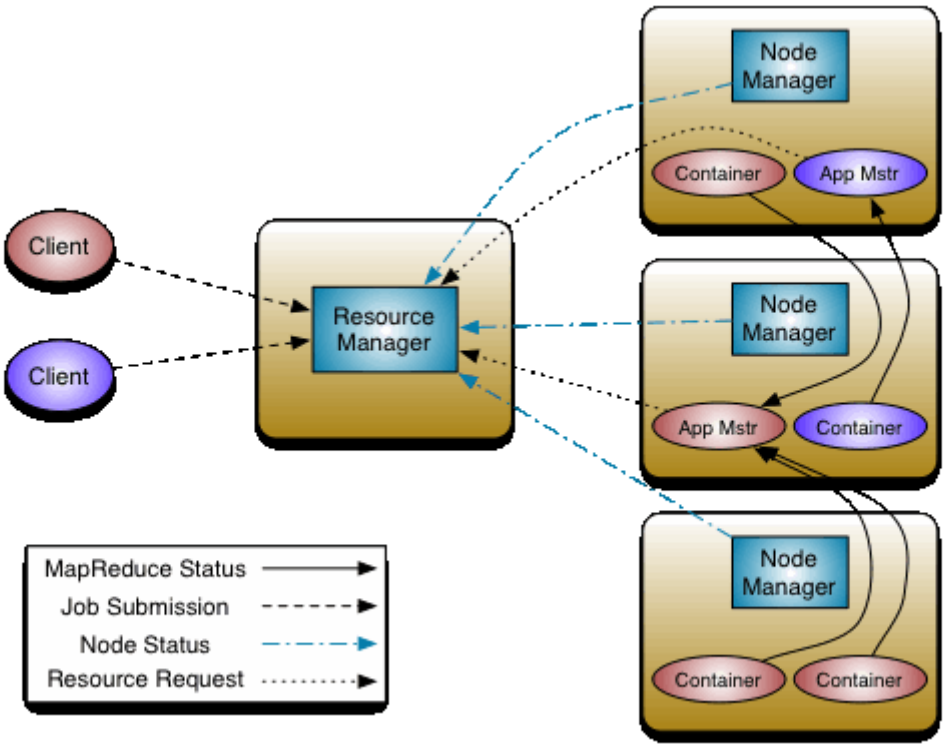

*HDFS Architecture (source: apache.org)*

The name system namespace in HDFS is very similar to the most other file systems. So as HDFS users we can easily create, copy, rename or remove files. The HDFS also supports the common file permissions/rights. The main difference from the user perspective is that HDFS does not support hard or soft links (e.g., shortcuts).

To be able to reliably store massive files across multiple hardware in a large cluster, HDFS stores each file as a sequence of blocks. The blocks of a file are also replicated for the maximum fault tolerance.

## YARN

YARN, the other major advance in Hadoop 2.0, is sometimes called as the operating system of Hadoop. The reason behind this naming is It does the main tasks that an ordinary operating system is usually responsible for. Managing and monitoring workloads, maintaining a multi-tenant environment, and implementing security controls are some of the YARN's responsibilities.




*YARN Architecture (source: apache.org)*

The ultimate goal of YARN is to split the functionalities of resource management and job scheduling into separate services. YARN consists of two core elements: a global ***ResourceManager*** (RM) and per-application ***ApplicationMaster*** (AM). The RM manages the resources among all the applications in the system. Its couple, ***NodeManager***, is the per-machine framework agent who is responsible for containers, monitoring their resource (e.g., CPU, memory, disk, network) usage. The AM is a library responsible for negotiating resources from the RM and working with the NodeManagers to execute and monitor the assigned tasks.

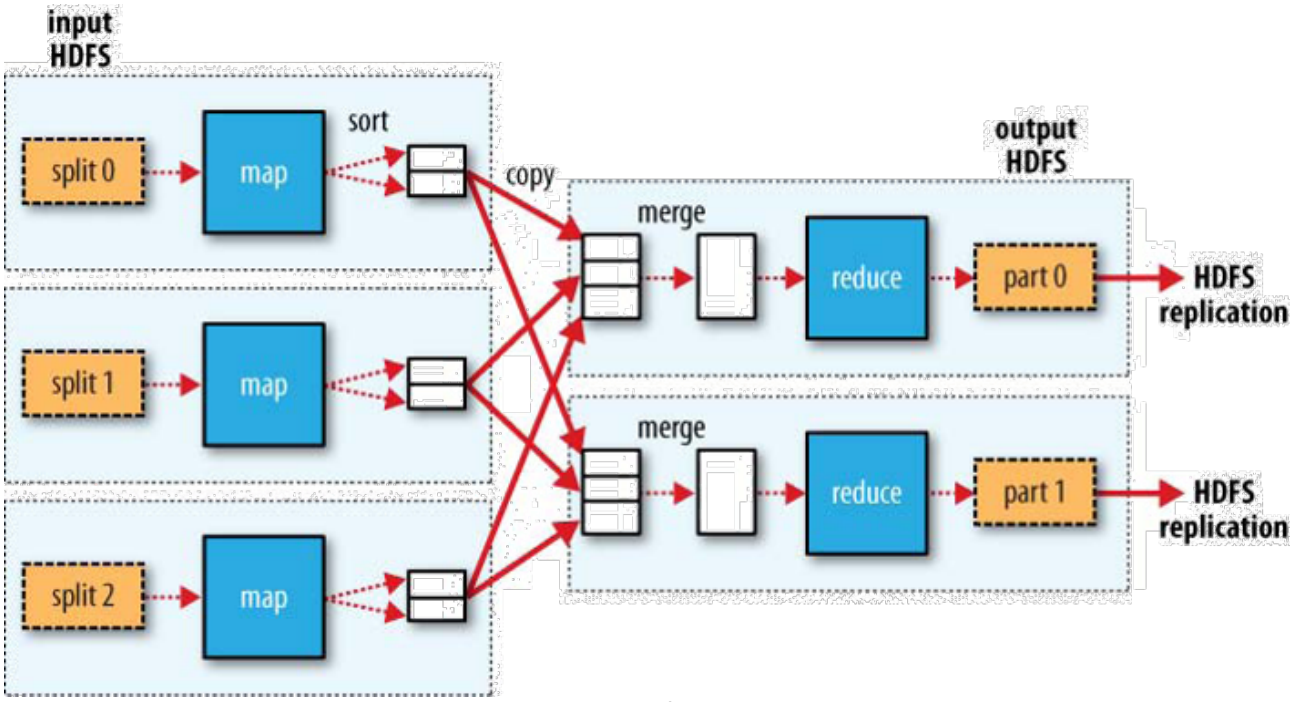
The RM consists of two main components: ***Scheduler*** and ***ApplicationsManager*** (not to be confused by ApplicationMaster). The Scheduler is responsible for allocating resources based on the resource requirements of the applications without performing any monitoring. The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the AM and restarting the AP in the case of failure.

## MapReduce

MapReduce is a framework for writing applications which process vast amounts of data in parallel on large clusters. A MapReduce ***job*** usually splits the input dataset into several independent chunks which are processed by the ***map*** tasks in a completely parallel fashion. Then, the framework sorts the outputs of the maps, to feed the ***reduce*** tasks. The jobs' inputs and outputs are usually stored in a file system. Working with MapReduce is easy as scheduling tasks, monitoring, and re-execution (in the case of failure) are handled by the framework.

The MapReduce framework operates exclusively on `<key, value>` pairs, which means the input and output of each job are nothing but `<key, value>` pairs. The following shows a simple MapReduce scenario where `<k1, v1>` is the input pair and `<k3, v3>` is the output pair:

`<k1, v1>` -> map -> `<k2, v2>` -> combine -> `<k2, v2>` -> reduce -> `<k3, v3>`




*MapReduce*

The MapReduce framework consists of three major parts. The ***reader***, the ***mappers***, and the ***reducers***.

## The Reader

The reader, as its name suggests, reads the input file and divides it into two or more splits. Then, each of these splits is assigned to a map processor.

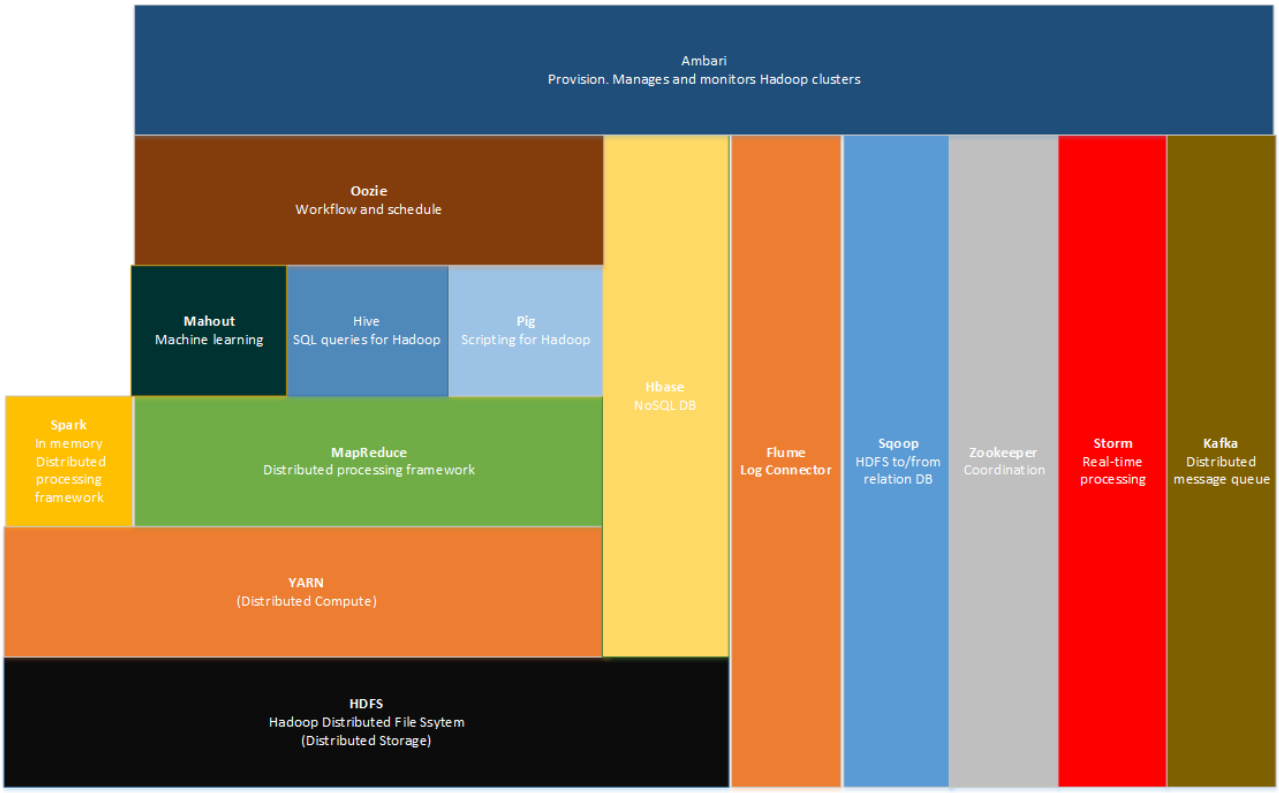
## The Mappers

The mapper mainly processes the input data - the input data is typically stored in the HDFS in the form of file or directory. Given the input data, the mapper reads them line by line, processes the data, and produces smaller fragments of processed data to be used in the reducer phase. The process in the mapper means to map the input <key, value> pairs to a set of intermediate <key, value> pairs. As can be observed from the above example, mapper maps input <key, value> pairs to a set of intermediate <key, value> pairs. Each map is an individual task that transforms an input record into zero to many intermediate records. Note that the input and output records can be in the same or different data types.

## The Reducers

The reducer reduces a set of intermediate records which have a common key to a smaller set of records. Typically, reducer has three primary phases: ***shuffle***, ***sort***, and of course, ***reduce***. The shuffler's job is to distribute sorting and aggregation to many reducers. At the same time, the sorter groups the reducer inputs by keys, such that all <key, value> pairs with a common key from a single group. The reducer then reduces the input group of pairs to a single pair. The output of the reduce task, which can have any arbitrary order, is finally written back to the filesystem.

# 2.2 Hadoop Technologies




*Hadoop Technologies (source: adriancitu.com)*

To leverage all the benefits that Hadoop is offering, we are not limited to directly use HDFS and MapReduce. Several tools and technologies are already developed based on Hadoop that make working with this framework easier. Each of these open source or commercial technologies is designed, developed, and matured to perform some specific tasks. Therefore, one may need to use a few of them based on the objectives and scale of the project. We will briefly introduce some of these technologies in the following. Other tools such as HBase and Spark will be covered in detail later.

In the following, we briefly introduce some of these technologies. Some of these tools such as HBase and Spark are covered in other chapters in more details.

## Spark

Spark (http://spark.apache.org/) is a general compute engine that offers scalable and efficient data analysis. It uses its own data processing framework instead of MapReduce. In contrast with Hadoop MapReduce, Spark can perform in-memory computing instead of on disk. By using in-memory computing, Spark workloads typically run between 10 and 100 times faster compared to disk execution. This is a key reason for its popularity and wide adoption.

Spark can be adopted independently of Hadoop. Nevertheless, it is usually used with Hadoop as an alternative to MapReduce programming scheme. Common use cases for Spark include real-time queries, ETL operations, stream processing (Spark Streaming (http://spark.apache.org/streaming/)), iterative algorithms,

graph computations (GraphX (http://spark.apache.org/graphx/)), and machine learning (MLib (http://spark.apache.org/mllib/) library). Unlike the core Hadoop, Spark supports SQL via Spark SQL (http://spark.apache.org/sql/). The Spark programming environment works interactively with Scala, Python, and R (http://spark.rstudio.com/).

The below diagram illustrates the open source ecosystem.

*Spark Open Source Exosystem*

# HBase

HBase (https://hbase.apache.org/), also known as Hadoop Database, is a non-relational, distributed, and column-oriented database that runs on top of HDFS. It helps us to achieve real-time read/write random access to datasets with billions of rows and millions of columns. In contrast with Hive, HBase does not support SQL-like queries. However, one can use Hive to leverage HQL on top of HBase.

HBase algorithm and structure are inspired by the famous Google BigTable (https://research.google.com/archive/bigtable.html) technology and used by many giant corporations such as Facebook as a part of its messenger.

HBase core components are:

- **HBase Master:** To negotiate load balancing across all RegionServers and maintains the state of the cluster.
- **RegionServer:** To deploy on each machine and host data and process I/O requests.

# Hive

Hive (https://hive.apache.org/) is a big player in the Hadoop ecosystem which provides an SQL-like interface

called HiveQL (HQL). Hive is a data warehouse infrastructure that provides easy data summarization, ad-hoc queries, and the analysis of large datasets stored in HDFS.

Hive facilitates a mechanism to project structure onto the data, query them, and return the results. Since the data stored in Hadoop clusters are unstructured (recall that HDFS is schema-agnostic), the ETL operations that Hive offers are very beneficial.

Hive is not a relational database system by itself. Instead, it is a query engine that supports the parts of SQL specific to querying data and some additional support for writing new tables or files. HQL queries are translated to MapReduce jobs to exploit the scalability of MapReduce.

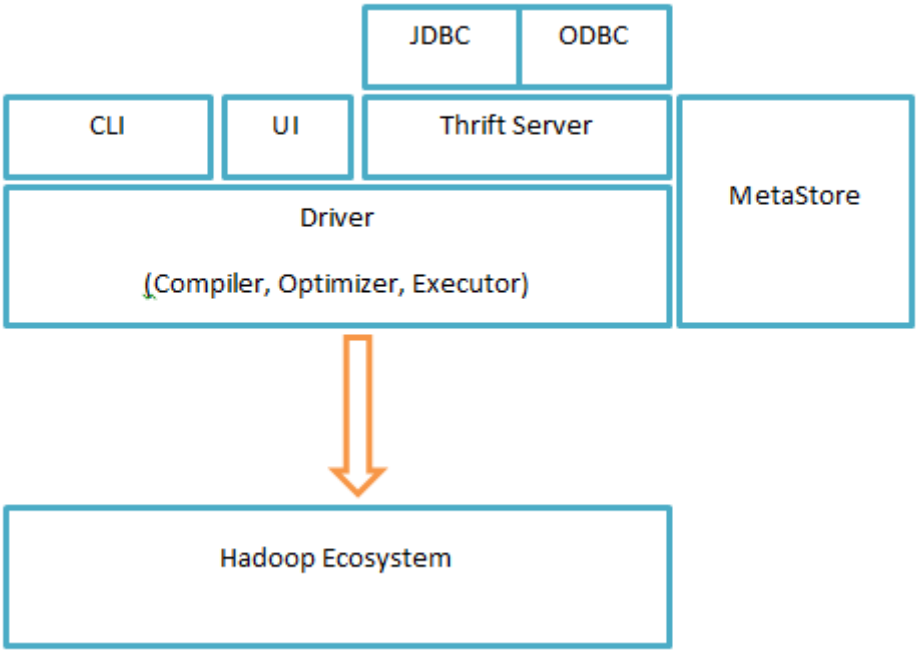
Hive jobs are very optimized for scalability purposes such as computing a summary function over all rows. In contrast, it is not designed to minimize the latency. Therefore, in cases such as when we need a few rows to be returned and perhaps needing them very quickly, Hive is not the best option.

Note that Hive allows the MapReduce programmers to plug in their custom codes whenever necessary. It also supports custom extensions written in Java for reading and optionally writing custom formats (e.g., JSON and XML formats). As a result, data analysts have flexibility in working with data from multiple sources and formats, with very minimal need for complex ETL processes.

The main building blocks of Hive are:

- **Metastore:** To store metadata about columns, partition and system catalog.
- **Driver:** To manage the life-cycle of a HiveQL statement
- **Query Compiler:** To compile HiveQL into a directed acyclic graph.
- **Execution Engine:** To execute the tasks in proper order which are produced by the compiler.
- **HiveServer:** To provide a ***Thrift*** interface and a JDBC / ODBC server.
- **HCatalog:** To help data processing tools read and write data on the grid (supports MapReduce and Pig).
- **WebHCat:** To provide HTTP/REST interface to run MapReduce, Yarn, Pig, and Hive jobs.

The following figure demonstrates the Apache Hive architecture.




*Apache Hive architecture (source: Wikipedia)*

# Pig

Pig (https://pig.apache.org/), similar to Hive, is a platform for analyzing and querying huge datasets. In contrast, Pig does not support SQL-like queries. Instead, it has its own scripting language called Pig Latin. Pig can be seen as an alternative to Java programming for MapReduce which translate Pig Latin to MapReduce functions.

Pig Latin is a high-level scripting language for expressing data analysis programs. It is essentially designed to fill the gap between the declarative style of SQL and the low-level procedural style of MapReduce. In comparison with SQL, Pig Latin leverages lazy evaluation (https://en.wikipedia.org/wiki/Lazy_evaluation), can perform ETL operations (i.e., extract, transform, load) and store data at any point in the pipeline.

Pig Latin contains many built-in data manipulation operations such as grouping, joining, and filtering. This large collection of operations makes Pig ideal for developers who are familiar with one or more scripting languages, but not so much SQL. Because of this great flexibility, users can develop their own functions using their preferred scripting language (e.g., Ruby and Python).

# Mahout

Mahout (http://mahout.apache.org/) is a scalable machine learning and data mining library that implements a wide range of algorithms in a distributed fashion. All Mahout algorithms are designed in a way to benefit from the efficiency and scalability of Hadoop platform. Mahout algorithms fall into four core categories:

- **Recommendation Systems:** collaborative filtering

- **Classifiers Algorithms:** Logistic Regression, Naive Bayes, and Hidden Markov Models
- **Clustering Algorithms:** Canopy, K-Means, and Spectral Clustering
- **Dimensionality Reduction:** Singular Value Decomposition and Principal Component Analysis

# Other Technologies

There is a wide range of other open source projects (under Apache license (https://www.apache.org/licenses/LICENSE-2.0)) that are built on top of Hadoop platform. Many of them are still in the incubator state, while the others are already matured. Considering the time and space limitation, we only discuss them very briefly.

- **Ambari (https://ambari.apache.org/):** A web-based tool that is created to help manage Hadoop. It can support many of Hadoop-based technologies such as HBase, HBase, Piq, Sqoop and Zookeeper. Ambari's dashboard keeps track of cluster health and helps diagnose performance issues.
- **Avro (https://avro.apache.org/):** A data serialization system.
- **Cassandra (http://cassandra.apache.org/):** A linearly scalable multi-master distributed database management system with no single points of failure.
- **Cascading (http://www.cascading.org/):** A popular high-level Java API that hides many of the complexities of MapReduce programming behind more intuitive pipes and data flow abstractions.
- **Cascalog (http://cascalog.org/):** A data processing and querying library that adds Logic Programming concepts inspired by Datalog to Cascading. Its name is a contraction of Cascading and Datalog.
- **Chukwa (http://chukwa.apache.org/):** A data collection system built on top of Hadoop platform for displaying, monitoring and analyzing large distributed systems.
- **Drill (https://drill.apache.org/):** A schema-free SQL query engine for Hadoop
- **Flume (https://flume.apache.org/):** A distributed framework for collecting, aggregating and moving huge amounts of streaming data (e.g. log data or text files) in and out of Hadoop.
- **Impala (https://impala.incubator.apache.org/):** A native analytic database for Hadoop that is shipped by BiG Data giants Cloudera, MapR, Oracle, and Amazon.
- **Kafka (https://kafka.apache.org/):** A distributing streaming platform that is often used instead of traditional message brokers in the Hadoop environment. It is designed for higher throughput and provides replication and greater fault tolerance than traditional tools.
- **Knox** (https://knox.apache.org/): An application gateway that provides REST API for the Hadoop Ecosystem.
- **Oozie (http://oozie.apache.org/):** A workflow scheduler and coordination system that manages how workflows start and execute. It is a server-based web application.
- **Ranger (http://ranger.apache.org/)**: A framework that enables monitoring data and provides data security across the Hadoop ecosystem.
- **Scalding (http://www.cascading.org/projects/scalding/):** An extension to Cascading that enables application development with Scala APIs. It provides implementations of common data analysis/manipulation operations as well as adds matrix and algebra models (useful for implementing machine learning algorithms).
- **Storm (http://storm.apache.org/):** A fast distributed computation system that makes it easy to reliably process unbounded streams of data in real-time. It has many use cases such as real-time analytics, online machine learning, continuous computation, and ETL.
- **Sqoop (http://sqoop.apache.org/):** A command-line interface that facilitates moving bulk data from Hadoop into relational databases (or other structured data stores).
- **Tajo (http://tajo.apache.org/):** A robust relational and distributed data warehouse system that is designed for low-latency and scalable ad-hoc queries, online aggregation, and ETL on large data sets stored on HDFS. Tajo, which supports SQL, allows direct control of distributed execution and data flow.
- **Tez (https://tez.apache.org/):** A generalized application framework which provides a powerful and flexible engine to execute a complex directed-acyclic-graph of tasks. Its main goals are to empowering end users and improve execution performance. Tez is being adopted by Hive, Pig and

other frameworks in the Hadoop ecosystem.

- **Zookeeper (https://zookeeper.apache.org/):** A high-performance coordination service for maintaining configuration information, naming, providing distributed synchronization and providing group services. Some Hadoop technologies such as HBase cannot operate without Zookeeper.

## References:


- Apache Hadoop (http://hadoop.apache.org/)
- Leading Big Data Technologies: Hadoop (https://www.thinkbiganalytics.com/leading_big_data_technologies/hadoop/)
- Hadoop Ecosystem (http://www.bmcsoftware.com.au/guides/hadoop-ecosystem.html)
- Big Data: Understanding Hadoop Ecosystem (https://devops.com/bigdata-understanding-hadoop-ecosystem/)
- Hadoop 101 Explanation (https://dzone.com/articles/hadoop-101-explanation-hadoop)

# 2.3
# Activity: Hadoop

This chapter is devoted to HDFS and MapReduce examples.

# 2.3.1
# HDFS Commands and Examples

In this set of activities, we connect to our VM using SSH connection and practice some of the basic HDFS commands. To get access to your Hadoop instance using the `ssh`, please follow the instructions mentioned here (https://confluence.apps.monash.edu/display/MCC/Access). If you are connecting from any network rather than Monash University Internet connection, you may need to use a VPN (https://www.monash.edu/esolutions/network/vpn) as a proxy.

## 1- Running HDFS

### Start HDFS

To start Hadoop on a Linux machine, we usually need to go to `home/user/hadoop-2.7.3/sbin/` address and run `start-dfs.sh`. However, for your convenience, we add the `start-dfs.sh` to the user path, therefore, you can call it anywhere. After running the script, you see a welcome string similar to the following:

```
user@ubuntu:~$ start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/user/hadoop-2.7.3/logs/hadoop-
user-namenode-ubuntu.out
localhost: starting datanode, logging to /home/user/hadoop-2.7.3/logs/hadoop-
user-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to
/home/user/hadoop-2.7.3/logs/hadoop-user-secondarynamenode-ubuntu.out
```

Your printed outputs can be slightly different. you can check whether the expected Hadoop processes are running using the `jps` command.

Always remember to stop HDFS using `stop-dfs.sh` script when you have completed all exercises.

### Start YARN

Recall that YARN is the main resource negotiator since Hadoop2.0. We need to run it as well.

```
user@ubuntu:~$ start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /home/user/hadoop-2.7.3/logs/yarn-user-
resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /home/user/hadoop-2.7.3/logs/yarn-
user-nodemanager-ubuntu.out
```

Now, if I call `jps` command, my output would be like:

```
user@ubuntu:~$ jps
```

```
23782 Jps
23046 ResourceManager
22265 NameNode
23177 NodeManager
22606 SecondaryNameNode
22415 DataNode
```

Remember to stop Yarn using `stop-yarn.sh` script before stopping HDFS.

## Create Home Directory

If it is the first time you run HDFS, you need to create a home directory using `hadoop fs -mkdir` for the further actions:

```
hadoop fs -mkdir -p /user/user
```

The above command creates a new folder called `user` under the path `/user`. Note that `/user` is a home mount on the HDFS (not our local filesystem). Later on, we can load our data into `/user/user` directory and query them. By creating such directory, we automatically get adequate permissions and storage access onto the HDFS. Note that the generic form is `hadoop fs -mkdir -p /user/`**`$USER_NAME`**, where the **`$USER_NAME`** will be your current login user. In my case, the user is named **`user`**.

You can check whether the folder is created at the right path by `hadoop fs -ls /user`

# 2- Basic HDFS Commands

## Create a Directory

`hadoop fs -mkdir` **`<path>`** creates a directory in HDFS at given path. For example:

```
user@ubuntu:~$ hadoop fs -mkdir newDir # creates newDir
user@ubuntu:~$ hadoop fs -ls           # lists files and folders in the current
path
Found 1 items
drwxr-xr-x - user supergroup 0 2017-06-06 12:46 newDir # can we create another
directory with the same name?
user@ubuntu:~$ hadoop fs -mkdir newDir
mkdir: `newDir': File exists
```

## List Files & Folders

`hadoop fs -ls` **`<args>`** lists the contents of a particular directory. In the following example, by issuing the "**.**", we can see the content of our user folder on the HDFS. As expected, it contains only one recently created `newDir` directory. Note that omitting the `<args>` (in this case ".") also lists the current directory.

```
user@ubuntu:~$ hadoop fs -ls  . # let's see what we got
Found 1 items
drwxr-xr-x   - user supergroup          0 2017-06-06 12:46 newDir
user@ubuntu:~$ hadoop fs -ls    # dropping . does the same thing
```

```
Found 1 items
drwxr-xr-x   - user supergroup          0 2017-06-06 12:46 newDir
user@ubuntu:~$ hadoop fs -ls newDir  # is there anything in the user directory?
user@ubuntu:~$ hadoop fs -ls noneDir  # what about listing the content of
nonexist folder?
ls: `noneDir': No such file or directory
```

As this example shows, we can also report the file/folder permission, their size, and the last modified date-time.

If you want to see entries in all subdirectories of a path, recursively, you need to use -R switch as in the following example:

```
user@ubuntu:~$ hadoop fs -mkdir ./newDir/anotherDir #let's create a subfolder
first
user@ubuntu:~$ hadoop fs -ls -R # recursive listing
drwxr-xr-x   - user supergroup          0 2017-06-06 13:06 newDir
drwxr-xr-x   - user supergroup          0 2017-06-06 13:06 newDir/anotherDir
```

As you can see, `-ls -R` behaves like `-ls`, but recursively displays entries in all subdirectories of the path.

## Upload & Download Files

If you do not have a real file to upload to HDFS, you can create a simple one using `touch` or `cat` commands on linux. The generic format of `touch` would be `touch` **`$FILE_NAME.FILE_EXTENSION`**. For example, `touch newFile.txt` create an empty file named `newFile.txt`. We can also create and then fill a file using `cat >` **`$FILE_NAME.FILE_EXTENSION`**. This command allows you to type some text, and then pressing `ctrl+d` to exit the typing mode. Note that both of these commands create file in our local file system (not distributed). We will upload them to HDFS later on.

```
user@ubuntu:~$ touch newFile.txt #create an empty file
user@ubuntu:~$ cat > anotherFile.txt #create and type into a file
Hello HDFS!
This is a simple text file. I'll press ctrl-d to exit the type mode in the next
line!
user@ubuntu:~$ ls *File.txt
anotherFile.txt  newFile.txt
```

Let's upload them using `hadoop fs -put` **`<local_src> ... <hdfs_dest>`** command. Note that we can upload the files one by one. However, since our files have a special pattern in their names (both of them end with `File.txt`), we can upload all files at once. In this particular example, we upload all files that match `*File.txt` to a subdirectory that we created previously (`newDir`):

```
user@ubuntu:~$ hadoop fs -ls # see! there is no txt file yet!
Found 1 items
drwxr-xr-x   - user supergroup          0 2017-06-06 13:06 newDir
user@ubuntu:~$ hadoop fs -put *File.txt newDir # upload all files ending
File.txt
user@ubuntu:~$ hadoop fs -ls -R
drwxr-xr-x   - user supergroup          0 2017-06-06 13:23 newDir
drwxr-xr-x   - user supergroup          0 2017-06-06 13:06 newDir/anotherDir
-rw-r--r--   1 user supergroup         98 2017-06-06 13:23
```

```
newDir/anotherFile.txt
-rw-r--r--   1 user supergroup          0 2017-06-06 13:23 newDir/newFile.txt
```

The hadoop `fs -get` **<hdfs_src> <local_dst>** command perform an opposite action. It download files from HDFS to our local filesystem. For example:

```
user@ubuntu:~$ ls *File.txt #let's see if still have the files on local fs!
anotherFile.txt  newFile.txt
user@ubuntu:~$ rm *File.txt #now remove them from local fs
user@ubuntu:~$ ls *File.txt #removed successfully?
ls: cannot access '*File.txt': No such file or directory
user@ubuntu:~$ hadoop fs -get newDir/* #download them from hdfs
user@ubuntu:~$ ls *File.txt #downloaded successfully?
anotherFile.txt  newFile.txt
```

There are two similar commands for uploading/downloading files from/to local directory: `hadoop fs -copyFromLocal` **<local_src> <hdfs_dest>** and `hadoop fs -copyToLocal` **<hdfs_src> <local_dest>**. You may wonder why we have commands that work the same. These two particular commands are designed to avoid confusion when we have similar directory structure of file names in our HDFS and local filesystems. SInce the source in -copyFromLocal and destination for -copyToLocal are restricted to local path, there will not be any confusion for Hadoop. We can achieve the same results using `-put` and `-get` if we explicitly use `file://` and `hdfs://` prefix to clarify the filesystem that we are pointing to.

## Read File Content

We can read the content of text files using `-cat` option.

```
user@ubuntu:~$ hadoop fs -cat newDir/newFile.txt #let's see what is newFile
user@ubuntu:~$ hadoop fs -cat newDir/anotherFile.txt #what about anotherFile?
Hello HDFS!
This is a simple text file. I'll press ctrl-d to exit the type mode in the next
line!
```

## Copy Files

We already know how to copy a file from local filesystem to HDFS and vice versa. Now, we want to copy one or more files/folders from a directory on HDFS file system (source) to another place on HDFS (destination). It can be done easily using `hadoop fs -cp` **<source> <dest>** command. Let's copy `newFile.txt` from `newDir` to `newDir/anotherDir`:

```
user@ubuntu:~$ hadoop fs -ls -R newDir/ #checking the list
drwxr-xr-x   - user supergroup          0 2017-06-06 16:55 newDir/anotherDir
-rw-r--r--   1 user supergroup         98 2017-06-06 13:23
newDir/anotherFile.txt
-rw-r--r--   1 user supergroup          0 2017-06-06 13:23 newDir/newFile.txt
user@ubuntu:~$ hadoop fs -cp newDir/anotherFile.txt newDir/anotherDir #copy a
file
user@ubuntu:~$ hadoop fs -ls -R newDir/ #check the list again
drwxr-xr-x   - user supergroup          0 2017-06-06 16:56 newDir/anotherDir
-rw-r--r--   1 user supergroup         98 2017-06-06 16:56
```

```
newDir/anotherDir/anotherFile.txt
-rw-r--r--   1 user supergroup         98 2017-06-06 13:23
newDir/anotherFile.txt
-rw-r--r--   1
```

## Move Files

hadoop fs -mv **<source> <dest>** command moves files) from source to destination. The only difference between move and copy is that move deletes the files from the source directory. For example, let's move newFile.txt from newDir to newDir/anotherDir. After this operation, there should be no such a file in the source directory.

```
user@ubuntu:~$ hadoop fs -ls -R newDir/ #check the files first
drwxr-xr-x   - user supergroup          0 2017-06-06 16:56 newDir/anotherDir
-rw-r--r--   1 user supergroup         98 2017-06-06 16:56
newDir/anotherDir/anotherFile.txt
-rw-r--r--   1 user supergroup         98 2017-06-06 13:23
newDir/anotherFile.txt
-rw-r--r--   1 user supergroup          0 2017-06-06 13:23 newDir/newFile.txt
user@ubuntu:~$ hadoop fs -mv newDir/newFile.txt newDir/anotherDir/ #move a file
user@ubuntu:~$ hadoop fs -ls -R newDir/ #check the listing again
drwxr-xr-x   - user supergroup          0 2017-06-06 17:23 newDir/anotherDir
-rw-r--r--   1 user supergroup         98 2017-06-06 16:56
newDir/anotherDir/anotherFile.txt
-rw-r--r--   1 user supergroup          0 2017-06-06 13:23
newDir/anotherDir/newFile.txt
-rw-r--r--   1 user supergroup         98 2017-06-06 13:23
newDir/anotherFile.txt
user@ubuntu:~$
```

## Remove Files

Removing files and empty folders is as easy as calling hadoop fs -rm **<arg>** and hadoop fs -rmdir **<arg>** commands. However, removing a lot of not empty folders and their files and subdirectories seems a big task. Fortunately, we can remove files and folders (and their subfolders) regardless of being empty or not using hadoop fs -rm -R **<arg>** commands. This latest command is called recursive remove.

```
user@ubuntu:~$ hadoop fs -ls -R newDir/ #print the list
drwxr-xr-x   - user supergroup          0 2017-06-06 17:23 newDir/anotherDir
-rw-r--r--   1 user supergroup         98 2017-06-06 16:56
newDir/anotherDir/anotherFile.txt
-rw-r--r--   1 user supergroup          0 2017-06-06 13:23
newDir/anotherDir/newFile.txt
-rw-r--r--   1 user supergroup         98 2017-06-06 13:23
newDir/anotherFile.txt
user@ubuntu:~$ hadoop fs -rm -R newDir/anotherDir #remove a folder and all its
contents
17/06/06 17:50:28 INFO fs.TrashPolicyDefault: Namenode trash configuration:
Deletion interval = 0 minutes, Emptier interval = 0 minutes.
Deleted newDir/anotherDir
```

```
user@ubuntu:~$ hadoop fs -ls -R newDir/ #check the list again
-rw-r--r--   1 user supergroup         98 2017-06-06 13:23
newDir/anotherFile.txt
```

# 3- Other Operations

The aforementioned commands accepts some special parameters (also called switches) that expand their applications. For example, by adding `-f` to the `-cp` command we can force HDFS to copy a file even if a file with exact name and extension exist in the destination (that file would be overwritten). Another example is `-p` in `-get` command which preserves the last access/modification times, ownership and the permissions of the file being uploaded.

In addition to these options, there are a number of other commands that we did not discussed here. The following is a list of some of these commands with a short explanation.

| Command | Explanation |
|---|---|
| appendToFile | Appends one or more source files from local file system to the destination file system. |
| checksum | Returns the checksum number of a file. |
| chmod/chown/chgrp | Change permission/ownership/group of a file/folder. |
| df/du | Display free/used space. |
| find | Finds all files that match a given expression. |
| getmerge | Concatenates files from source into a destination local file. |
| setrep | Changes the replication factor of a file/folder. |
| tail | Displays last parts of a text file. |

For more information about a particular command issue `-help` or `-usage`.

# References


1. HDFS Users Guide (https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html)
2. File System Shell Guide (https://hadoop.apache.org/docs/r1.2.1/file_system_shell.html)
3. Hadoop Environment Setup (https://www.tutorialspoint.com/hadoop/hadoop_enviornment_setup.htm,)
4. Hadoop Command Reference (https://www.tutorialspoint.com/hadoop/hadoop_command_reference.htm)
5. Top 10 Hadoop Shell Commands to Manage HDFS (https://dzone.com/articles/top-10-hadoop-shell-commands)
6. Running Hadoop on Ubuntu Linux (Single-Node Cluster)
   (http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/)

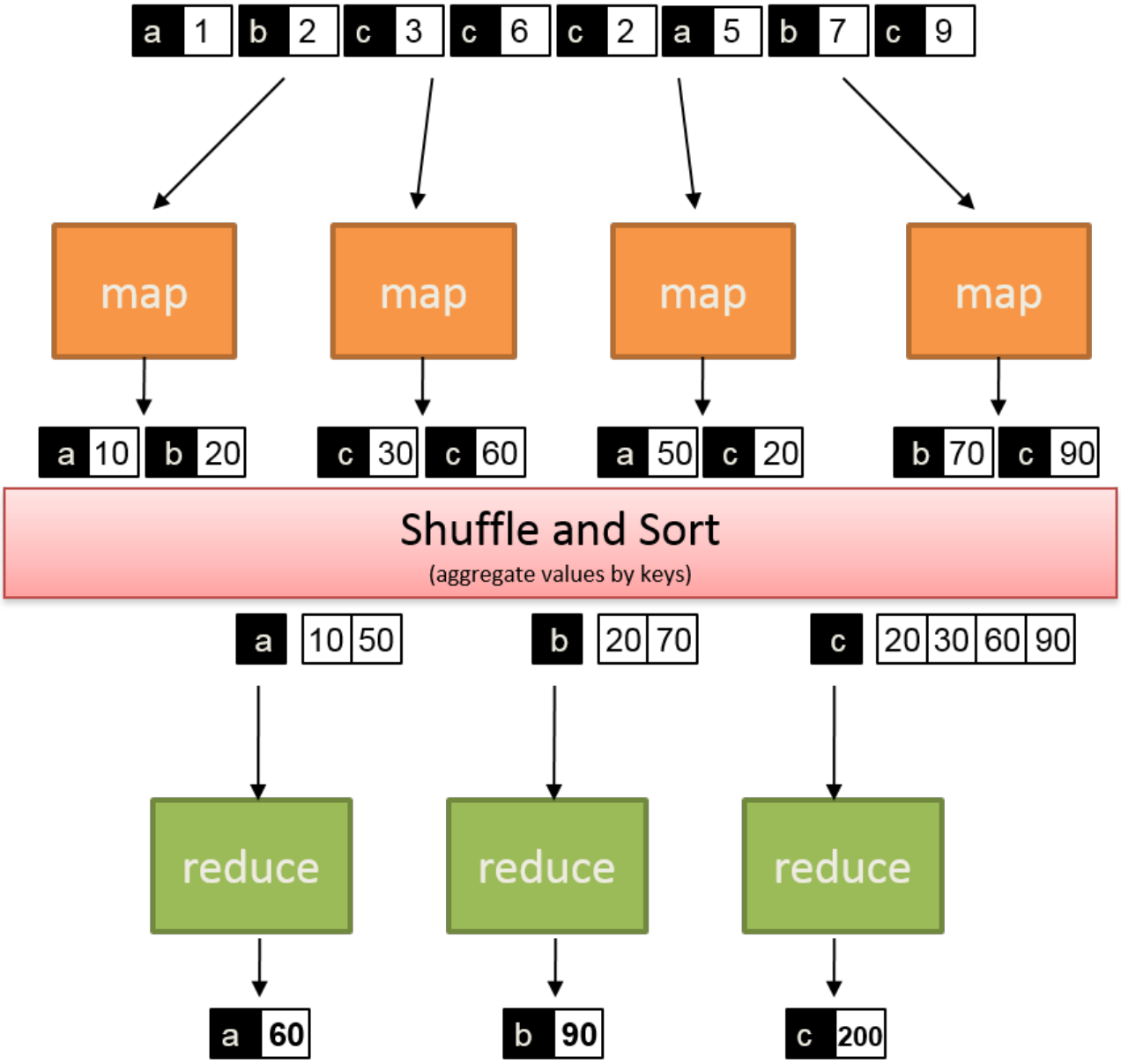
# 2.3.2 MapReduce Examples

In this section, we provide some simple to challenging MapReduce examples. Regardless of the interface, you may choose to develop a MapReduce program (e.g., Hadoop and Scala, PySpark, or SparkR), the foundation and workflow of the programs are the same.

## A Simple Numerical Example

The following figure illustrates a MapReduce job example. The input of the job is a dataset where each entry has two values: a single English letter (e.g., 'a' or 'b') and an integer number. Note that letters and numbers might be repeated in more than a record. For simplicity, you can consider the dataset as a two column table where the first column contains the letters and the other one the integers.

The task is to multiply each number by 10, then for each letter report the sum of the numbers associated with it. The MapReduce solution for this task is very simple. We only need to consider the letters as key and the numbers as values. Therefore, the mappers perform the multiplications, then the sorter and shuffler group the resulting numbers according to the keys (i.e., the records with similar letters) and finally the reducer sums up the results in each group and report them.




*MapReduce Numeric Example*

# Classic Word Count Problem

Assume we have a large text document and want to find the frequency of each single word in this document. Also, assume the file is already divided into the blocks and stored in a few DataNodes. Now, our job is to write the map and reduce functions to count the number of occurrence of each word.

The traditional way to solving this problem is to concatenate all the blocks to create the original large text document. Then, scan the file line by line. If we observe a new word, we simply add it to a dictionary and count it as one occurrence. If we see a word that we saw before, we only need to increase its number of occurrence.

The implementation of the above approach is very simple. However, it does not leverage the parallel processing at all. This means, the execution of such program, regardless of the efficiency of the language and the power of processors, would take a long time. Therefore, instead of taking those steps, we want to find a solution than can be executed in parallel. In other words, we are looking for an algorithm that can

do most of the work for each block of text in separate with the other blocks. This way, we can maximize the leverage of memory and processors attached to each DataNode.

This is the simple MapReduce solution. We count the number each word appears in each block. This can be done without the need of communicating between the blocks. This is the part that the mappers handle. After the subsolutions (i.e., dictionaries of the words and their count for each block) have been developed, we only need to aggregate the dictionaries and sum the numbers for the similar words. This part is done by the reducer. In the following, we discuss map and reduce functions in more details.

## The Map Step

The input of each mapper is a block of text. Each block consists of a series of lines each of which has one or more words. Since we need to aggregate the word counts, we have to develop a dictionary (i.e., a collection of key-value pairs) which the words are the keys. Therefore, the output of the mapper is a list of the words it scanned. The value of these keys is simply 1. For example, if we have a line like `'Hello world!'` as the mapper input, the output would be `<'hello', 1>, <'world', 1>` and if we had `'Hello the big big world!'` then the mapper returns `<'hello', 1>, <'the', 1>, <'big', 1>, <'big', 1>, <'world', 1>`.

The question emerges here is that why the value should be always 1, even when we scan a word more than once? The answer is that to perform all the operations in parallel, mappers do not care about the number of occurrences! Counting the words is the reducer's responsibility. Therefore, for each word a mapper sees, regardless of whether already beeing seen or not, it appends `<'word', 1>` to its dictionary. This is how our map function looks like:

```
class WordCountScala extends MapReduceBase with Mapper[LongWritable, Text, Text,
IntWritable] {
    val one = new IntWritable(1);
    val word = new Text();

    def map(key: LongWritable, value: Text, output: OutputCollector[Text,
IntWritable], reporter: Reporter): Unit = {
    var line = value.toString();
    line.split(" ").foreach(a => {word.set(a);output.collect(word, one);});
    }
}
```

## The Reduce Step

Before executing the reduce function, the mappers returned several dictionaries. Each dictionary is a collection of key-value pairs that the keys are the words (might be repetitive) and the values are always 1. The reducer job is to group all pairs with the same key, and then summarize their values. Since the values are always 1, then the sum of the values is the number of time that the mappers scanned that particular word. Our reduce function looks like the following code:

```
public static class Reduce extends MapReduceBase implements Reducer<Text,
IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
    OutputCollector<Text, IntWritable> output, Reporter reporter)
    throws IOException {
        int sum = 0;
```

```
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

**Activity 1:** Write a complete Scala program that reads a text file from HDFS, counts the words and returns the word counts in another text file in HDFS. Feel free to use the above sample codes.

***Hint:*** *You need to write a config class that calls your mapper and reducer and then run the jobs. A working solution is available here (https://github.com/milesegan/scala-hadoop-example). More comprehensive examples are also available here (https://github.com/deanwampler/scala-hadoop) and here (https://github.com/derrickcheng/ScalaOnHadoop).*

# 3
# Big Data Processing Technologies

## Module Learning Objectives

At the end of this module, students should be able to:

- Distinguish and discuss the well-known available technologies for Big Data that are developed based on (or on top of) Hadoop platform
- Compare different SQL overlays for Hadoop (eg, Hive, and Spark SQL) and evaluate
- Explain key concepts such as distributed transactions and concurrency control
- Perform basic ETL tasks using HBase/ technology

## Assessments

- Quizzes that target the students' understanding of the Hadoop framework and related technologies
- Develop/Complete/Interpret one or more SQL tasks for distributed and unstructured data
- Assignment 2

## Resources

- Apache HBase, Drill, Hive, Spark SQL Documentation
- FIT3143 lecture notes on distributed transactions and concurrency control (D Abramson et al)
- Chapter 2, Mastering Scala Machine Learning - Alex Kozlov: Data pipelining and modelling; basic components of a data-driven system
- Big Data teaching VM for labs and assignments

# 3.1
# Structured and Unstructured Data

In the previous modules, we discussed the Hadoop framework and the MapReduce programming approach which are the foundations of many Big Data technologies. In this module, we introduce some scalable database concepts and practice Hbase as a Big Data processing tool. We also discuss how distributed transactions and concurrency control work.

Before going deep into these concepts and tools, it is important to understand the differences between structured and unstructured data and the values each one of them brings into today's projects.

## Structured Data

Structured datasets are usually presented as one or more tables. Each table is a collection of rows and columns. The column of a table may or may not have relationships with the other columns in other tables (that is where the term ***relational databases*** comes in). In structured datasets, the data elements are stored in the tables' cells such that each row is considered as one data entry (a.k.a. datapoint, sample, or observation).

Spreadsheet (https://en.wikipedia.org/wiki/Spreadsheet) (e.g. Microsoft Excel (https://office.live.com/start/Excel.aspx?) and Google Sheets (https://www.google.com.au/sheets/about/)) is possibly the most famous example of a structured data representation. Other well-known examples are the ***Relational DataBase Management Systems*** (RDBMS (https://en.wikipedia.org/wiki/Relational_database_management_system)) such as Oracle (https://www.alexandriarepository.org/wp-content/uploads/20160819164034/index.html), MySQL (https://www.mysql.com/), PostgreSQL (https://www.postgresql.org/) and Microsoft SQL Server (https://www.microsoft.com/en-au/sql-server). To create a structural database or import data to a spreadsheet/table the data structure or schema must be fully and precisely defined a priori. Then, we can populate the dataset gradually or at once. To read, insert, update or remove the data in a structural database, the spreadsheet ***lookup*** or ***Structured Query Language*** (SQL) calls should be made.

## Unstructured Data

Unstructured data may be coming in many different forms and formats. Notably, audios and videos, books, analog signals and machine logs all are instances of unstructured data types. Row-based processing frameworks such as spreadsheets and traditional RDBMS tables become difficult and highly inefficient. Columnar (https://en.wikipedia.org/wiki/Column-oriented_DBMS) storage and processing tools are designed to ease this process and enhance the performance of the traditional systems.

Note that the applications of columnar databases are not limited to unstructured data. We can store structured data in a columnar database if required. We will discuss the columnar storage and processing in more details. However, before jumping into the details of scalable DBMS, we should understand why we need a database management system while we already have Hadoop distributed file system which is efficient and effectively scalable.

# 3.2
# Distributed Transactions

Previously we learned that Hadoop provides a platform to work with large datasets in a distributed fashion. We also know that its architecture is highly fault-tolerant and linearly scalable. We also briefly introduced several tools for managing databases that built on top of Hadoop platform. The question emerges here, regarding the Hadoop capabilities, why we need to acquire these tools?

Let's take one step back and see where we need a database management. In general, we use DBMS for transaction processing. A transaction involves creating, reading, updating, deleting (a.k.a. CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) operations) or a combination of them. These transactions have certain requirements that the core Hadoop cannot satisfy (some of) them.

The limitations that make Hadoop unsuited for transaction processing are not supporting structured data, no random access, high latency and not ACID compliant. In the following, we discuss each one of these limitations.

## Hadoop Distributed File System Limitations

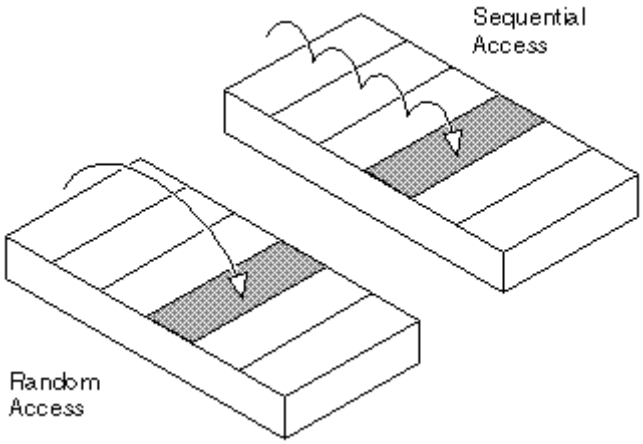
### Structured Data and HDFS

As we discussed earlier, Hadoop stores the data in HDFS clusters. HDFS is a schemaless technology which means it treats all types of data as if they were unstructured. Therefore, even when we store JSON or CSV files which have rich structures, HDFS has no idea about their schema so does not acknowledge rows, columns and the relationship between them. Recall that HDFS sees the data as chunks of files (no recognition of rows or tables).

In addition, HDFS imposes zero constraints on the stored data. Therefore, we cannot enforce any constraints such as column type, not-null or unique properties at the time of creation, insertion or updating the datasets.

### Random Access

Random or direct access refers to the ability to access any part of a record (e.g. row/cell of a table) of the dataset. For example, changing the address of an employee or querying the title of an email both need direct access to those entries. Transactional databases should be able to directly access any item in the dataset regardless of its size in a timely manner. The following diagram compares random access with sequential access.




*Random vs Sequential Access (Source: Webopedia)*

As we know, the finest granularity in HDFS is the file chunks, which means the rows and cells are not directly addressable and accessible. In fact, HDFS only supports sequential access. Therefore, we need to process the whole dataset to access a particular part of a file that we are interested, which is computationally expensive. The data manipulation, in this case, is even more challenging.

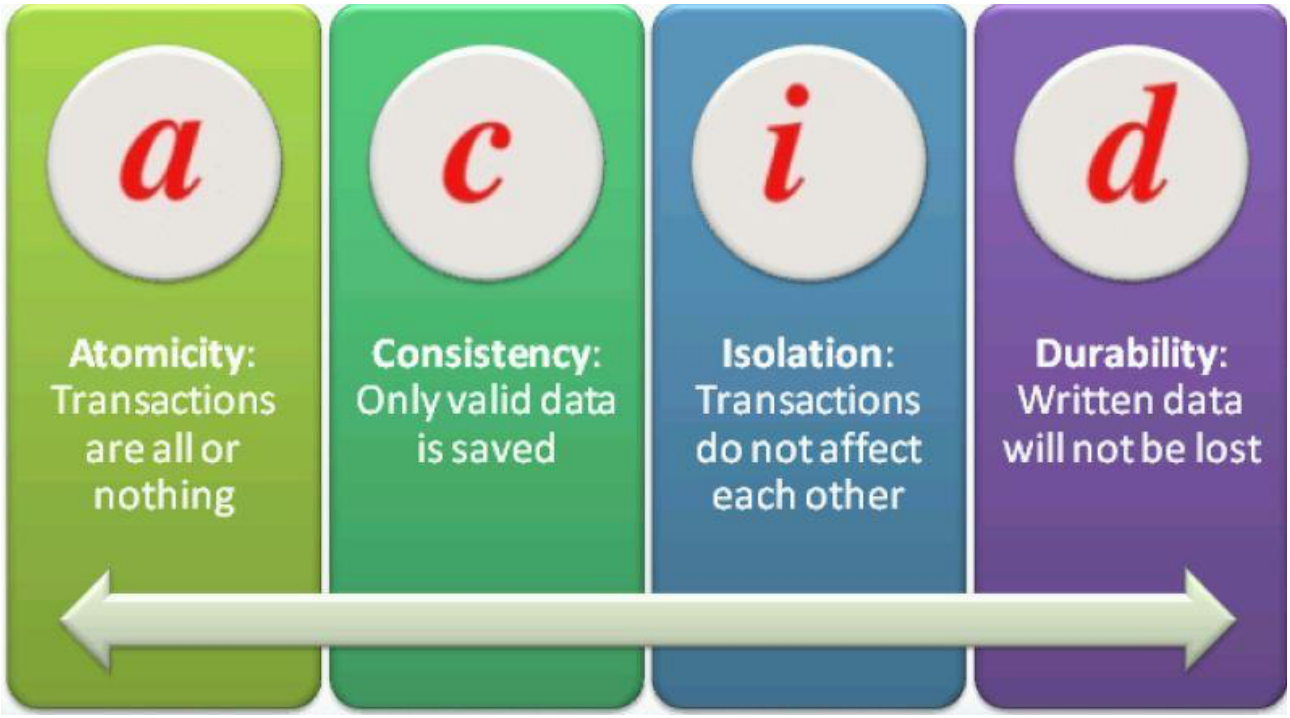
## Latency

In transaction processing, we need to perform CRUD operations as fast as possible. For example, we may need to update the status of an item from available to sold or search our large dataset to find the best flights according to specified preferences. Ideally, such operations should not take more than a few seconds to complete.

However, as discussed above, the sequential access implemented in HDFS does not allow us to perform CRUD operations in a timely manner. In fact, MapReduce is very efficient in processing the whole datasets, but not such transactional processes. In practice, these tasks may take minutes to hours to complete even using large clusters.

## ACID

Databases must guarantee ACID (https://en.wikipedia.org/wiki/ACID) properties to always maintain the integrity of the stored data. The term ACID stands for Atomicity, Consistency, Isolation, Durability.




*ACID diagram (source: Information Technology Sales)*

## Atomicity

In the database context, atomicity (https://en.wikipedia.org/wiki/Atomicity_(database_systems)) implies that the transactions regardless of their number of components and subcomponents must be all-or-nothing. For example, consider a cash withdrawal from an ATM machine. This transaction consists of two smaller subtransactions: 1) updating the cash balance and 2) updating the account balance. Obviously, if any of these two subtransactions fails, the whole transaction is failed. Therefore, the other successful subtransaction should be rolled back.

## Consistency

The consistency (https://en.wikipedia.org/wiki/Consistency_(database_systems)) in the database systems means any changes to the database must not violate any specified constraints. This guarantees all transactions should be validated against the rules and constraint defined by the designer before committed to the dataset.

## Isolation

It is very common for a database system to perform the transactions in parallel to leverage maximum benefit from the processing and networking facilities. This concurrent execution of transaction should be carried out in a way to not violate the system rules. The isolation (https://en.wikipedia.org/wiki/Isolation_(database_systems)) or serializability property guarantees that the results of these concurrent executions are exactly the same as if the transactions were executed sequentially.

In database systems, the concurrency control (https://en.wikipedia.org/wiki/Concurrency_control) comprises the underlying mechanisms that guarantee the isolation property. These mechanisms should be designed in a way to minimize the overhead that is enforced by introducing more constraints on the operations. Generally, there are three categories of concurrency control algorithms: using locks, using timestamps, and the optimistic approach.

## Lock-based Concurrency Control

This is the oldest and most widely used concurrency control algorithm. When a process needs to access shared resource as part of a transaction, it first locks the resource. If the resource is locked by another process, the requesting process waits until the lock is released.

The basic scheme of this algorithm is restrictive. However, it can be improved by distinguishing read locks from write locks. Therefore, the data to be read (referenced data) is locked in the shared mode. On the other hand, the data to be updated (modified data) is only locked in the exclusive mode.

Transactions can lock a resource in two cases:

1. If a resource is not locked or locked in the shared mode, a transaction can lock it in only shared mode.
2. If the resource is currently not locked, the transaction can lock it in the exclusive mode.

The following table summarizes the above-mentioned notes on a Lock-based concurrency control system. The table illustrates whether a lock (Shared or Exclusive) should be granted to the transactions.

| | **Shared Lock** | **Exclusive Lock** |
|---|---|---|
| **SHARED LOCK** | TRUE | FALSE |
| **EXCLUSIVE LOCK** | FALSE | FALSE |

As you can see in the above table, a transaction may be granted a lock on an item, if the requested type of lock is compatible with the locks which are already possessed on that item by other transactions. This means, for example, any number of transactions can hold shared locks on an item. However, if any transaction holds an exclusive lock on an item, no other transaction can hold any kind of lock (Shared or Exclusive) on that particular item, until the transaction releases the exclusive lock.

## Using Timestamps Concurrency Control

In this approach, we assign a timestamp at the beginning of the transaction to it. Each resource also has a read and write timestamps. The timestamps are unique in all circumstances. If a transaction needs to read a value of a resource that is already overwritten, the request will be rejected and the transaction is aborted. Otherwise, the transaction is allowed to read the resource.

When a transaction attempts to write an obsolete value of a resource that is already overwritten, the request will be rejected and the transaction is aborted. On the other hand, if the transaction tries to write on a resource that already was read, then the value of the resource that the transaction is producing was needed previously, and the system assumed that value would never be produced. Therefore, the request is rejected and the transaction is aborted. In none of these cases occurs, the transaction is allowed to update the resource.

Put it simply, there is a simple rule in granting the execution of a given transaction $T_i$ (known as the Thomas' write rule (https://en.wikipedia.org/wiki/Thomas_write_rule)): If transaction $T_i$ wants to get access to the item X, then the operation is granted if `TS(Ti) >= W-timestamp(X)`; otherwise, the operation is rejected! It means that the access to the item X for the transaction $T_i$ is granted only if the timestamp of transaction $T_i$ is greater than or equal to the write operation on item X.

There is also a Timestamp ordering protocol as the following (https://en.wikipedia.org/wiki/Timestamp-based_concurrency_control):

- **If a transaction Ti issues a read(X) operation −**
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  - If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation −**
  - If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  - Otherwise, Operation executed.

## Optimistic Concurrency Control

In this approach, the transactions are optimistic about conflicts. At the end of each transaction, when it is committing into the database, the existence of conflict is examined. If no conflict is detected then the local copies are written on the real resource. Otherwise, the transaction is aborted.

# Durability

The durability (https://en.wikipedia.org/wiki/Durability_(database_systems)) property guarantees that once a transaction is executed, the changes in the database are permanent. Therefore, no failure after the commit can undo the results or cause these to be lost.

Overal, ACID guarantees require that the database management system is fully aware of the structure of the data and protects the integrity of its content. However, since HDFS is only a file storage system, it has no such awareness. Thus, Hadoop by itself is not an ACID compliant system. For these reasons, we need some database management systems that not only satisfy all the requirements but also horizontally scale.

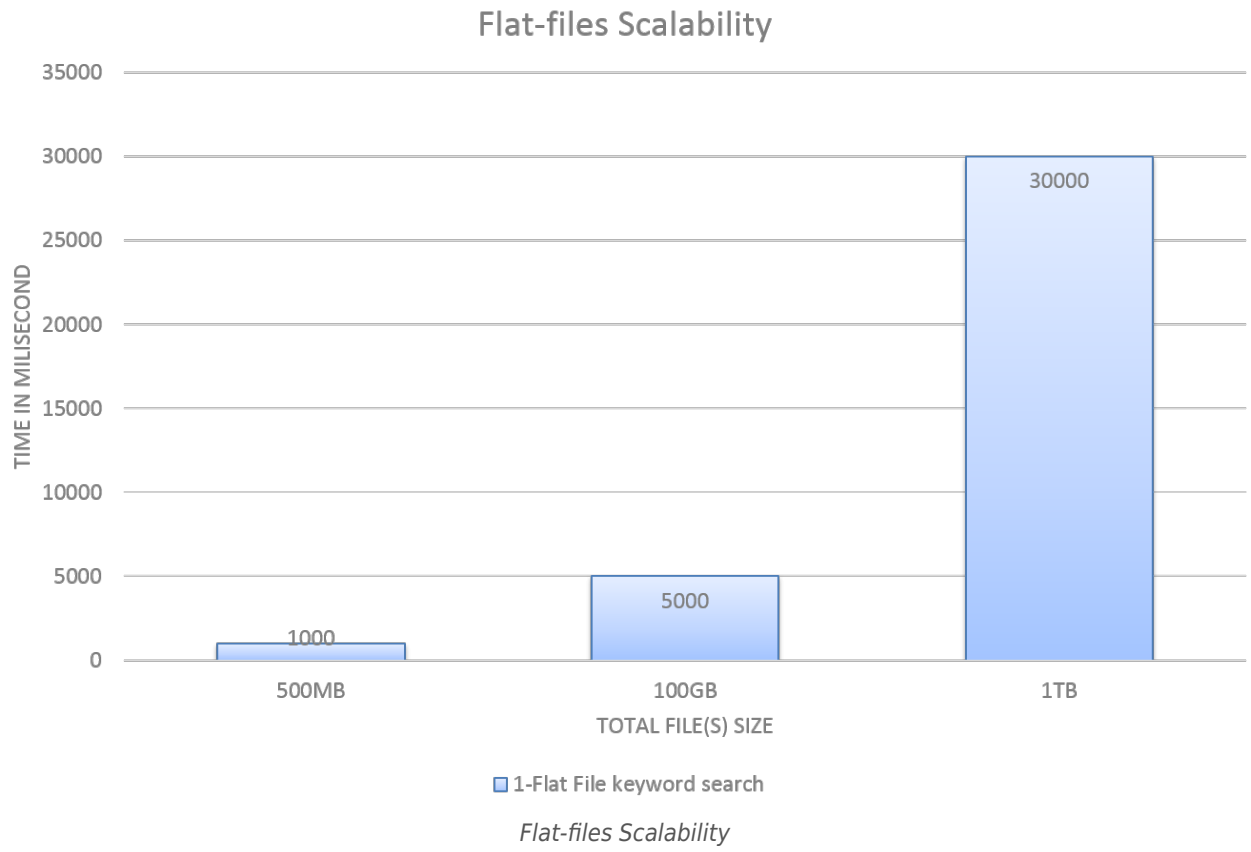
# 3.3
# Columnar Databases

## Introduction

Previously we discussed the necessity of having database management systems in general as the HDFS does not support some of the main requirements to perform CRUD operations in a timely manner. Here, we explain why the traditional databases are not efficient enough. Next, we introduce ideas that help us to design a new system that scales well. Finally, we review the history of such systems in brief.

### The rise of Big Data

Traditionally, we used to store structured data in spreadsheets or Relational Databases (RDBs) and unstructured datasets in one or more flat files. However, the rise of Big Data makes using these old technologies rather difficult. For example, the following plot shows that how long it takes to search a keyword in one flat-file dataset as the size of the file grows.

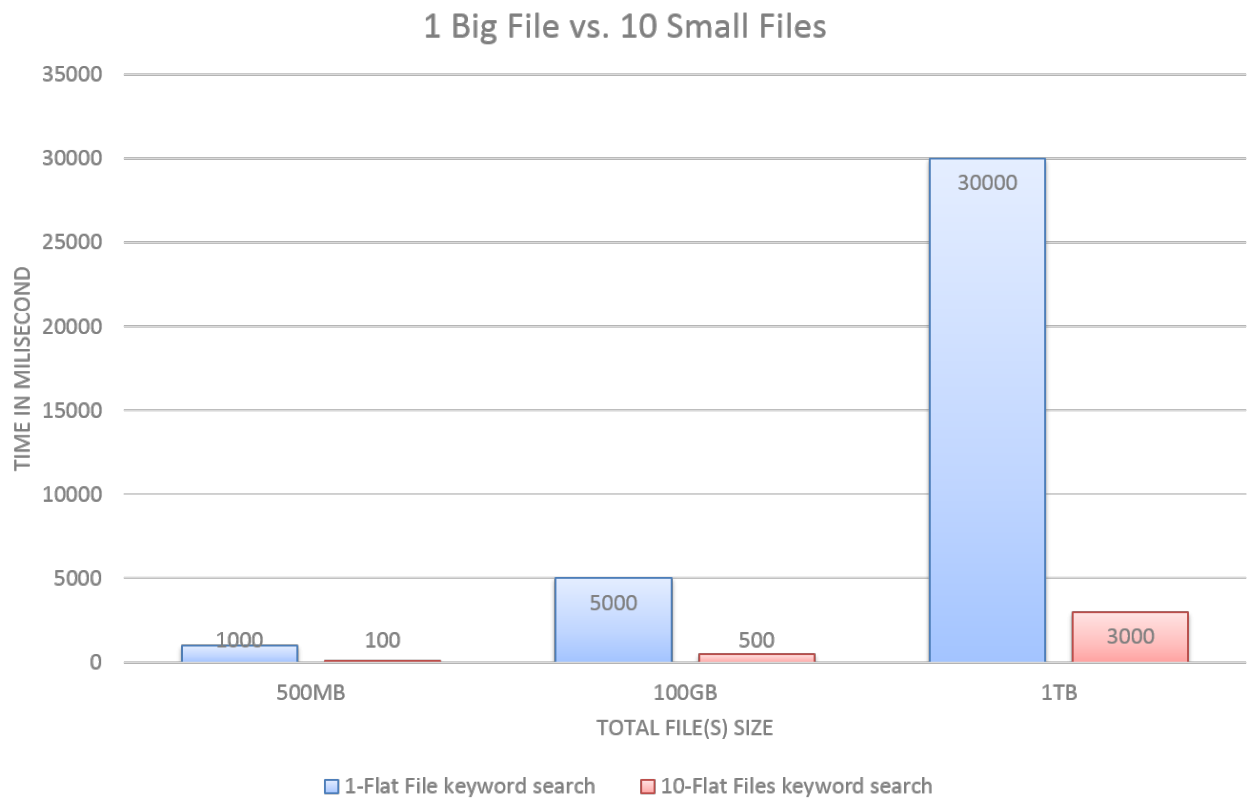



*Flat-files Scalability*

As the figure shows, the main efficiency concern of storing data in large flat-files is the speed which is limited by hard disk access time. While lifting this physical barrier might be expensive, if possible, we can tweak the way we store the data to maximize our leverage from the current hard disk technology.

One of the simplest solutions is to store the data in more than a file. Indeed, we can break to original

large file to a number of smaller files and ideally store each one of them on a separate machine. The following barplot shows how much speed enhancement we theoretically can gain by this simple *divide-and-conquer* approach:




*1 Big Flat-file vs 10 Smaller Flat-files*

In the previous modules, we discussed the Hadoop Distributed File System that can be used in the above scenario. Indeed, the MapReduce programming approach can also be adopted to significantly increase the data processing speed.

## Row-Oriented vs. Column-Oriented

Another adjustment that we can make to practically improve the query performance in very large data sets without spending a large portion of our budget on the storage devices is the use of columnar or column-oriented databases instead of traditional row-oriented approach. Traditionally, databases store the structured data row by row. This approach can be very effective when the dataset is manageable such that the data can be read/write a limited number of hard disk seeks. For example, assume we want to store the following table:

| rowID | boxID | itemName | itemName2 | boxPrice |
|---|---|---|---|---|
| 001 | A001 | Apple | Green | 100 |
| 002 | A002 | Apple | Red | 75 |
| 003 | B001 | Banana | (null) | 120 |

In a row-oriented database, the above table would be stored as:

```
001: A001, Apple, Green, 100;
002: A002, Apple, Red, 75;
003: B001, Banana, ,120;
```

Typically, for reading each row of the above table we need one hard drive seek. For example, to calculate the sum of `boxprice column`, for example, we need three seeks. This overhead is not very challenging for a small dataset like this example. However, if we have billions of rows, then such summary queries would be very time-consuming. The solution for this type of queries is the column-oriented database. The same table in a columnar database would be stored as

```
A001:001, A002:002, B001:003;
Apple:001, Apple:002, Banana:003;
Green:001, Red:002;
100:001, 75:002, 120:003;
```

As the above example shows, all the first column values are physically stored together, followed by all the second column values, etc. This allows individual data elements (e.g. `boxprice)` to be accessed in one seek event *almost* regardless of the number of rows. This approach can increase the speed of the summarization and visualization queries, which are very common in Big Data processing tasks, to a great extent.

Another benefit of the column-based DBMSs is that data can be highly compressed. The compressed datasets not only consume less storage space, but also allow rapid columnar operations (e.g., `min`, `max`, `sum`, `count`, and avg). Yet another advantage is that the columnar storage is self-indexing. It means we do not need to spend a considerable amount of valuable space to store the indexes, which results in less disk space usage than a row-oriented RDBMS containing the same indexed data.

Although the efficiency of columnar DBMSs in OLAP-like operation is very great, they suffer from shortcomings in OLTP-like tasks. In other words, this approach is not as efficient as the row-based techniques for ordinary transactional queries (e.g., inserting a new row or updating the value of a cell). Therefore, it is very common to have multiple DBMS, both row-oriented and column-oriented, to work together on a common dataset. For further information on OLAP and OLTP and their operations, please refer to the following resources: OLAP1 (http://olap.com/olap-definition/), OLAP2 (https://en.wikipedia.org/wiki/Online_analytical_processing), and OLTP (https://en.wikipedia.org/wiki/Online_transaction_processing).

## A Short History

The TAXIR (http://www.mombu.com/programming/t-taxir-the-first-column-oriented-database-in-1969-paper-11968851.html) system is known as the first application of a columnar DBMS that was released in 1969. Soon after, RAPID (https://www.dbbest.com/blog/column-oriented-database-technologies/) was developed by Statistics Canada to store and process the Canadian Census of Population and Housing data. The first commercially available columnar DBMS is known to be KDB, the predecessor of KDB+ (https://kx.com/solutions/) (the first in-memory column-oriented DBMS) developed during 1993-95.

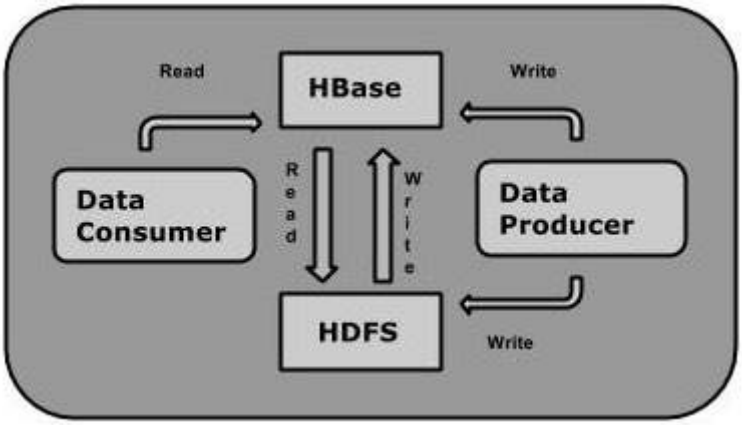
MonetDB (https://www.monetdb.org/Home), an extension of Troll (https://www.monetdb.org/AboutUs) that was developed during 1979-92, is the first open-source columnar DBMSs that was released in 2004. At the same time, Google was busy developing its own columnar storage system that can handle a huge volume of data, called BigTable (https://cloud.google.com/bigtable/). Recall that, the Hadoop HDFS and MapReduce designs are also originated from the earlier work by Google. All of these technologies were designed to work in a unified ecosystem. The BigTable, similar to other columnar DBMSs, was designed to map row-key, column-key, and timestamp into one associated byte array. A year later, in 2005, Vertica (https://en.wikipedia.org/wiki/Vertica) was eventually developed and in 2011 acquired by HP (https://www.vertica.com/). This is the year that the famous Druid (http://druid.io/druid-powered.html) was born and very soon is adopted by technology companies such as Alibaba, Airbnb, Cisco, eBay, Netflix, Paypal, and Yahoo!.

In 2006, the development of HBase (https://hbase.apache.org/) as a tool that provides a BigTable-like capability over Hadoop core (i.e., HDFS and MapReduce) started. In 2008, Apache acquired the project and since 2010 it has been one of Apache top-level project. So far, HBase is adopted by many companies such as Adobe and Spotify.

In the next chapter, we discuss HBase as one of the open-source columnar DBMS that is widely used in the Big Data projects.

# 3.4 Introduction to HBase

Apache HBase, as mentioned earlier, is a distributed, versioned, NoSQL DBMS which provides Bigtable-like functionality on top of Hadoop MapReduce and HDFS. The following depicts the relationship between HBase and HDFS.




*HBase and HDFS (source: tutorialspoint.com)*

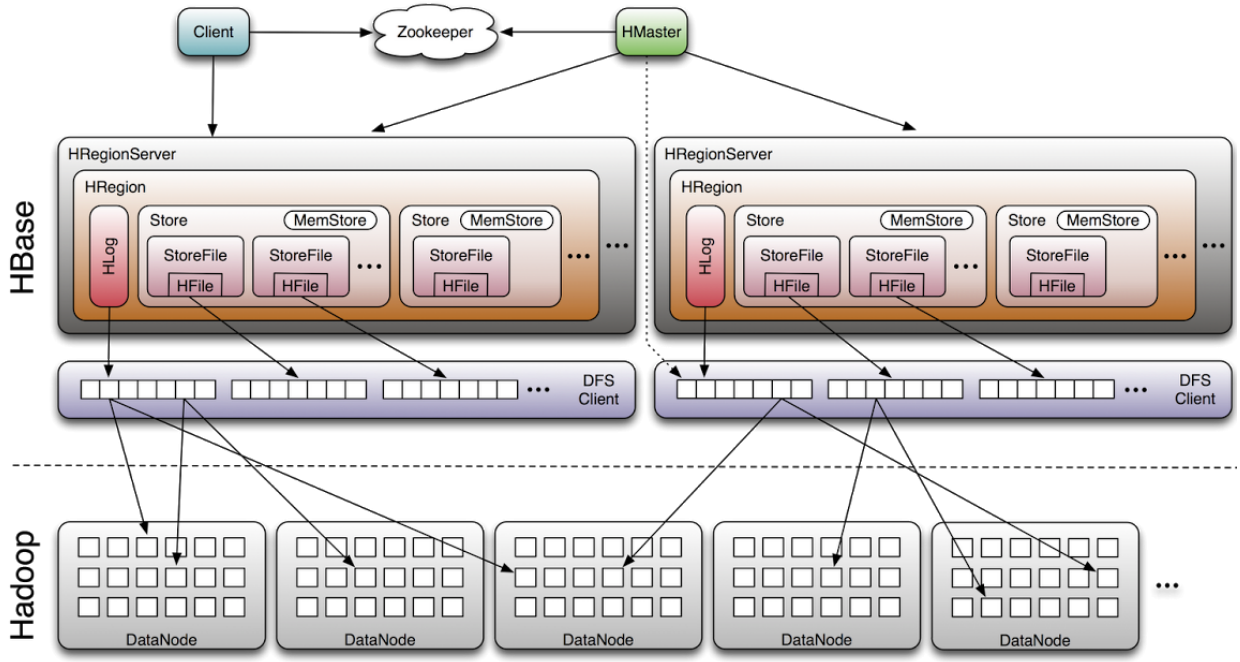
As the above diagram shows, one can store the data in HDFS either directly or through HBase (and of course many other tools). Recall that HDFS is just a distributed file system while HBase is a DBMS on top of this file system. HBase provides faster lookups for larger tables which means the data consumer can access to any single row from billions of records with low latency. In other words, HBase can be used as a proxy to access the data stored on HDFS.

## HBase Architecture

HBase framework (https://mapr.com/blog/in-depth-look-hbase-architecture/) comprises the following main components:

1. **Master Server (HMaster):** The master server which runs on the *namenode* monitors all instances of the region servers. It also acts as the common interface for all metadata (i.e., schema) changes made within the cluster.
2. **Region Servers (HRegion):** The region servers are responsible for managing HBase regions, each of which comprises a subset of rows. A region may be split as the upper limit for row allocation per region is exceeded. The region servers usually run on the *datanode*. Recall that the datanode stores the file and their replicas in Hadoop clusters. Region servers have three main components:
    1. **Block Cache:** It is used for fast in-memory access to more frequently read data items
    2. **MemStore:** It is a fast in-memory storage for the transactions that have not committed yet.
    3. **Write-Ahead-Log (WAL):** Region server updates are written to WAL first, and then to MemStore to ensure a durable write. Without WAL, there is the possibility of data loss in the case of a region server failure.
3. **Catalog Tables (HLog):** A catalog table stores the metadata and contains information on data distribution among the region servers such as when a new region is formed or a region splits in two or more. The locations of catalog tables are stored in the Zookeeper. There are two types of catalog table:

    1. **Root Table:** It keeps track of the location of Meta table.
    2. **Meta Table:** It stores a list of all regions in the system.
4. **Zookeeper:** It is a distributed resource management system which implements a simple interface to provision a centralized coordination service comprising:
    1. Distributed configuration service
    2. Synchronisation service
    3. Distributed naming (space) registry
5. **Client (HTable):** The HBase client is responsible for finding region servers that are serving the particular row range of interest. At startup, the client uses zookeeper for obtaining the location of the relevant region server.




*HBase Architecture (source: larsgeorge.com)*

# HBase Tables

HBase effectively stores data as ***key-value*** pairs. Each HBase database is partitioned into one or more tables. The tables are further split into ***column families*** which are sets of related columns. For example, "address" column family may include the columns: "city", "street", "postcode", and "state".

The columns in a column family do not need to be logically subtopics of the main column family. In fact, we group columns that usually appear in a single query. For instance, since "to", "from", "date", "time", and "title" of an email are usually queried together, we may group them as "message" column family.

Each key-value pair in HBase is defined as a cell. Each key consists of row-key, column family, column, and timestamp. A row in HBase is a grouping of key-value mappings identified by the row-key. Rows are lexicographically sorted with the lowest order appearing first in a table. The start and end of a table are marked with an empty byte array.

## Activity: Getting Started

The virtual machine provided with this unit contains a fully configured HBase instance. We took the

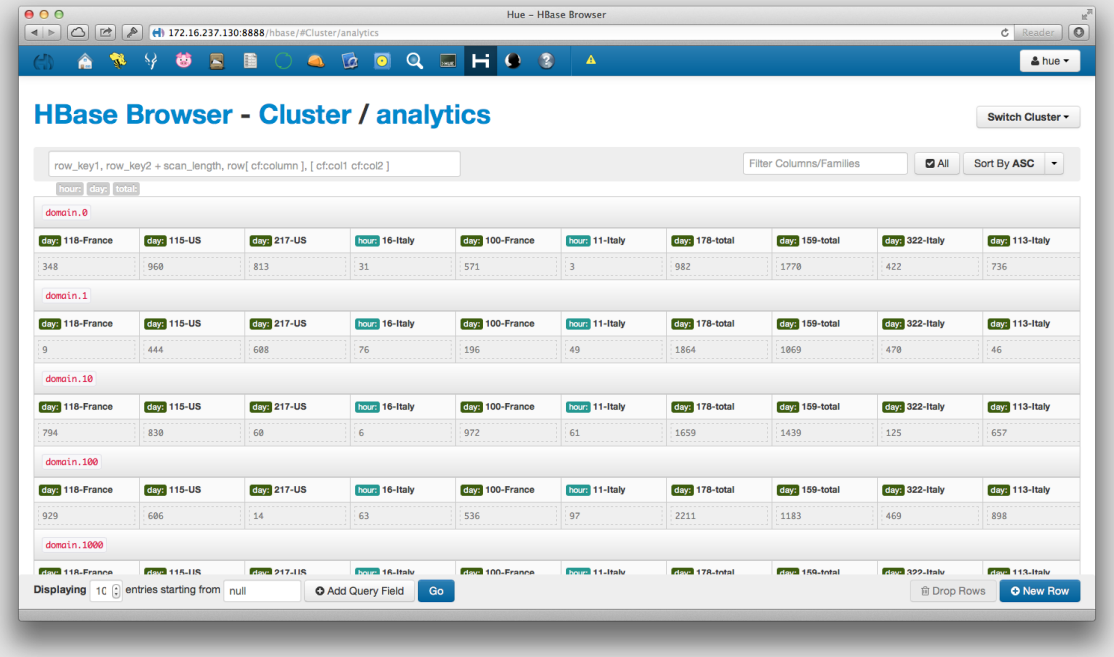
following steps to setup this instance:

1. Downloading and expanding HBase distribution archive in the user's home directory
2. Setting up and exporting JAVA_HOME environmental variable
3. Editing HBase's site configuration and setting the working directory paths for HBase and Zookeeper processes to write their outputs
4. The VM also comes installed with Hadoop core, which should be started before starting the HBase instance

To start the environment first call `start-dfs.sh` and then `start-hbase.sh`. Now, you can try some general commands such as `status`, `version`, and `whoami` to see some information about the number of servers, HBase version, and user. To stop them, call `stop-hbase.sh` first, and then `stop-dfs.sh`

## HBase Shell vs HUE

When it comes to working with HBase, there are two major interfaces: the HBase Shell and HBase Browser. Hadoop User Experience, or HUE (http://gethue.com/), is an open-source Web interface that supports Hadoop and some parts of its ecosystem. It is developed by Cloudera but licensed under the Apache license. Big Data scientist may use HUE as a UI (https://www.youtube.com/watch?v=BY9P1K34xBg) for querying HBase (http://gethue.com/category/hbase/). Although HUE is a powerful and widely-used tool, it is still not as flexible as HBase original Shell. Therefore, we do not cover HUE interface in this chapter. Instead, in the next section, we discuss some of the basic operations that can be performed using both interfaces




*HUE HBase Browser screenshot (Source: Cloudera)*

# 3.5
# HBase Shell Commands

## HBase Commands

In the following, we briefly introduce the HBase most commonly used commands that are categorized into Data ***Definition***, Data ***Manipulation***, and Data ***Validation*** commands.

### 1- Data Definition Commands

These commands operate on the tables in HBase to create, alter or drop a table.

- Create a table:

```
create '<table name>','<column family>'
```

- Enable/disable a table:

```
enable '<table name>'
disable '<table name>'
```

- Changes to an existing table. For example, one can change the maximum number of cells of a column family, set/delete table scope operators, and delete a column family from a table.

```
% changing the maximum number of cell to N:
alter '<table name>', NAME => '<column family>', VERSIONS => N
% delete a column family from a table:
alter '<table name>', 'delete' => '<column family>'
% makes the table read-only:
alter '<table name>', READONLY
```

- Drop a disabled table from HBase:

```
drop '<table name>'
```

- Drop the tables matching the 'regex' given in the command. The tables should be disabled before dropping them.

```
drop_all 'regular expression'
```

### 2- Data Manipulation Commands

The following commands manipulate (e.g., insert, update, delete) the data in the tables.

- Insert a row to an existing table:

```
% inserts a new row to the specified table
put '<table name>','row','<colfamily:colname>','<value>'
```

- Set a cell value at a specified column in a specified row in a particular table.

```
% updates one row of the specified table:
```

```
put '<table name>','row','<colfamily:colname>','<new value>'
```

- Fetch the contents of a row or cell.

```
% reads a specific row:
get '<table name>','row'
% reads a specific cell:
get '<table name>', 'rowid', {COLUMN => 'column family:column name'}
```

- Delete a cell value in a table.:

```
delete '<table name>', '<row>', '<column name >', '<time stamp>'
```

- Delete all the cells in a given row of a table:

```
deleteall '<table name>', '<row>'
```

- Disable, drop, and recreate a specified table, all at once:

```
truncate '<table name>'
```

## 3- Scan and Validation Commands

- List all the tables previously created in HBase.

```
list
```

- Return the table data.

```
scan'<table name>'
```

- Count the number of rows in a table.

```
count '<table name>'
```

- Verify whether a table exists.

```
exists '<table name>'
```

- Return the description of a table.

```
describe '<table name>'
```

- Verify whether a table is enabled/disabled.

```
is_enabled '<table name>'
is_disabled '<table name>'
```

# 3.5.1
# Activity 1: Hands on HBase

In this activity, we intend to become more familiar with HBase shell by practicing its basic commands in some simple but common scenarios.

## Basic HBase Exercises

### 1. Run Services

First things first! Let's start the required services which are Hadoop, YARN, and HBase. To do that, we need to run start-dfs.sh, start-yarn.sh, and start-hbase.sh in this exact order. The outputs would be something similar, but not exactly the same, to the followings:

```
user@ubuntu:~$ start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/user/hadoop-2.7.3/logs/hadoop-
user-namenode-ubuntu.out
localhost: starting datanode, logging to /home/user/hadoop-2.7.3/logs/hadoop-
user-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to
/home/user/hadoop-2.7.3/logs/hadoop-user-secondarynamenode-ubuntu.out
user@ubuntu:~$ start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /home/user/hadoop-2.7.3/logs/yarn-user-
resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /home/user/hadoop-2.7.3/logs/yarn-
user-nodemanager-ubuntu.out
user@ubuntu:~$ start-hbase.sh
localhost: starting zookeeper, logging to
/home/user/hbase-1.2.4/bin/../logs/hbase-user-zookeeper-ubuntu.out
starting master, logging to /home/user/hbase-1.2.4/bin/../logs/hbase-user-
master-ubuntu.out
starting regionserver, logging to /home/user/hbase-1.2.4/bin/../logs/hbase-
user-1-regionserver-ubuntu.out
user@ubuntu:~$
```

### 2. Connect to HBase

Now we can connect to our running instance of HBase using the `hbase shell` command, located in the `bin/` directory of the HBase. The HBase Shell prompt ends with a > character:

```
user@ubuntu:~$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.4, r67592f3d062743907f8c5ae00dbbe1ae4f69e5af, Tue Oct 25 18:10:20
CDT 2016
```

```
hbase(main):001:0>
```

## 3. Display HBase Shell Help Text.

To display some basic usage information for HBase Shell and several example commands, we can simply type `help` and press Enter:

```
hbase(main):001:0> help
HBase Shell, version 1.2.4, r67592f3d062743907f8c5ae00dbbe1ae4f69e5af, Tue Oct
25 18:10:20 CDT 2016
Type 'help "COMMAND"', (e.g. 'help "get"' -- the quotes are necessary) for help
on a specific command.
Commands are grouped. Type 'help "COMMAND_GROUP"', (e.g. 'help "general"') for
help on a command group.

COMMAND GROUPS:
  Group name: general
  Commands: status, table_help, version, whoami

  Group name: ddl
  Commands: alter, alter_async, alter_status, create, describe, disable,
disable_all, drop, drop_all, enable, enable_all, exists, get_table, is_disabled,
is_enabled, list, locate_region, show_filters

  Group name: namespace
  Commands: alter_namespace, create_namespace, describe_namespace,
drop_namespace, list_namespace, list_namespace_tables

  Group name: dml
  Commands: append, count, delete, deleteall, get, get_counter, get_splits,
incr, put, scan, truncate, truncate_preserve

  Group name: tools
  Commands: assign, balance_switch, balancer, balancer_enabled,
catalogjanitor_enabled, catalogjanitor_run, catalogjanitor_switch, close_region,
compact, compact_rs, flush, major_compact, merge_region, move, normalize,
normalizer_enabled, normalizer_switch, split, trace, unassign, wal_roll, zk_dump

  Group name: replication
  Commands: add_peer, append_peer_tableCFs, disable_peer,
disable_table_replication, enable_peer, enable_table_replication, list_peers,
list_replicated_tables, remove_peer, remove_peer_tableCFs, set_peer_tableCFs,
show_peer_tableCFs

  Group name: snapshots
  Commands: clone_snapshot, delete_all_snapshot, delete_snapshot,
list_snapshots, restore_snapshot, snapshot

  Group name: configuration
  Commands: update_all_config, update_config
```

```
  Group name: quotas
  Commands: list_quotas, set_quota

  Group name: security
  Commands: grant, list_security_capabilities, revoke, user_permission

  Group name: procedures
  Commands: abort_procedure, list_procedures

  Group name: visibility labels
  Commands: add_labels, clear_auths, get_auths, list_labels, set_auths,
set_visibility

SHELL USAGE:
Quote all names in HBase Shell such as table and column names.  Commas delimit
command parameters.  Type <RETURN> after entering a command to run it.
Dictionaries of configuration used in the creation and alteration of tables are
Ruby Hashes. They look like this:

  {'key1' => 'value1', 'key2' => 'value2', ...}

and are opened and closed with curley-braces.  Key/values are delimited by the
'=>' character combination.  Usually keys are predefined constants such as
NAME, VERSIONS, COMPRESSION, etc.  Constants do not need to be quoted.  Type
'Object.constants' to see a (messy) list of all constants in the environment.

If you are using binary keys or values and need to enter them in the shell, use
double-quote'd hexadecimal representation. For example:

  hbase> get 't1', "key\x03\x3f\xcd"
  hbase> get 't1', "key\003\023\011"
  hbase> put 't1', "test\xef\xff", 'f1:', "\x01\x33\x40"

The HBase shell is the (J)Ruby IRB with the above HBase-specific commands added.
For more on the HBase Shell, see http://hbase.apache.org/book.html
```

As the above guide mentions, the table names, rows, columns all must be enclosed in quote characters.

## 4. Create a Table

WE can create our first table simply using the `create` command. The name of the table and the column-family name(s) must be specified. For example:

```
hbase(main):002:0> create 'tableName', 'colFamily1'
0 row(s) in 1.5100 seconds

=> Hbase::Table - tableName
```

## 5. Information About Tables

Now we created our first table `tableName`. Let's see how much HBase knows about our table:

```
hbase(main):003:0> list 'tableName'
TABLE
tableName
1 row(s) in 0.0290 seconds

=> ["tableName"]
```

Note that issuing list command with no argument will list all created tables.

To see more information about our table and how HBase manages it, let's call `describe` command:

```
hbase(main):004:0> describe 'tableName'
Table tableName is ENABLED
tableName
COLUMN FAMILIES DESCRIPTION
{NAME => 'colFamily1', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'fals
e', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVE
R', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE
=> '65536', REPLICATION_SCOPE => '0'}
1 row(s) in 0.0140 seconds
```

## 6. Insert Data

We use the put command to insert data into our table:

```
hbase(main):005:0>  put 'tableName', 'row01', 'colFamily1:A', 'value01'
0 row(s) in 0.2170 seconds

hbase(main):006:0>  put 'tableName', 'row02', 'colFamily1:B', 'value02'
0 row(s) in 0.0040 seconds

hbase(main):007:0>  put 'tableName', 'row03', 'colFamily1:B', 'value03'
0 row(s) in 0.0050 seconds
```

In the above example, we inserted three rows with three different values: `value01`, `value02`, valXue03. Note that all of these values are inserted in the column-family `colFamily1`, but into two different columns: A and B. Here, we inserted each of these values one at a time. There are other ways to insert a new row into a table that you can try:

```
hbase> put 'ns1:t1', 'r1', 'c1', 'value'
hbase> put 't1', 'r1', 'c1', 'value'
hbase> put 't1', 'r1', 'c1', 'value', ts1
hbase> put 't1', 'r1', 'c1', 'value', {ATTRIBUTES=>{'mykey'=>'myvalue'}}
hbase> put 't1', 'r1', 'c1', 'value', ts1, {ATTRIBUTES=>{'mykey'=>'myvalue'}}
hbase> put 't1', 'r1', 'c1', 'value', ts1, {VISIBILITY=>'PRIVATE|SECRET'}
```

In these examples, `ns1`, `t1`, `r1`, and `c1` are name-space, table, row and column names, respectively.

As the last three lines show, we can also set some cell attributes at the time of insertion if we wish.

## 7. Scan a Table

One of the ways to get data from HBase is to scan the whole table at once using the `scan` command. Obviously, it is not the best idea if we have a table with lots of rows. Fortunately, we can limit the scanning to rows that match a certain pattern or satisfy a condition. For now, let's fetch all available data:

```
hbase(main):008:0> scan 'tableName'
ROW                   COLUMN+CELL
 row01                column=colFamily1:A, timestamp=1496897587295,
value=value01
 row02                column=colFamily1:B, timestamp=1496897606345,
value=value02
 row03                column=colFamily1:B, timestamp=1496897613569,
value=value03
3 row(s) in 0.0370 seconds
```

As the printed results show, HBase not only displays the rows and columns information but prints the time of the insertion. Can you guess what happens when a cell value is changed multiple time?

Let's limit the scanning to only cells in column A. Recall that A in this example is a column under the column-family `colFamily01:`

```
hbase(main):015:0> scan 'tableName', {COLUMNS => 'colFamily1:A'}
ROW                   COLUMN+CELL
 row01                column=colFamily1:A, timestamp=1496897587295,
value=value01
1 row(s) in 0.0110 seconds
```

Is it what we expected? Yes! From the previous scan example, we learn that we have one cell under A and two other rows under column B. Let's see how we can set a limit on the number of displayed rows in scan command. Compare the output of the following example:

```
hbase(main):018:0> scan 'tableName', {COLUMNS => 'colFamily1:B'}
ROW                   COLUMN+CELL
 row02                column=colFamily1:B, timestamp=1496897606345,
value=value02
 row03                column=colFamily1:B, timestamp=1496897613569,
value=value03
2 row(s) in 0.0240 seconds

hbase(main):019:0> scan 'tableName', {COLUMNS => 'colFamily1:B', LIMIT => 1}
ROW                   COLUMN+CELL
 row02                column=colFamily1:B, timestamp=1496897606345,
value=value02
1 row(s) in 0.0170 seconds
```

By default, HBase limits the number of rows by counting from the first retrieved row. We could change this behaviour using STARTROW filter which accepts a row name. Issue help `'scan'` to find out how

REVERSE filter changes the order of the results, how to limit the results based on more than one column, and also how to only show the rows that started by a common prefix.

## 8. Fetch a Single Row

To retrieve a single row from a time, use the `get` command. Compare the following examples:

```
hbase(main):021:0> get 'tableName', 'row2'
COLUMN                CELL
0 row(s) in 0.0020 seconds

hbase(main):022:0> get 'tableName', 'row02'
COLUMN                CELL
 colFamily1:B         timestamp=1496897606345, value=value02
1 row(s) in 0.0170 seconds
```

## 9. Disable a Table

To remove a table from memory or before we can delete a table or change its settings, we need to disable the table using the `disable` command. We can re-enable it by `enable` command.

```
hbase(main):030:0> disable 'tableName'
0 row(s) in 2.2280 seconds

hbase(main):031:0> get 'tableName', 'row02'
COLUMN                CELL

ERROR: tableName is disabled.

hbase(main):032:0> enable 'tableName'
0 row(s) in 1.2300 seconds

hbase(main):033:0> get 'tableName', 'row02'
COLUMN                CELL
 colFamily1:B         timestamp=1496897606345, value=value02
1 row(s) in 0.0220 seconds
```

## 10. Drop a Table

To delete a table, we use the `drop` command. For example:

```
hbase(main):035:0> drop 'tableName'

ERROR: Table tableName is enabled. Disable it first.

Here is some help for this command:
```

```
Drop the named table. Table must first be disabled:
  hbase> drop 't1'
  hbase> drop 'ns1:t1'


hbase(main):036:0> disable 'tableName'
0 row(s) in 2.2460 seconds

hbase(main):037:0> list 'tableName'
TABLE
tableName
1 row(s) in 0.0060 seconds

=> ["tableName"]
hbase(main):038:0> drop 'tableName'
0 row(s) in 1.2470 seconds

hbase(main):039:0> list 'tableName'
TABLE
0 row(s) in 0.0040 seconds

=> []
hbase(main):040:0> enable 'tableName'

ERROR: Table tableName does not exist.

Here is some help for this command:
Start enable of named table:
  hbase> enable 't1'
  hbase> enable 'ns1:t1'
```

As the above examples show, we should disable a table before dropping it. Note that a table is still sitting on the filesystem after disabling, but completely removed after dropping (refer to the outputs of `list` command)

## 11. Exit The Shell

To exit the HBase Shell and disconnect from our cluster, simply issue quit command.

```
hbase(main):041:0> quit
user@ubuntu:~$ stop-hbase.sh
stopping hbase..................
localhost: stopping zookeeper.
user@ubuntu:~$ stop-yarn.sh
stopping yarn daemons
stopping resourcemanager
localhost: stopping nodemanager
no proxyserver to stop
user@ubuntu:~$ stop-dfs.sh
```

```
Stopping namenodes on [localhost]
localhost: stopping namenode
localhost: stopping datanode
Stopping secondary namenodes [0.0.0.0]
0.0.0.0: stopping secondarynamenode
user@ubuntu:~$ jps
24676 Jps
user@ubuntu:~$
```

Note that HBase is still running in the background, although we disconnected ourselves. In order to stop it completely, we should run the bin/stop-hbase.sh script (it can take a few minutes to disconnect). If we do not need to keep Hadoop and YARN running in the background, we can also stop them using stop-yarn.sh and stop-dfs.sh scripts, respectively. You can call jps command to make sure everything (including HMaster and HRegionServer processes) are shut down completely.

## Reference

- Apache HBase TM Reference Guide (https://hbase.apache.org/book.html#shell_exercises)

# 3.5.2
# Activity 2: Hadoop and HBase

There are many ways (https://www.cloudera.com/documentation/enterprise/5-4-x/topics/admin_hbase_import.html) to import data from other sources to HBase tables. In the previous activity, we learned how to use put command to insert one row at a time. One can extend that example and develop a simple script to read data from an external source (e.g., SQL table or CSV file) and insert them into an HBase table one by one. Although these approaches are possible, none of them leverages the parallel processing facilities of MapReduce. Fortunately, there are other ways to load a large volume of data into HBase tables while getting benefit from MapReduce advantages. In this activity, we will learn how to import the content of a CSV hosted on HDFS to an HBase table. Let's rock!

## From HDFS to HBase

### 1. Initiate The Daemons

As always, we need to run `start-dfs.sh`, `start-yarn.sh`, and `start-hbase.sh` before going further. You can check if these services are already running in the background using `jps` command.

### 2. Create A Table

Let's create a simple table called `tab` with only one column-family called `colFamily`:

```
user@ubuntu:~$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.4, r67592f3d062743907f8c5ae00dbbe1ae4f69e5af, Tue Oct 25 18:10:20
CDT 2016


hbase(main):001:0> create 'tab', 'colFamily'
0 row(s) in 1.4710 seconds

=> Hbase::Table - tab
hbase(main):002:0> list 'tab'
TABLE
tab
1 row(s) in 0.0190 seconds

=> ["tab"]
hbase(main):003:0> scan 'tab'
ROW                   COLUMN+CELL
0 row(s) in 0.1140 seconds
hbase(main):004:0> quit
user@ubuntu:~$
```

Great! We created a table using HBase Shell and quit the environment.

## 3. Upload The Data

Now it is the time to upload a CSV file to an HDFS directory. In the previous module, we learned how to do it using `hadoop fs -put` command. In this example, we upload a CSV file called `test.csv` into `/tmp/hbase/` directory. We can safely remove this file after importing the data into HBase table.

```
user@ubuntu:~$ hadoop fs -put test.csv /tmp/hbase/
user@ubuntu:~$ hadoop fs -ls /tmp/hbase/
Found 4 items
-rw-r--r--   1 user supergroup        108 2017-01-14 00:12 /tmp/hbase/hbase-
sensor.csv
-rw-r--r--   1 user supergroup         24 2017-01-14 03:41 /tmp/hbase/test.csv
user@ubuntu:~$ hadoop fs -cat /tmp/hbase/test.csv
a,b,c
1,2,3
4,5,6
7,8,9
user@ubuntu:~$
```

As shown in the above example, `test.csv` in this example only contains four rows (including header) and three columns.

## 4. Import to HBase

Here, we use `ImportTsv` (http://hbase.apache.org/0.94/book/ops_mgt.html#importtsv) command to import the content of the uploaded file into the created table. At the first sight, the command looks too lengthy and complicated. However, we will discuss each part of it in a minute!

```
user@ubuntu:~$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=HBASE_ROW_KEY,colFamily:a,colFamily:b '-
Dimporttsv.separator=,' tab /tmp/hbase/test.csv
```

This long command consists of six parts which we will discuss them in the following. Note that there should not be any extra space in any of these components. For example, you should not separate `-Dimporttsv.columns=HBASE_ROW_KEY,` and `colFamily:a,colFamily:b` from each other as they are inseparable parts one single component.

1. **`hbase`** Self-explanatory!
2. **`org.apache.hadoop.hbase.mapreduce.ImportTsv`** The main command that imports a delimited file (e.g., tab or comma separated value files) to an HBase table.
3. **`-Dimporttsv.columns=HBASE_ROW_KEY,colFamily:a,colFamily:b`** This part (starts with a hyphen symbol) declares the column names that we want to insert into. See the `put` command in the previous activity. In this example, we target two columns `a` and `b` under the column-family `colFamily`.
4. **`-Dimporttsv.separator=,'`** This part (also starts with a hyphen symbol) defines the delimiter that is used in the source file. In this example, we use a CSV file where the values are separated using a `,` symbol.
5. **`tab`** This is the name of our target HBase table. Recall that we just created this table in the second step.
6. **`/tmp/hbase/test.csv`** This is the source file that we stored on our HDFS.

## 5. Validate The Data

The insertion is already done! Here we just want to make sure everything is alright. To do so, we run HBase Shell once again and query our table. Since we do not have many data, we can simply issue a `scan` command without any limit.

```
user@ubuntu:~$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.4, r67592f3d062743907f8c5ae00dbbe1ae4f69e5af, Tue Oct 25 18:10:20
CDT 2016

hbase(main):001:0>scan 'tab'
ROW                   COLUMN+CELL
 1                    column=colFamily:a, timestamp=1496905476058, value=2
 1                    column=colFamily:b, timestamp=1496905476058, value=3
 4                    column=colFamily:a, timestamp=1496905476058, value=5
 4                    column=colFamily:b, timestamp=1496905476058, value=6
 7                    column=colFamily:a, timestamp=1496905476058, value=8
 7                    column=colFamily:b, timestamp=1496905476058, value=9
 a                    column=colFamily:a, timestamp=1496905476058, value=b
 a                    column=colFamily:b, timestamp=1496905476058, value=c
4 row(s) in 0.0400 seconds
```

As shown in the above-printed results, our tab table has four rows as expected. The first column of the `test.csv` table is treated as the row names (e.g., `a, 1, 4, 7`), and the second and third columns are imported as column `colFamily:a` and `colFamily:b`. Is there any observation that surprises you?

Now we can quit the HBase Shell, safely stop the daemons and remove the temporarily CSV file from the `/tmp/hbase/` directory.

## Reference


- Apache HBase TM Reference Guide (https://hbase.apache.org/book.html#shell_exercises)

# 4
# NoSQL, Spark, and Graph Processing

## Agenda

The main topics that we discuss in this module are NoSQL, Spark, and Graph Processing. We will learn the main attributes of NoSQL Databases and their key differences with other DataBase Management Systems (DBMS). Next, we will be learning Apache Spark and its main components along with its data abstraction scheme. Moving forward, we will be familiarizing ourselves with graph processing and an introduction to Spark GraphX with a little introduction to Neo4j. Finally, we will learn the main attributes of spatial databases and queries.

## Learning Objectives

At the end of this module, students should be able to:

- Explain and be able to differentiate among Data Frames, Datasets, and RDDs. Learn how to operate on RDDs, caching and developing a self-contained Spark application.
- Work with Spark Shell and perform basic ETL tasks and interactions with Hadoop-HDFS using Spark-Scala APIs.
- Gain an understanding of graphs, and be able to use GraphX by learning about property graphs, Pregel API, Vertex and Edge RDDs, Optimised Representations, and implement PageRank algorithm
- Introduction to Neo4j and Cypher as a complementary graph processing framework, implementation, and its basic use.

## References

Main references to this module are listed in the following:


- Apache Spark (http://spark.apache.org/)
- Resilient distributed datasets (RDDs): A fault-tolerant abstraction for in-memory cluster computing (https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf)
- Apache Spark Programming Guide (http://spark.apache.org/docs/latest/programming-guide.html)
- Key-Value Database (https://en.wikipedia.org/wiki/Key-value_database)
- Document-Oriented Database (https://en.wikipedia.org/wiki/Document-oriented_database)
- Bigtable (https://en.wikipedia.org/wiki/Bigtable)
- Graph Database (https://en.wikipedia.org/wiki/Graph_database)
- Neo4j (https://neo4j.com/), Graph Databases (https://neo4j.com/developer/graph-database/), Cypher (https://neo4j.com/developer/cypher-query-language/), and Cypher Documentation (https://neo4j.com/docs/developer-manual/current/cypher/)
- Bulk Synchronous Parallel (BSP) (https://en.wikipedia.org/wiki/Bulk_synchronous_parallel)
- GraphX Programming Guide (http://spark.apache.org/docs/latest/graphx-programming-guide.html)

# 4.1
# Introduction to NoSQL

## Introduction to NoSQL


*NoSQL is not "No SQL"! (Source: The Business Intelligence Blog)*

Given the variety of data generated by diverse sources, storage and retrieval of those produced data elements in stable, reliable and accessible repositories with high-levels of performance become a critical requirement. This includes both the historical data from previous events/interactions (that has already been captured and stored) and current streaming data (which is occurring right now). Therefore, we need certain storage facilities to save and query data elements as requested. This results in the emergence of data stores (https://en.wikipedia.org/wiki/Data_store). A data store is defined as a repository to store and manage huge collections of data elements, persistently. Prevalent types of data stores can be categorized in two groups *traditional* and *modern* repositories.

### Traditional Repositories

The traditional ways of storing our data are mainly categorized into two major categories:

- *Files and File Systems* - A file is a digital computer location that stores data. A file system (https://en.wikipedia.org/wiki/File_system), whether it is your Windows file system or Hadoop HDFS, manages the storage and retrieval of the data stored in a storage medium - mostly hard disk drives
- *Relational Databases* - According to Oracle (https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html), a database, in general, is a tool for storing the given information in such a way we can retrieve it whenever we want. In the simplest terms, a relational database, in particular, is one that presents information in tables with some rows and columns. Each row represents one (part of a) record and columns are the features or variables that describe the information. A table is referred to as a relation in the sense that it is a collection of objects of the same type (i.e., rows). The data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. Also, tables can (and usually do) relate to each other mainly using foreign keys. As mentioned in the previous chapters, a Database Management System (DBMS) such as HBase handles the way the data is stored, maintained, and retrieved. In the case of a relational database (https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html), a Relational Database Management System (RDBMS) such as MySQL (https://www.mysql.com/) or Teradata (http://www.teradata.com.au/?LangType=3081&LangSelect=true) performs these tasks. Database organization is based on the relational model proposed by Codd (https://dl.acm.org/citation.cfm?doid=362384.362685). Other famous examples of RDBMSs are Oracle Database (https://www.alexandriarepository.org/wp-content/uploads/20160819164034/index.html), Microsoft SQL Server (https://www.microsoft.com/en-au/sql-server/sql-server-2016), PostgreSQL (https://www.postgresql.org/). One main common concept that all these RDBMSs have in common is the Structured Query Language

(https://en.wikipedia.org/wiki/SQL) or SQL (http://sqlzoo.net/) that you already familiar with.

## Modern Repositories

Nowadays, modern databases and NoSQL (https://www.mongodb.com/nosql-explained) databases almost refer to the same category of DBMSs. As we discussed earlier, the term NoSQL which stands for "Not only SQL" refers to a complementary addition to RDBMSs and SQL. NoSQL databases are the next generation DBMS that are non-relational, open-source and horizontally scalable and are capable of handling large sets of distributed data. NoSQL databases provide more flexibility in storing different kinds of data than relational databases. Data elements in a NoSQL database are stored and retrieved using key-value pairs instead of storing them in rows/columns which give the ability to store unstructured data. In the previous chapter, we discussed HBase as one of the most popular NoSQL databases. Here, we provide a broader picture and briefly introduce various types of NoSQL databases:

- *Key-Value Store (Database)* - A key-value store (http://db-engines.com/en/article/Key-value+Stores) is a plain database which utilizes an associative array in the form of a data dictionary as the essential data model where each key is associated with one and only one value in a collection. This relationship is referred to as a key-value pair. The key in each pair is illustrated by a string such as a filename, URI or hash. The value, on the other hand, can be any type of data like a document, image, audio, video, or user preference file/location. Sometimes, key-value stores are referred to as the simplest NoSQL databases. The Oracle NoSQL (https://www.oracle.com/database/nosql/index.html), BerkeleyDB (https://www.oracle.com/database/berkeley-db/db.html), Redis (https://redis.io/), and Riak (http://basho.com/products/#riak) are some famous examples of key-value databases.
- Document Databases (https://en.wikipedia.org/wiki/Document-oriented_database) - store semi-structured data (similar to XML or JSON formats) and its description in document format and are one of the main categories of NoSQL databases. The document format refers to the complex data structures which are paired to their corresponding keys in a NoSQL database. Some examples of document databases are MongoDB (https://www.mongodb.com/nosql-explained), OrientDB (http://orientdb.com/orientdb/), and CouchDB (https://couchdb.apache.org/).
- *Wide-Column Stores (Columnar Database)*- large datasets tables are organized as columns (https://www.mongodb.com/nosql-explained) instead of rows and storage/retrieval processes of huge data volumes are performed faster than the conventional relational databases. Examples are Google Bigtable (https://cloud.google.com/bigtable/), Apache Cassandra (https://cassandra.apache.org/), Apache HBase (https://hbase.apache.org/).
- *Graph-Based Databases* - store information about networks of data where data elements are called nodes and connection among nodes are called edges and construct a graph (http://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL) of data elements and their interrelationships. Examples are Neo4J, Apache Giraph (https://giraph.apache.org/), GraphDB (http://graphdb.ontotext.com/), IBM Graph (https://www.ibm.com/us-en/marketplace/graph), Allegro Graph (http://franz.com/agraph/allegrograph/). In this chapter, we will explain this type in more details.

In the previous chapter, we discussed columnar database in details. In this chapter, we explain graph-based databases more. However, before we go any further, we need to be familiar with Spark technology that most of the NoSQL databases built on or benefiting (https://dzone.com/articles/why-spark-and-nosql) from it in one or another way. In addition, Spark helps us in many other tasks such as processing stream data, visualization of large-scale data and fast performing analytics (e.g., building machine learning models). Therefore, learning Spark before discussing other concepts seems inevitable.

# 4.2
# Introduction to Spark

## Introduction

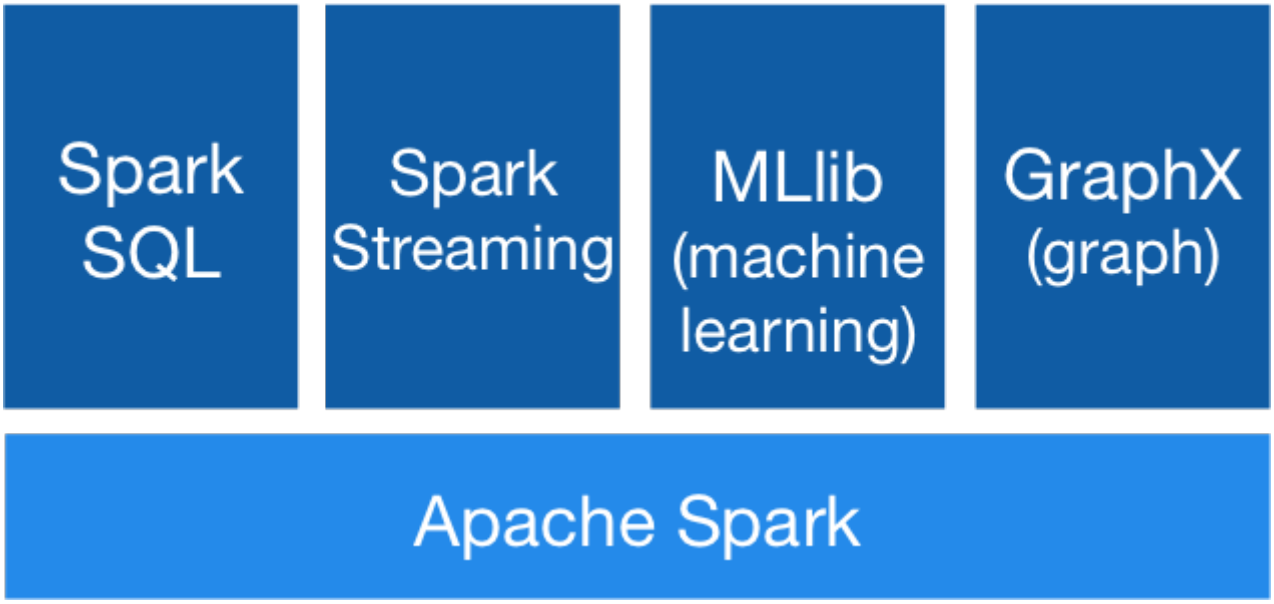
Given the diverse types of data and the fact that we live in the big data era, utilization of efficient and robust data processing models have become a critical task. Apache Spark is one efficient mean to address this requirement and to effectively provide tools for big data manipulation. In this chapter, we use Spark for graph processing. In the next chapters, we discuss its application in stream data processing.

## Apache Spark

As noted in section 2.2, Apache Spark (https://spark.apache.org/) is a fast open source cluster computing framework capable of large-scale data processing. It enables efficient, in-memory processing of data with an expressive development API. The main advantages of Spark are:

- **Speed** - Spark was designed to address MapReduce performance issues, as per Spark website, programs using Spark can run up to 100x faster (https://spark.apache.org/) than Hadoop MapReduce with memory intensive operations.
- **Ease of use** - Spark provides different sets of API (https://spark.apache.org/docs/latest/) for several programming languages such as Scala, Java, Python, and R.
- **Generality** - Spark supports a rich set of high-level tools to support SQL, machine learning, analytics, graphs and streaming data processing.

### Spark Architecture



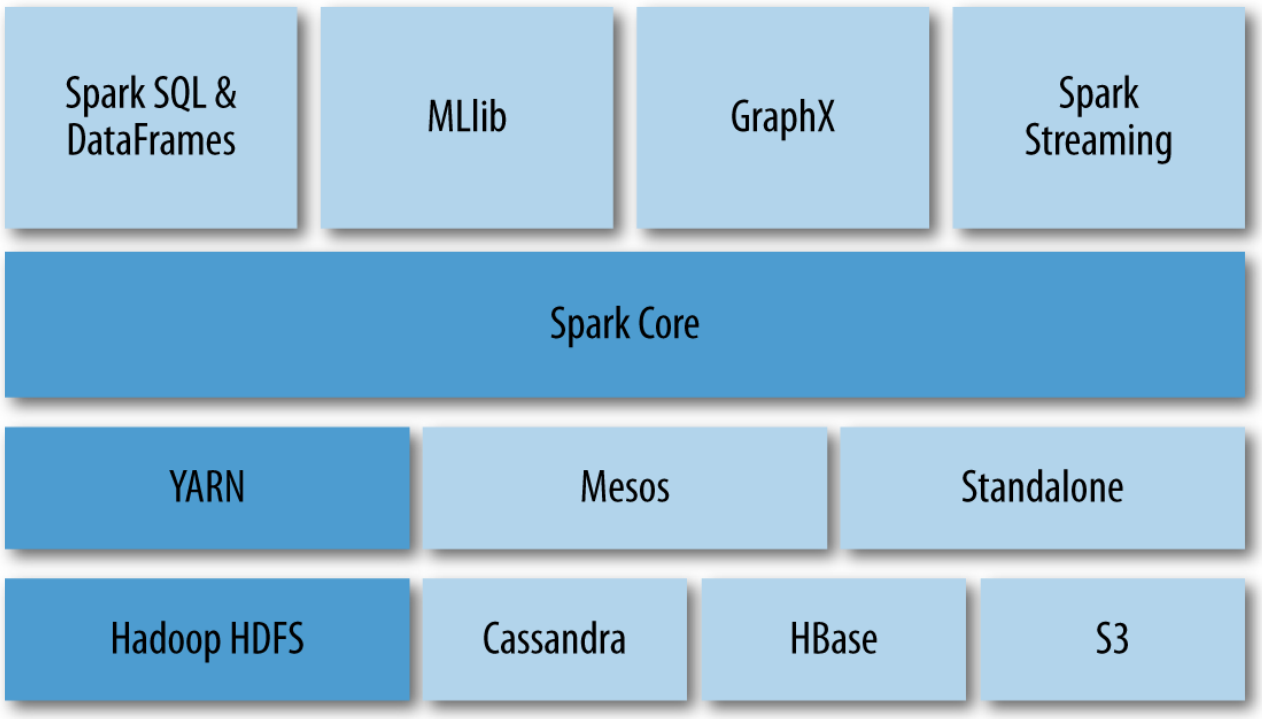

*Spark Architecture (Source: Spark's official website)*

As the above diagram shows, main components of Apache Spark are the core, Spark SQL, Spark

Streaming, MLlib, and GraphX.

- **Apache Spark Core** - is the underlying execution engine and is the foundation of Apache Spark which provides in-memory computing. It is also responsible for distributed task dispatching, scheduling, and some I/O functionalities through its available high-level API named RDD (Resilient Distributed Dataset) API which will be discussed later. RDD API comprises two kinds of operations: *transformations* and *actions*.
- **GraphX** - is Spark's distributed graph processing framework by introducing a [new](https://spark.apache.org/docs/latest/graphx-programming-guide.html) Graph abstraction. GraphX encompasses a rich set of graph algorithms which makes graph processing and analytics an easy task. It provides an expressive collection of operators (such as *subgraph, joinVertices,* and *aggregateMessages*) and an optimized version of the [Pregel API](https://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel) (Google's scalable, flexible and fault-tolerant platform to express arbitrary graph algorithms).
- **Spark Streaming** - enables scalable stream analytics based on Spark Core's fast scheduling capability over the streaming data. It collects streaming data from diverse sources such as Kafka, Flume, Kinesis, or TCP sockets. Prior to processing the received data, [Spark Streaming](https://spark.apache.org/docs/latest/streaming-programming-guide.html) divides them into mini-batches. The acquired data then could be processed using some functions such as *map*, *reduce*, *join*, and *window* by the Spark engine. Finally, the results of the processed data could be disseminated to filesystems, databases, and so forth.
- **Machine Learning Library (MLlib)** - is Spark's distributed [machine learning library](https://spark.apache.org/docs/latest/ml-guide.html) which provides scalable machine learning (ML) tools such as different ML algorithms e.g., classification, clustering, regression, feature extraction, and reduction techniques.
- **Spark SQL** - It supports the structured and semi-structured data [processing](https://spark.apache.org/docs/latest/sql-programming-guide.html).

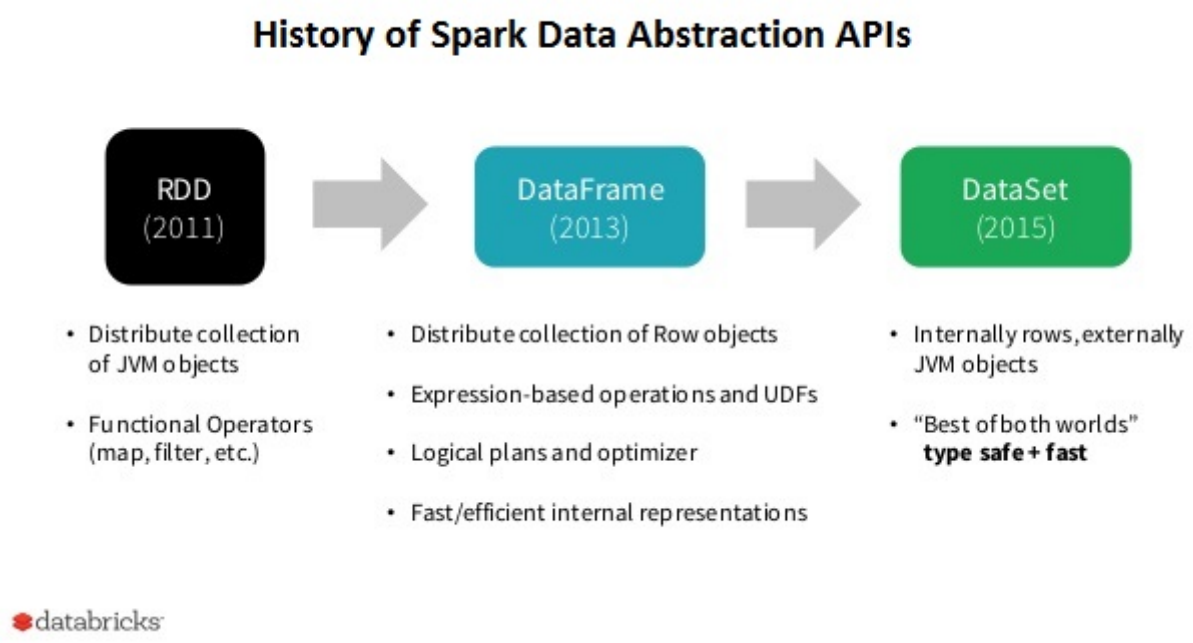
In this module, we learn Spark core and GraphX. The Spark streaming and MLlib are covered in the next modules.




*Spark and Hadoop (Source: Data Analytics with Hadoop by Kim and Bengfort)*

## Spark Data Abstraction APIs

Spark framework is built on the concept of distributed datasets with multiple programming language tools. To be able to utilize Spark's power in manipulating a huge volume of data, it provides some expressive and high-level data abstraction APIs. The following demonstrates the history of the Spark data abstraction APIs.




*Short history of Spark core APIs (credit: Databricks)*

In the following, we elaborate on Spark's three major data abstraction APIs. Please note that given the same data elements, all these three abstractions will provide the user with the same results.

### 1. Resilient Distributed Dataset (RDD)

RDD is the oldest Spark data abstraction from Spark 1.0. It is the building block of spark and its fundamental data structure. No matter which abstraction DataFrame or DataSet we use, the final computation is done on RDD. RDD lazily evaluates an immutable parallel collection of objects exposed with lambda functions. It is quite simple and straightforward because it provides familiar object-oriented programming (https://en.wikipedia.org/wiki/Object-oriented_programming) (OOP) based APIs with compile time safety. We can load any data from a source, convert them into an RDD and store it in memory to compute the results. An RDD can be easily cached if the same set of data needs to be recomputed.

Main features of RDDs are listed in the following:

- Support of unstructured data such as media streams or streams of text (without thinking of any particular schema like the columnar format).
- Support of functional programming constructs to help developers manipulate data.
- Support of data immutability. Given that RDD is comprised of partitions (fundamental elements of parallelism in RDD), each of which represents a division of immutable data, it supports the data immutability which preserves computation consistency.
- Support of lazy transformations. As all transformations in Spark are lazy (they are only computed if

an action asks for a result to be computed and returned), the transformation results are not calculated right away.

RDD deals with a critical challenge: performance! Because RDD should manage the overhead of garbage collection (https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) (GC) and serialization issues of the in-memory Java Virtual Machine (http://www.javatpoint.com/internal-details-of-jvm) (JVM) objects. This hinders the performance when the data volume increases. Another limitation of RDD is that it just handles structured data. Unlike DataFrame and Datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

## 2. Spark DataFrame

DataFrame is the next generation of data abstraction released with Spark 1.3. Spark proposed DataFrame to address major RDD issues. A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database. It provides a domain-specific language for structured data manipulation in different programming languages. It is similar to RDD in the way that it is also an immutable distributed collection of data. However, unlike RDD, the data is organized into named columns which provide a schema view of the data.

DataFrames are also lazy triggered like RDDs; however, the performance measurements have been improved significantly given its powerful features such as *Custom Memory Management* (known as Project Tungsten (https://spark-summit.org/2015/events/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal/)) and *Optimized Execution Plans* (known as Catalyst Optimizer (https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)).

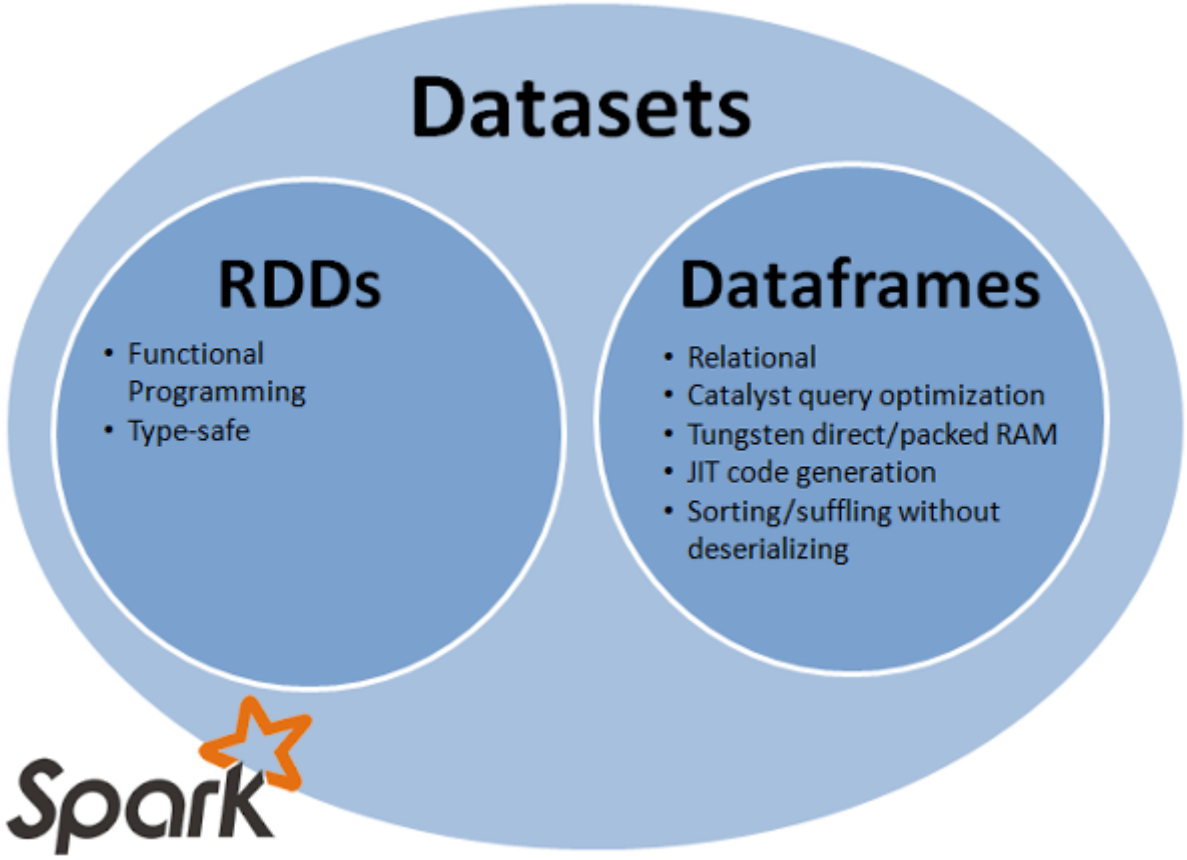
So, key features of DataFrames can be categorized as the following:

- Distributed collection of row objects similar to the table concept in RDBMS with more optimized capabilities.
- Supporting diverse data format processing. DataFrame supports structured and unstructured data formats as its input and output.
- Supporting of useful optimizations and compatibility with Hive.

Although DataFrames addressed some key challenges of RDDs, they have issues themselves. DataFrames suffer from the lack of type safety. Programming errors are fetched at run-time.

## 3. Spark DataSet

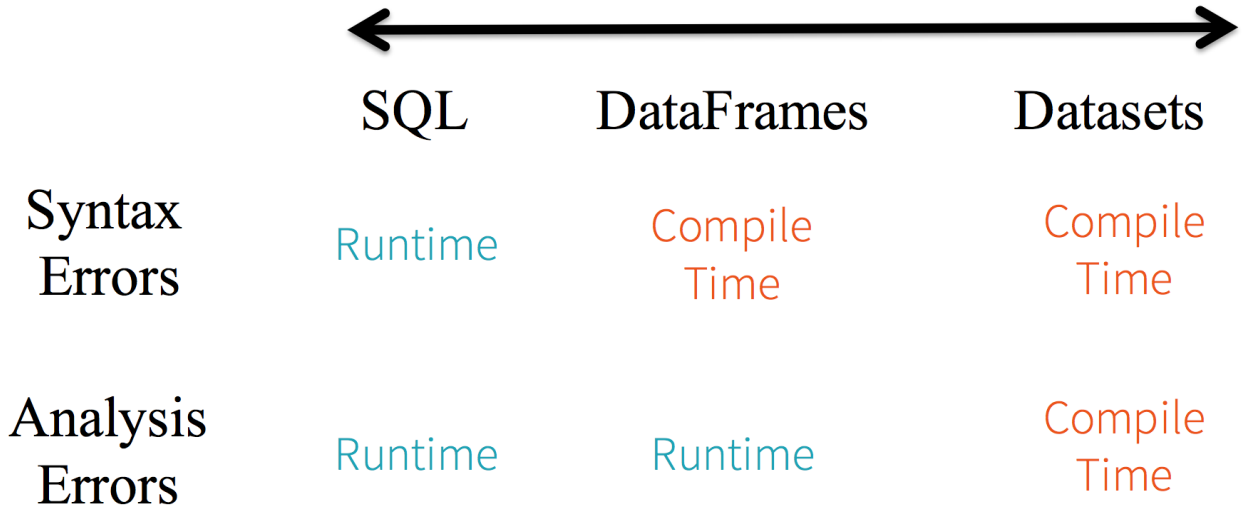
DataSet is the latest abstraction introduced with Spark1.6. It is proposed to address the shortcomings of RDDs and DataFrames and to act as the best of the two. The following Venn diagram shows the interrelationships between RDD, DataFrame, and DataSet.




*Spark DataFrame vs RDD (Source: Chandan Prakash's Blog)*

DataSet is a collection of distributed data in the forms of ***strongly-typed*** JVM objects. It provides compile-time safety OOP based APIs as well as performance improvements on DataFrame Catalyst Optimizer and Custom Memory Management. Some major benefits of Spark DataSets are static typing and runtime type safety, high-level abstraction and schema-based view over structured and semi-structured data, expressive and developer friendly APIs, and improved performance and optimization factors.

The following figure compares the type-safety spectrum regarding syntax and analysis errors in SQL, Spark DataFrames, and Spark DataSets.




*SQL, Spark DataFrames, and DataSets type-safety spectrum (Source: Databricks)*

To sum up, RDDs are the low-level and DataFrames and DataSets are the high-level computational APIs provided by Apache Spark. It means that RDDs support low-level functionality and control, while DataFrames and DataSets allow custom view and structure which support high-level and domain specific operations. Given the Spark's shift in computation from unstructured (RDDs) towards structured (DataFrames and DataSets), developers are provided with higher-level APIs and more performance optimization techniques. DataFrame's support of the structured computation lacked the type safety. Therefore, DataSet was proposed to address the issue that acted as a unification of the two (RDD and DataFrame) to bring the best abstraction possible.

For more details about Spark APIs, please refer to this (https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html) and this (https://ideata-analytics.com/apache-spark-api-comparision/). Also, for a wider comparison of the three (RDD, DataFrame, DataSet) in terms of their features, please refer to this resource (http://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/).

# 4.3
# Spark Data Abstraction

So far, we introduced the three data abstraction types that Spark provides. Now, we can explore each of them further.

## 1. Resilient Distributed Dataset (RDD)

RDD is an immutable, distributed, and fault-tolerant collection of elements for in-memory parallel cluster computing. The term was coined by the University of Berkeley (https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf): "*RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators*". Put simply, RDD is an abstract representation of the data which is divided into the partitions and distributed across different nodes in the cluster.

### 1.1 RDD Creation

There are two ways to create (https://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds) RDDs:

**Parallelized Existing Collections** - by parallelizing an existing collection in the program. By calling Spark's provided methods, one can create an RDD instance using existing collection of objects. The elements of the collection are copied to create a distributed dataset. Next, the created distributed dataset can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5:

```
// create Spark Context object
val sc = SparkContext()
// create a data collection
val data = Array(1, 2, 3, 4, 5) // data is a collection

// parallelize collection to create RDD
val distData = sc.parallelize(data) // distData is an RDD object
```

**External Datasets** - by referencing a dataset persisted in an external storage using `SparkContext`'s `textFile` method. The dataset can be stored in a shared filesystem, HDFS, HBase, Cassandra or any external storage supported by Hadoop (more specifically, offering a Hadoop *InputFormat*). Here is an example invocation:

```
// assuming the data is physically stored in data.txt
val distFile: RDD[String] = sc.textFile("data.txt") // distFile is an RDD object
```

### 1.2 RDD Operations

RDD provides two types of operations

(https://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds); ***transformations***, and ***actions***:

**A. Transformations** create a new dataset from an existing one. They are lazy which means they are not applied until an action is called and requests for results to be returned to the driver program. Instead, they just "remember" the operation to be performed and the dataset (e.g. file) to which the operation is to be performed on. This assist Spark to operate more efficiently. A list of common transformation operations are as the following:

- **`map(`***`func`***`)`** - Returns a new distributed dataset formed by passing each element of the source through a function `func`.
- **`filter(`***`func`***`)`** - Returns a new dataset formed by selecting those elements of the source on which `func` returns true.
- **`union(`***`otherDataset`***`)`** - Returns a new dataset that contains the union of the elements in the source dataset and the argument.
- **`flatMap(`***`func`***`)`** - Similar to map, but each input item can be mapped to 0 or more output items (so `func` should return a sequence rather than a single item).
- **`distinct(`***`[numTasks])`***`)`** - Returns a new dataset that contains the unique elements of the source dataset.
- **`groupByKey(`***`[numTasks]`***`)`** - When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Iterable<V>)` pairs.
  - **Note 1:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance.
  - **Note 2:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
- **`reduceByKey(`***`func, [numTasks]`***`)`** - When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function func, which must be of type `(V,V) => V`. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
- **`sortByKey(`***`[ascending], [numTasks]`***`)`** - When called on a dataset of `(K, V)` pairs where K implements *Ordered*, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument (optional).
- **`aggregateByKey(`***`zeroValue)(seqOp, combOp, [numTasks]`***`)`** - When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, U)` pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
- **`join(`***`otherDataset, [numTasks]`***`)`** - When called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.

**B. Actions** return a value or export data to the driver program after performing a computation on the RDD. Actions force a transformation to be applied to the dataset. A set of most frequently used action operations are listed in the following:

- **`reduce(`***`func`***`)`** - Aggregates the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- **`collect()`** - Returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- **`count()`** - Returns the number of elements in the dataset.
- **`take(n)`** - Returns an array with the first n elements of the dataset.

- **`first()`** - Returns the first element of the dataset (similar to `take(1)`).
- **`saveAsTextFile(`*`path`*`)`** - Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
- **`saveAsSequenceFile(`*`path`*`)`** - Writes the elements of the dataset as a Hadoop *SequenceFile* in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement *Hadoop's Writable Interface*. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
- **`countByKey()`** - Only available on RDDs of type `(K, V)`. Returns a hashmap of `(K, Int)` pairs with the count of each key.

The following summarizes RDD operations in one table (https://dl.acm.org/citation.cfm?id=2228301):

| | | | |
|---|---|---|---|
| **Transformations** | *map*(*f* : T ⇒ U) | : | RDD[T] ⇒ RDD[U] |
| | *filter*(*f* : T ⇒ Bool) | : | RDD[T] ⇒ RDD[T] |
| | *flatMap*(*f* : T ⇒ Seq[U]) | : | RDD[T] ⇒ RDD[U] |
| | *sample*(*fraction* : Float) | : | RDD[T] ⇒ RDD[T] (Deterministic sampling) |
| | *groupByKey*() | : | RDD[(K, V)] ⇒ RDD[(K, Seq[V])] |
| | *reduceByKey*(*f* : (V, V) ⇒ V) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *union*() | : | (RDD[T], RDD[T]) ⇒ RDD[T] |
| | *join*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))] |
| | *cogroup*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))] |
| | *crossProduct*() | : | (RDD[T], RDD[U]) ⇒ RDD[(T, U)] |
| | *mapValues*(*f* : V ⇒ W) | : | RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning) |
| | *sort*(*c* : Comparator[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *partitionBy*(*p* : Partitioner[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| **Actions** | *count*() | : | RDD[T] ⇒ Long |
| | *collect*() | : | RDD[T] ⇒ Seq[T] |
| | *reduce*(*f* : (T, T) ⇒ T) | : | RDD[T] ⇒ T |
| | *lookup*(*k* : K) | : | RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs) |
| | *save*(*path* : String) | : | Outputs RDD to a storage system, *e.g.*, HDFS |

*Spark RDD Summary*

## 1.3 RDD Caching (and Persistence)

Caching or persisting (https://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence) of data in memory through various operations is one of the significant advantages of Spark framework. Caching as an optimization technique refers to storing the interim results in-memory or another robust and reliable storage in a way that they can be used in likely future operations. RDDs can be cached using cache operation. They can also be persisted using `persist()` operation.

According to Spark website, utilization of caching helps actions run faster (sometimes more than 10x). Caching is a critical approach in iterative and interactive processes. You can mention any particular RDD to be persisted by calling `persist()` or `cache()` methods on them. After tagging an RDD to be persisted, it will be kept in memory on the nodes the very first time it is computed in an action. Note that Spark Caching process is fault tolerant which means it will automatically re-compute any RDD instance which is lost using the transformations that created it in the beginning.

# 2. Spark DataFrames

A DataFrame, which resembles R and Python (Pandas) dataframes, is a distributed collection of data organized into named columns. It is conceptually similar to a table in a relational database.

### Spark DataFrame Creation

We can create a Spark DataFrame either from an existing RDD or external data sources:

**Using RDD** - by calling `toDF()` method. This method is a part of *sqlContext* `implicits` package:

```
// create a SQL Context
val sqlContext = new SQLContext(sc)

// import implicits package
import sqlContext.implicits._

// convert the existing RDD to a DataFrame
rdd.toDF()

df.show()
```

**External Datasets -** by directly calling sqlContext read methods. For an example, the following creates a DataFrame based on the content of a JSON file:

```
// create a SQL Context
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// assuming the data is stored in data.json
val df = sqlContext.read.json("data.json")
```

# 3. Spark DataSets

Spark DataSet API is an extension to DataFrames that provides a type-safe, object-oriented programming interface. It is a strongly-typed, immutable collection of objects that are mapped to a relational schema.

### DataSet Creation

There are two ways to create a DataSet; from JVM object collection and external sources:

**From JVM Object Collection -** by importing `sqlContext.implicits` package (similar to DataFrame creation) and then using `.toDS()` method to create a DataSet:

```
// create a SQL Context
val sqlContext = new SQLContext(sc)

// import implicits package
import sqlContext.implicits._

// convert a sequence to a DataSet
val ds = Seq(1, 2, 3).toDS()
```

**External Source -** by directly calling sqlContext read methods (similar to creating a DataFrame). For an example, the following creates a DataFrame based on the content of a JSON file:

```
// create a SQL Context
val sqlContext = new SQLContext(sc)

// assuming the data stored in data.json file
val dataSet = sqlContext.read.json("data.json").as[Root] // dataSet is a DataSet
object
```

# 4.4
# Activity: Spark Shell

## Introduction

Spark (https://spark.apache.org/docs/latest/quick-start.html) Shell is an interactive shell that supports data exploration/manipulation in Apache Spark. The shell is written in Scala programming language and gives an interactive and auto-complete compatible command-line environment where developers can write their own codes, SQL scripts, and so forth to get themselves familiar with Spark features. Spark Shell provides an easy and convenient way to prototype certain operations quickly, without having to develop a full program, packaging it and then deploying it. To start with Spark Shell, one can simply download Apache Spark from the website or use one of the available sandboxes or virtual machines that include Spark. The following is a list of examples in Scala programming language to run Apache Spark Shell and do some basic tasks.

## Getting Started

To be able to perform the same operation we discuss here, please make sure the necessary data files are accessible by your Spark instance. If these files are not already available, please upload this zip file and unzip it in a folder such as spark-sample in your home directory.

### Running Spark Shell

To initiate running Spark Shell, we should go to the directory where Spark is installed and execute **`$SPARK_HOME$`**`/spark-shell`. For your convenience, we added the `spark-shell` script to the global path, therefore, you can call it anywhere. Note that you can run the above statement in any shell. For the sake of consistency, all our examples are based on Scala Shell. Therefore, you should try them in a Scala environment to see similar outputs.

The following shows the Spark Shell welcome message. Based on your instance of Spark, the output you see may be slightly different.

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

## 1. Create an RDD

As mentioned earlier, RDDs are Spark's foundational abstraction APIs and are distributed collections of items. According to the notes in RDD **Creation** section, RDDs can be created in two ways: ***parallelized existing collections*** or ***external datasets*** using Hadoop's `InputFormat()` like HDS files. The following example creates a new RDD from an existing collection (a text file):

```
scala> val smallText = sc.textFile("smallTextFile.txt") //creates an RDD
smallText: org.apache.spark.rdd.RDD[String] = smallTextFile.txt
MapPartitionsRDD[1] at textFile at <console>:24
```

Next, we provide some simple examples of calling Spark operations (**actions** and **transformations**). As mentioned before, actions do computations and generate results (values) and transformations generate new RDDs' locations.

## 2. Simple Actions

The following is an example of calling two distinct actions. The first one counts the number of items in the RDD (in this example, each item is a line from a text file). The second example retrieves the first item of the RDD.

```
scala> smallText.count() // Number of items in this RDD
res0: Long = 54
```

Notice that when `textFile()` creates an RDD from a text file, it treats each line of the files as a separate item. Therefore, the number of items that `count()` returns, in this case, is actually the number of lines of the source text file. Let's see what is the first item stored in this RDD (can you guess?):

```
scala> smallText.first() // First item in this RDD
res0: String = Apache License
```

Right! The first item is the first line of the fed text file. Do you know how we can take more than one items from an RDD?

```
scala> smallText.take(10)
res24: Array[String] = Array(Apache License, "", Version 2.0, January 2004, "",
http://www.apache.org/licenses/, "", TERMS AND CONDITIONS FOR USE, REPRODUCTION,
AND DISTRIBUTION, "", 1. Definitions., "")
```

Did you notice that some of the items have zero length (i.e., ""? These items represent empty lines (technically the lines with only one newline character). What if we want to randomly sample from our RDD?

```
scala> smallText.takeSample(false, 5) // returns 5 random samples without
replacement
res25: Array[String] = Array(2. Grant of Copyright License. Subject to the terms
and conditions of this License, each Contributor hereby grants to You a
perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of, publicly display,
publicly perform, sublicense, and distribute the Work and such Derivative Works
in Source or Object form., "", 7. Disclaimer of Warranty. Unless required by
applicable law or agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES
OR CONDITIONS OF ANY KIND, either express or implied, including, without
limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT,
MERCHANTABILITY, or FITNESS FOR A P...
```

The false in the above examples indicate that we want random sampling with no replacement (no repeated item). Note that `takeSample()` and `sample()` look similar but perform differently. Both methods randomly sample from an RDD. However, `takeSample()` returns an array of the sampled results whilst `sample()` creates another RDD from the sample population. Therefore, the later one is a transformation rather than an action.

## 3. Simple Transformation

This example represents the usage of transformation operations. Let's issue `sample()` method and compare its results with `takeSample()` from the previous example:

```
scala> val sampleText = smallText.sample(false, 0.5) //randomly select 50% items
sampleText: org.apache.spark.rdd.RDD[String] = PartitionwiseSampledRDD[6] at
sample at <console>:26

scala> sampleText.count() //how many items we selected?
res26: Long = 24
```

Now, let's go further and filter some of the items based on their content. Since our items are character strings (i.e., each one is a line of text), we can filter all items that contain a particular word using `filter` operation. Note that `filter()` is also a transformation operation that creates another RDD containing a subset of the items:

```
scala> val linesWithLicense = smallText.filter(line => line.contains("license"))
linesWithLicense: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[8] at
filter at <console>:26

scala> linesWithLicense.count()
```

```
res40: Long = 5

scala> linesWithLicense.take(5)
res41: Array[String] = Array(http://www.apache.org/licenses/, 2. Grant of
Copyright License. Subject to the terms and conditions of this License, each
Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-
charge, royalty-free, irrevocable copyright license to reproduce, prepare
Derivative Works of, publicly display, publicly perform, sublicense, and
distribute the Work and such Derivative Works in Source or Object form., 3.
Grant of Patent License. Subject to the terms and conditions of this License,
each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-
charge, royalty-free, irrevocable (except as stated in this section) patent
license to make, have made, use, offer to sell, sell, import, and otherwise
transfer the Work, where such license ...
scala>
```

The above example filters the items from `smallText` that contain word "license" and stores them in `linesWithLicense` RDD.

To make it more interesting, one can combine the two Spark operations in one example. For instance, we can combine the two first lines of the above example to one single statement:

```
scala> smallText.filter(line => line.contains("license")).count()
res0: Long = 5
```

As another example, we can write simple codes to calculate the length of each line of the file, in terms of the number of words or letters. To find the number of letters in each line, we can perform a one to one mapping of each line to its length. In Scala, length returns the number of characters in a string:

```
scala> smallText.map(line => line.length).collect()
res54: Array[Int] = Array(14, 0, 25, 0, 31, 0, 60, 0, 15, 0, 138, 0, 115, 0,
455, 0, 106, 0, 167, 0, 221, 0, 236, 0, 458, 0, 883, 0, 178, 0, 382, 0, 946, 0,
223, 0, 94, 100, 257, 944, 0, 363, 439, 0, 275, 0, 583, 0, 705, 0, 646, 0, 27,
0)
```

Note that Spark does not perform the calculation unless we call collect() method. To find the number of letters in the longest line of the file, we could issue:

```
scala> smallText.map(line => line.length).max()
res53: Int = 946
```

What about the number of words in each line? Let's assume that words are separated by single whitespace `" "`. Now, we can simply split lines and then map the array of the split items (words) to the number of items that each item has. Since this is a one to one mapping between a character string (line) to a single `Int` (number of words in the line), we can use the conventional `map` function:

```
scala> smallText.map(line => line.split(" ").length).collect()
res51: Array[Int] = Array(2, 1, 4, 1, 1, 1, 8, 1, 2, 1, 22, 1, 18, 1, 75, 1, 16,
1, 23, 1, 32, 1, 41, 1, 74, 1, 137, 1, 27, 1, 53, 1, 142, 1, 35, 1, 18, 17, 43,
153, 1, 55, 64, 1, 45, 1, 84, 1, 114, 1, 99, 1, 5, 1)
```

Note that the empty lines have the length of 1 as they still contain the newline symbol. Can you change

one keyword in the above example to find the number of words in the longest line?

We will see more examples of actions and transformations at the end of this activity.

## 4. Create a DataFrame

As we mentioned before, with a `SparkSession`, applications can create DataFrames from an existing RDD, from a Hive table, or from Spark data sources. Let's try converting the `smallText` RDD in the previous examples to a DataFrame:

```
scala> val df = smallText.toDF("lines") //RDD to DF
df: org.apache.spark.sql.DataFrame = [lines: string]

scala> df.show()
+--------------------+
|               lines|
+--------------------+
|      Apache License|
|                    |
|Version 2.0, Janu...|
|                    |
|http://www.apache...|
|                    |
|TERMS AND CONDITI...|
|                    |
|     1. Definitions.|
|                    |
|"License" shall m...|
|                    |
|"Licensor" shall ...|
|                    |
|"Legal Entity" sh...|
|                    |
|"You" (or "Your")...|
|                    |
|"Source" form sha...|
|                    |
+--------------------+
only showing top 20 rows
```

In the above statements, `"lines"` is the name of the column that we added to the created DataFrame. As another example, the following creates a DataFrame based on the content of a JSON (https://en.wikipedia.org/wiki/JSON) file. The content of our source file (`people.json`)is as follows:

```
{"name":"Tom"}
{"name":"Andy", "age":22}
{"name":"Emma", "age":19}
```

Let's read this file and create a Spark DataFrame to store its content:

```
scala> val df = spark.read.json("people.json")
```

```
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> df.show()
+----+----+
| age|name|
+----+----+
|null| Tom|
|  22|Andy|
|  19|Emma|
+----+----+
```

## 5. Create a DataSet

We can easily convert an RDD to a DataSet almost in the same way we convert to a DataFrame:

```
scala> val ds = smallText.toDS() // RDD to DS
ds: org.apache.spark.sql.Dataset[String] = [value: string]

scala> ds.show()
+--------------------+
|               value|
+--------------------+
|      Apache License|
|                    |
|Version 2.0, Janu...|
|                    |
|http://www.apache...|
|                    |
|TERMS AND CONDITI...|
|                    |
|     1. Definitions.|
|                    |
|"License" shall m...|
|                    |
|"Licensor" shall ...|
|                    |
|"Legal Entity" sh...|
|                    |
|"You" (or "Your")...|
|                    |
|"Source" form sha...|
|                    |
+--------------------+
only showing top 20 rows
```

We can also create a DataSet using an arbitrary case class which also helps us to work around Spark limit on the number of fields a DataSet can have (currently 22 fields). For example:

```
scala> case class Person(name: String, age: Long)
defined class Person

scala> val caseClassDS = Seq(Person("Andy", 32)).toDS()
```

```
caseClassDS: org.apache.spark.sql.Dataset[Person] = [name: string, age: bigint]

scala> caseClassDS.show()
+----+---+
|name|age|
+----+---+
|Andy| 32|
+----+---+
```

We can use this case class to import our JSON file (that we used to create a DataFrame) into a DataSet:

```
scala> val peopleDS = spark.read.json("people.json").as[Person]
peopleDS: org.apache.spark.sql.Dataset[Person] = [age: bigint, name: string]

scala> peopleDS.show()
+----+----+
| age|name|
+----+----+
|null| Tom|
|  22|Andy|
|  19|Emma|
+----+----+
```

# Word Count Example

In the Hadoop MapReduce activities, we learned how to count the number of times each word repeated in a text file. Here, we want to solve this problem using Spark operations. In comparison with the Hadoop version, this example is very short and easy to read:

```
scala> val counts = smallText.flatMap(line => line.split(" ")).map(word =>
(word,1)).reduceByKey(_+_)
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[7] at reduceByKey
at <console>:26
```

In the above example, we split each line to its words. Therefore, we need to use `flatMap` instead of `map` because each single item (line) is mapped to a series of new items (multiple words). Then, each word is mapped to a key-value tuple `(word,1)` where `word` serves as the key and 1 as its value. These keys are then used in the reduce part. Since we want to sum the occurrence of a word, we issue `reduceByKey` to aggregate all tuples with the common word. The `_+_` symbol is nothing but a summation function similar to: `((Int: a, Int: b) => Int = a + b)`.

The only thing remains is to collect the results:

```
scala> counts.collect()
res1: Array[(String, Int)] = Array((under,6), (Unless,2), (Contributions),1),
(offer,1), (NON-INFRINGEMENT,,1), (agree,1), (its,3), (event,1),
(intentionally,2), (Grant,2), (have,2), (include,2), (responsibility,,1),
(writing,1), (MERCHANTABILITY,,1), (Contribution,3), (express,1), ("Your"),1),
((i),1), (However,,1), (been,2), (files;,1), (This,1), (stating,1),
(conditions.,1), (non-exclusive,,2), (appropriateness,1), (marked,1), (risks,1),
(any,28), (IS",1), (filed.,1), (Sections,1), (fee,1), (losses),,1), (out,1),
(contract,1), (from,,1), (4.,1), (names,,1), (documentation,,2), (contract,,1),
(unless,1), (below).,1), (wherever,1), (verbal,,1), (ANY,1), (version,1),
```

```
(file.,1), (are,6), (no-charge,,2), (2.,1), (assume,1), (reproduction,,3),
(file,3), (offer,,1), (licenses,1), (grant,1),...
scala>
```

Nice and simple!

Now, can you modify the above example to calculate how many words we have in this document? What about only counting the words with the length between 3 to 7 letters? Or perhaps only counting words that start with an a or b but not end to c and d! As you can see, there are many possible ways to change or expand this simple word count example to practice Spark action and transformation operations.

# Caching

Caching of frequently accessed data (into its cluster-wide memory) is one of the advantages of Spark. Such facility is extremely helpful when we need to frequently read an RDD or tune an iterative algorithm such as PageRank. Let's give it a try.

First of all, we need a function to help us record the time it takes to perform a task. The following example (http://biercoff.com/easily-measuring-code-execution-time-in-scala/) does the job:

```
//borrowed from:
//biercoff.com/easily-measuring-code-execution-time-in-scala/
def time[R](block: => R): R = {
    val t0 = System.nanoTime()
    val result = block    // call-by-name
    val t1 = System.nanoTime()
    println("Elapsed time: " + (t1 - t0) + "ns")
    result
}
```

Now, we do a simple experiment. First, the execution time of word count example without any caching. Then, we cache the RDD, and time the word count code once again.

```
scala> def time[R](block: => R): R = {
     |     val t0 = System.nanoTime()
     |     val result = block    // call-by-name
     |     val t1 = System.nanoTime()
     |     println("Elapsed time: " + (t1 - t0) + "ns")
     |     result
     | }
time: [R](block: => R)R

scala> val counts = smallText.flatMap(line => line.split(" ")).map(word =>
(word,1)).reduceByKey(_+_)
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[16] at reduceByKey
at <console>:26


scala> time{counts.collect()}
Elapsed time: 31014981ns
res14: Array[(String, Int)] = Array((under,6), (Unless,2), (Contributions),1),
(offer,1), (NON-INFRINGEMENT,,1), (agree,1), (its,3), (event,1),
```

```
(intentionally,2), (Grant,2), (have,2), (include,2), (responsibility,,1),
(writing,1), (MERCHANTABILITY,,1), (Contribution,3), (express,1), ("Your"),1),
((i),1), (However,,1), (been,2), (files;,1), (This,1), (stating,1),
(conditions.,1), (non-exclusive,,2), (appropriateness,1), (marked,1), (risks,1),
(any,28), (IS",1), (filed.,1), (Sections,1), (fee,1), (losses),,1), (out,1),
(contract,1), (from,,1), (4.,1), (names,,1), (documentation,,2), (contract,,1),
(unless,1), (below).,1), (wherever,1), (verbal,,1), (ANY,1), (version,1),
(file.,1), (are,6), (no-charge,,2), (2.,1), (assume,1), (reproduction,,3),
(file,3), (offer,,1), (licenses,1), (grant,1)...

scala> time{counts.count()}
Elapsed time: 44354361ns
res15: Long = 540

scala> counts.cache()
res16: counts.type = ShuffledRDD[16] at reduceByKey at <console>:26

scala> time{counts.collect()}
Elapsed time: 12177474ns
res17: Array[(String, Int)] = Array((under,6), (Unless,2), (Contributions),1),
(offer,1), (NON-INFRINGEMENT,,1), (agree,1), (its,3), (event,1),
(intentionally,2), (Grant,2), (have,2), (include,2), (responsibility,,1),
(writing,1), (MERCHANTABILITY,,1), (Contribution,3), (express,1), ("Your"),1),
((i),1), (However,,1), (been,2), (files;,1), (This,1), (stating,1),
(conditions.,1), (non-exclusive,,2), (appropriateness,1), (marked,1), (risks,1),
(any,28), (IS",1), (filed.,1), (Sections,1), (fee,1), (losses),,1), (out,1),
(contract,1), (from,,1), (4.,1), (names,,1), (documentation,,2), (contract,,1),
(unless,1), (below).,1), (wherever,1), (verbal,,1), (ANY,1), (version,1),
(file.,1), (are,6), (no-charge,,2), (2.,1), (assume,1), (reproduction,,3),
(file,3), (offer,,1), (licenses,1), (grant,1)...
scala>

scala> time{counts.count()}
Elapsed time: 27033399ns
res18: Long = 540
```

As expected, the time of collecting the result from a cached RDD is less than performing the same job without caching. However, the difference in this example is not significant. To observe a significant improvement, one can try the similar experiment on a larger RDD. A fairly large text (http://norvig.com/big.txt) file is available on Peter Norvig personal blog (originally from The Project Gutenberg).

# Shared Variables

Spark defines two types of shared variables (https://spark.apache.org/docs/latest/programming-guide.html#shared-variables) to help Spark programs run more efficiently in a cluster. These common shared variables are:

**A. Broadcast Variables:** Broadcast variables allow to keep read-only variable cached on each machine instead of sending a copy of it with tasks. They can be used to give the nodes in the cluster copies of large input datasets more efficiently. Broadcast variables are created from a variable `v` by calling `SparkContext.broadcast(v)`. The broadcast variable is a wrapper around `v`, and its value can be accessed by calling the `value` method:

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

**B. Accumulators:** Accumulators are only added using an associative operation and can, therefore, be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Tasks running on the cluster can add to an accumulator variable using the add method. However, they cannot read its value. Only the driver program can read the accumulator's value. A numeric accumulator can be created by calling SparkContext.longAccumulator() or SparkContext.doubleAccumulator() to accumulate values of type Long or Double, respectively:

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name:
Some(My Accumulator), value: 0)

scala> accum.value
res1: Long = 0

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))

scala> accum.value
res3: Long = 10
```

# 4.5
# Exercise: Page Rank Algorithm

In this exercise, we help you to implement a well-known algorithm called PageRank. This algorithm is already implemented and available as a part of Apache Spark GraphX, however, implementing such algorithms helps you to practice the concepts that we covered in the previous sections.

## Page Rank Algorithm

You are probably familiar with the story of Google and how its founders' algorithm revolutionized the internet search industry. This algorithm that ranks web pages based on the input and output links is called Page Rank (https://en.wikipedia.org/wiki/PageRank). Although information retrieval technologies have been advanced to a great scale, Page Rank is still a prominent algorithm that can be applied to a variety of graph problems.

Assume we have $N$ documents (each document is considered as a page, regardless of its length) and we want to order them based on their ranks. In the first step, all pages receive a rank of $\frac{1}{N}$. Then, Page Rank updates each document's rank by adding contributions from documents liking to it iteratively. On each iteration, each document sends a contribution of $\frac{r}{n}$ to its neighbours, where $r$ is its rank and $n$ is the number of neighbours. It then updates its rank to $\frac{\alpha}{N} + (1-\alpha)\sum c_i$, where the sum is over the contributions it received. The following is the steps you should take to implement a Page Rank algorithm.

### 1. Read the input file

For this task, you need a list of URLs and their outlinks. Note that a URL usually have more than one outlinks. For example, we may use a code like the following:

```
val links = spark.textFile(...).map(...).persist() // RDD of (URL, outlinks)
```

The links should be an RDD where the key part of each entry is the URL and the value is an Array of the outlinks.

### 2. Initial Ranks

Since we do not know much about the documents, let's assume all pages have the same rank. The best value is probably $\frac{1}{N}$ where $\frac{1}{N}$ is the total number of documents.

```
val N = ... // number of documents
var ranks = links.map(...)// RDD of (URL, rank) pairs
```

The outcome should be an RDD that contains a list of distinct URLs and their initial ranks. We will update the ranks in the next steps.

## 3. Set a Termination Criterion

We can think of many ways to stop this iterative algorithm. For example, we can stop it if the ranks converged to some values and do not change dramatically. Another (much simpler) approach is to set a maximum number of iterations:

```
maxIter = 100 // Maximum number of updates
```

## 4. Update Ranks

Until the termination criterion is met, we should update the pages' ranks by propagating $\frac{r}{n}$ where $r$ is the rank of the source page and $n$ is the number of outlinks from the source page. For example:

```
alpha = ... // update coefficient
for (i <- 1 to maxIter) {
// Build an RDD of (outlink, newContribution) pairs
val contribs = links.join(...).flatMap {
(url, (links, rank)) => links.map(dest => (dest, rank/links.size))
}
// Sum contributions by URL and get new ranks
ranks = contribs.reduceByKey(...).mapValues(sum => alpha/N + (1-alpha)*sum)
}
```

In the above example, `alpha`, `rank` and `links.size` are $\alpha$, $r$, and $n$, respectively. The contribs is an RDD that each of its elements is a pair of an outlink and a share of the contribution that it receives from the source document.

The last statement in the loop calculates the sum of the contributions for each outlink and then adds it to the previous rank. Note that the updates can be very small for large values of $\alpha$, while very small $\alpha$ values may cause dramatic changes in every update.

## 5. Return The Results

This is the easiest step:

```
ranks...
```

Now, you can create an input text file, put all the above steps to gather, complete the incomplete parts, and run the code. We deliberately leave this as an exercise for you.

# 4.6
# Graph Processing

## Introduction

Graph databases (https://en.wikipedia.org/wiki/Graph_database) are emerging kinds of databases where data items are represented and stored using graph (https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) structures. They mainly comprise of nodes (vertices) which represent data items and edges which represent the relationship between data items (nodes). In some cases, attributes may be associated with nodes and edges. Unlike the traditional RDBMS, graph databases are concerned with data items' relationships (https://neo4j.com/why-graph-databases/). This way, applications using the graph databases do not need to identify connections among data items utilizing concepts like foreign keys or MapReduce. All common RDBMS operations such as Create, Read, Update and Delete (CRUD) are accessible in a graph database which is built on a graph data model.

Some of the important properties of graph database technologies (https://neo4j.com/why-graph-databases/) are:

- **Graph Storage:** graph databases can be stored in native graph storage (designed exactly for storage/retrieval procedures of graphs), RDBMS, or object-oriented databases.
- **Graph Processing Engine:** there are two types of graph processing methods: ***native*** and ***non-native***.
    - Native processing is the most efficient method as data items (nodes) are physically connected to each other in the database.
    - Non-native processing, on the other hand, processes the CRUD operations by utilizing other means.

### Why graph databases?

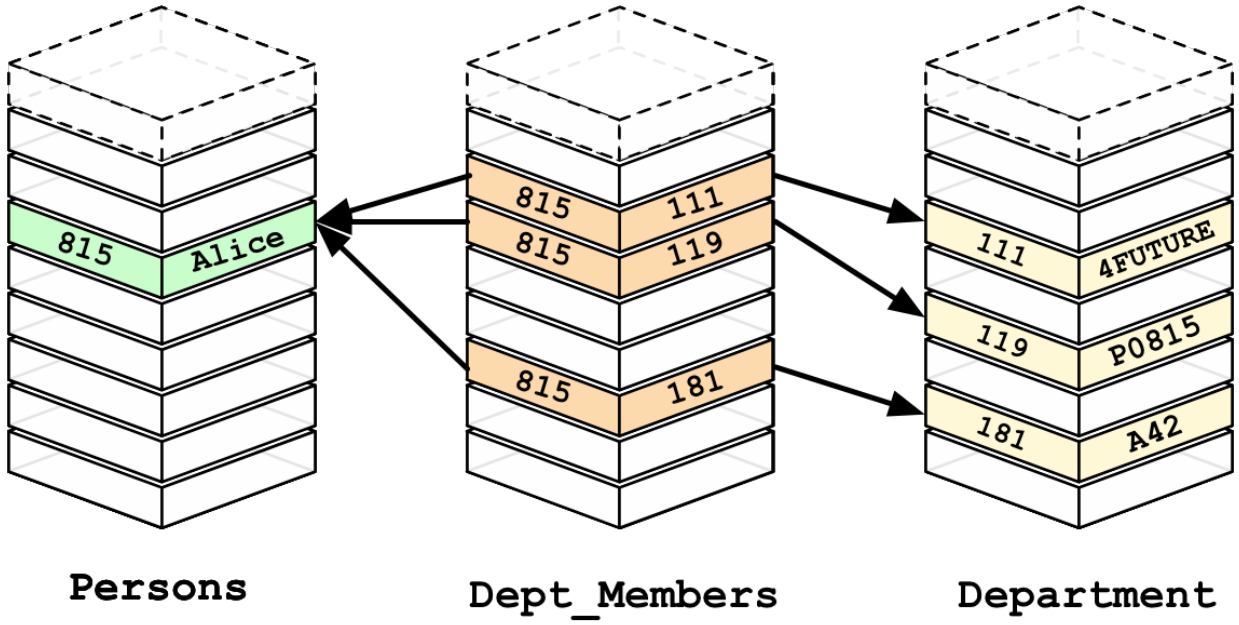
Given the large volumes of data with diverse variety generated recently, traditional relational databases seem to be inefficient at managing relationships and new data structures generated by dynamic data sources. For example, Web2.0, social networks, large distributed systems such as Yahoo, Google, Amazon are producing a huge amount of diverse data. Moreover, data generated by smart sensors and smart devices (in the context of Internet of Things (https://en.wikipedia.org/wiki/Internet_of_things) or IoT) should also be considered as emerging sources of generating dynamic and streaming data.

RDBMS are traditionally incapable of handling the relationships among huge volumes of dynamic data. Also, horizontal scalability of relational data stores has always been a challenge to RDBMS. With regard to dynamic sources of data, other well-known example that generate graph-based data are: Social Graphs (e.g., Facebook, Twitter, Google+, LinkedIn), Endorsement Graphs (e.g., Web Link Graph, Paper Citation Graph), and Location Graphs (e.g., Maps, Power Grids, Telephone Networks). Mentioned sources can produce big graphs of data.

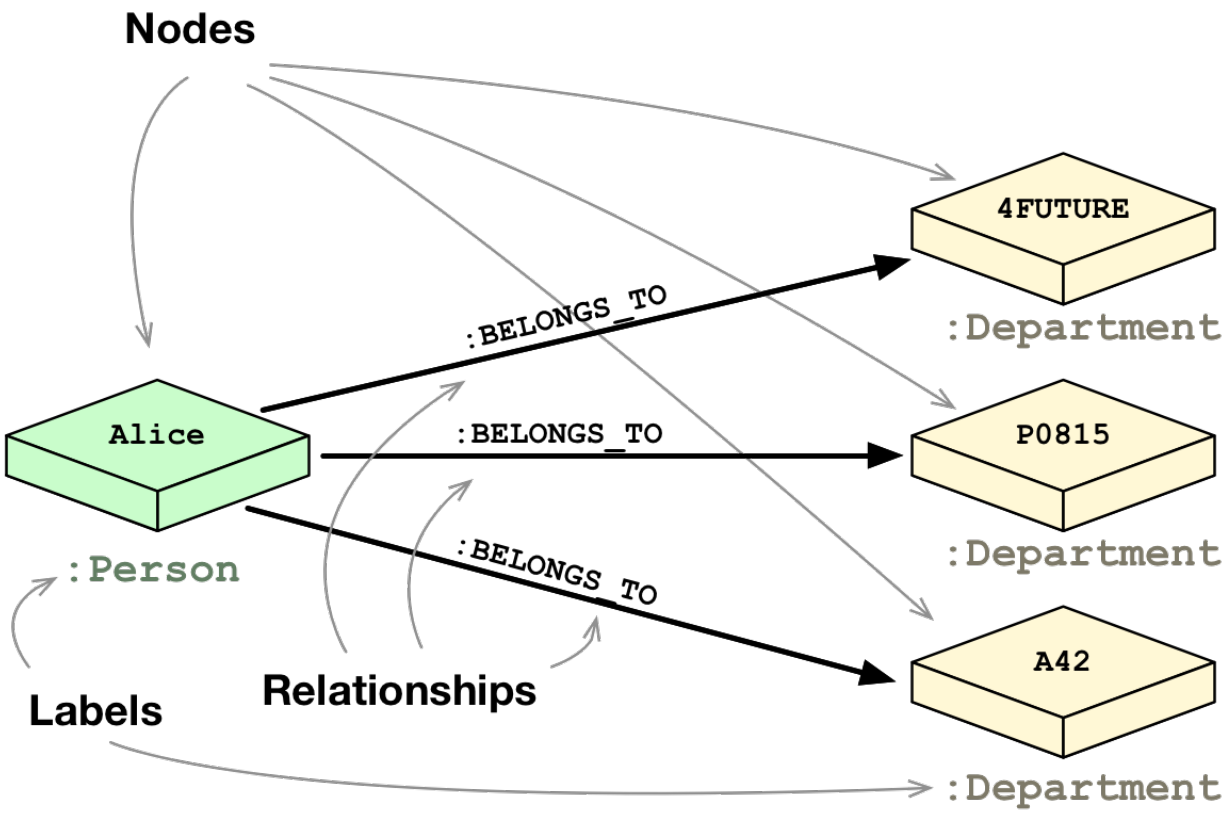
On the other hand, graph databases are capable of analyzing complex relationships among data items. They also give organizations greater ability to move reporting into a real-time or near-real-time mode. Graph databases effectively store data items' relationships, flexible with data growth/change and perform data related operations effectively. Performance, flexibility, and agility are three main characteristics/advantages of graph databases.

# Graph vs. Relational Databases

The following pictures (https://neo4j.com/developer/graph-db-vs-rdbms/) depict the difference between relational and graph databases given the same scenario relating each person to their corresponding department. In relational databases, the references to other tables are demonstrated by referring to their key attributes (usually their primary key) via foreign key attributes.

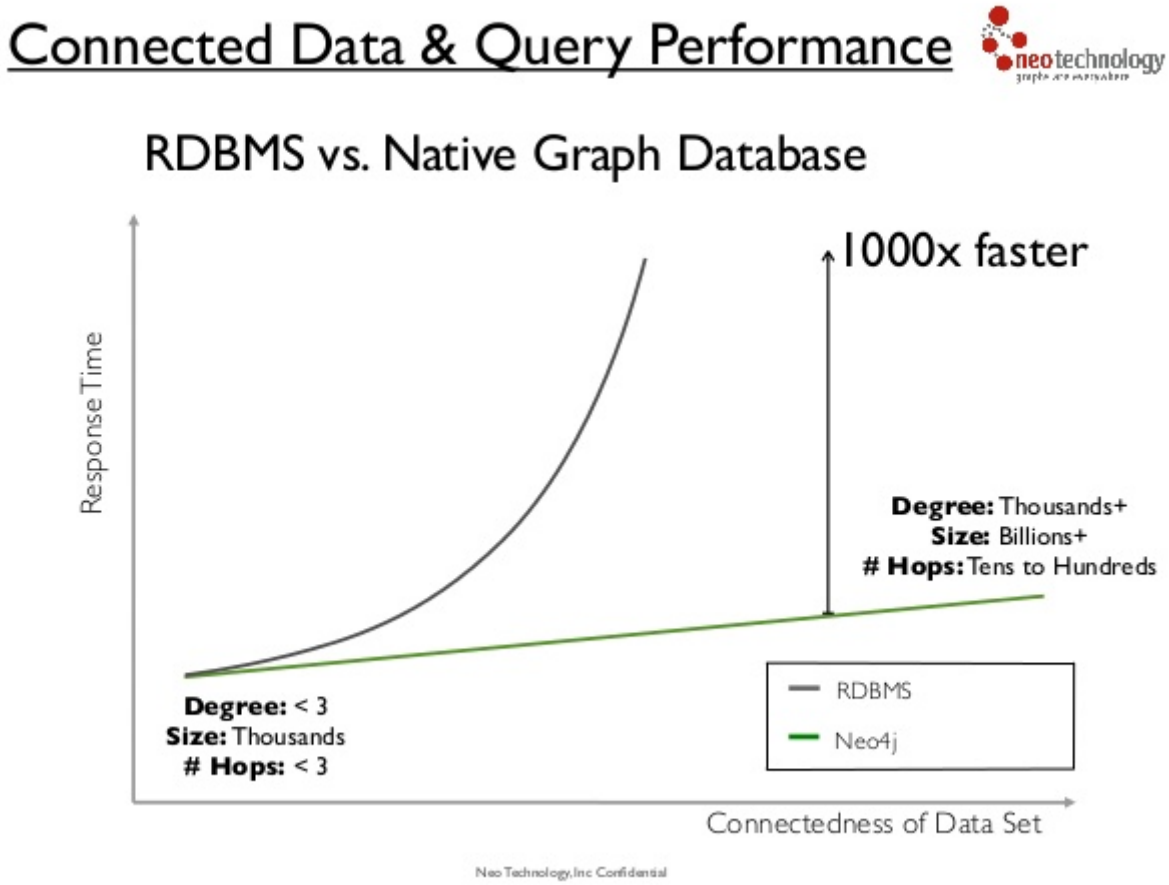



*The relationship between Person and Department Tables in a relational database by accessing their foreign keys in on intermediary table (Dept_Members). Source: neo4j.com*




*The representation of the same scenario in a graph database elaborating on Nodes, Relationships, and their Labels. Source: neo4j.com*

Graph databases outperform the relational databases in performance given the connected data. The following figure compares the two in terms of Query Performance (Response Time) and Connectivity of the sample data (Connectedness of Dataset). It is evident that by increasing the degree of data connectivity, native graph databases (like the Neo4j here) outperform the traditional relational databases by the factor of 1000!




*Graph vs. Relational Databases (Query Performance and Connectivity). Source: neo4j.com*

## Integrating Databases

The general rule of thumbs in data model transformation process (integrating graph databases with relational databases) are elaborated as the following (https://neo4j.com/developer/graph-db-vs-rdbms/):

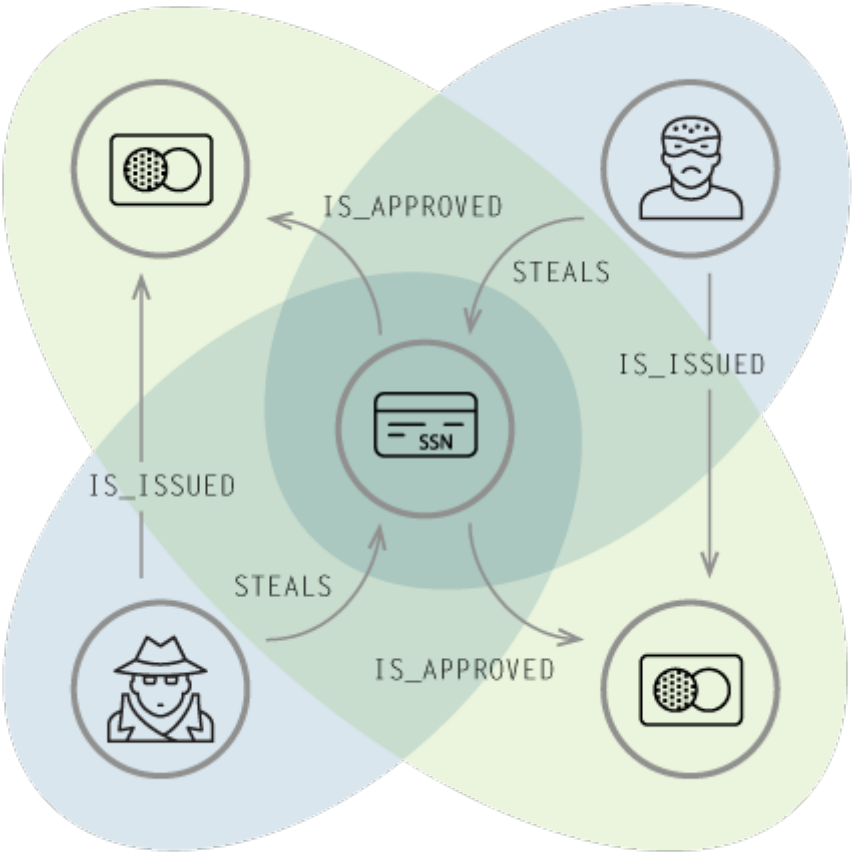
- Tables in relational databases:
  - Entity tables are represented by labels on nodes in graph databases.
  - Each row (record) in tables is represented as a node in graph databases.
  - Table columns are represented as node properties in graph databases.
- Foreign keys in each table in relational databases are replaced with relationships to the other table in a graph database.
- All default values for data items are removed in graph databases.
- Indexed column names in relational databases may be transformed into array properties in graph databases (such as address1, address2, address3)
- Join tables in relational databases are transformed into relationships in graph databases through which their columns are represented as relationship properties.

# Applications

Some of the common use cases of graph databases are briefly discussed below.

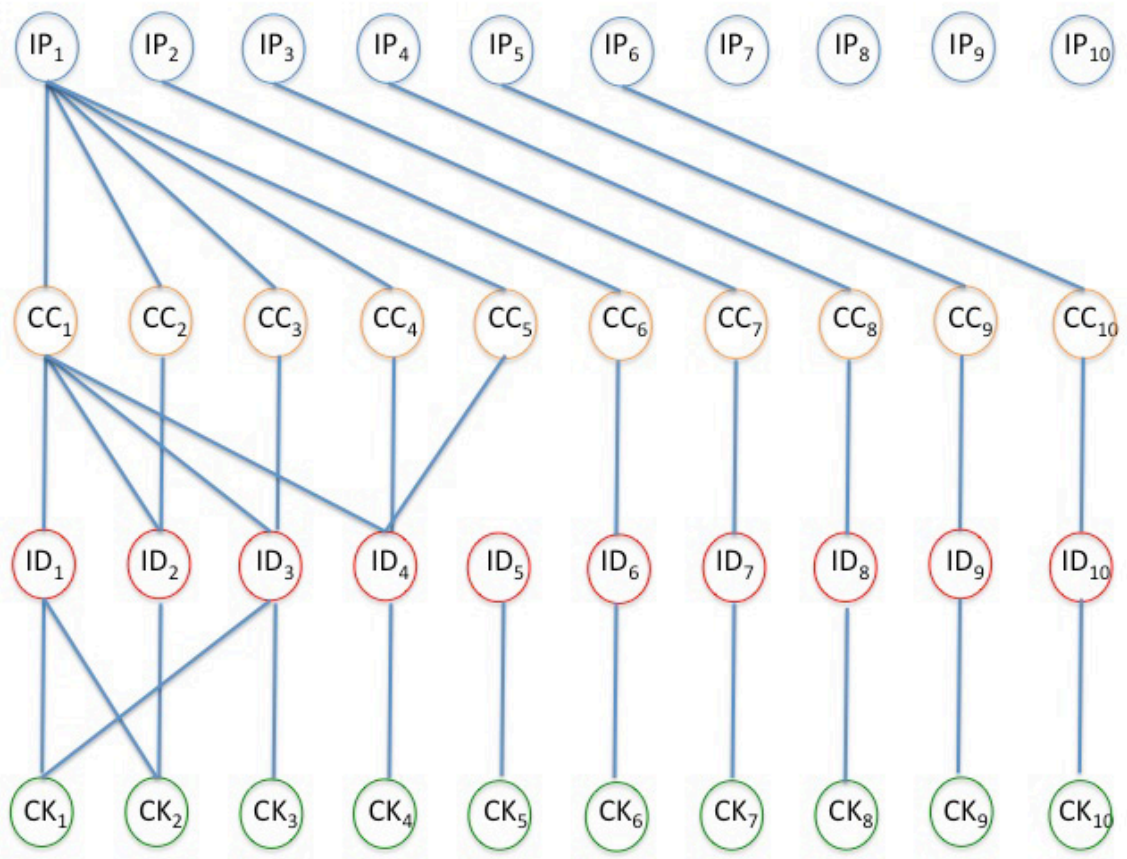
## Fraud detection

Traditional fraud prevention measures focus on discrete data points such as specific accounts, individuals, devices or IP addresses. To unravel such fraud rings, it is important to look beyond individual data points (to the connections that link them). The following is a simple scenario of fraud detection represented in a graph database.




*Graph Database Application: fraud detection (Source: neo4j.com)*

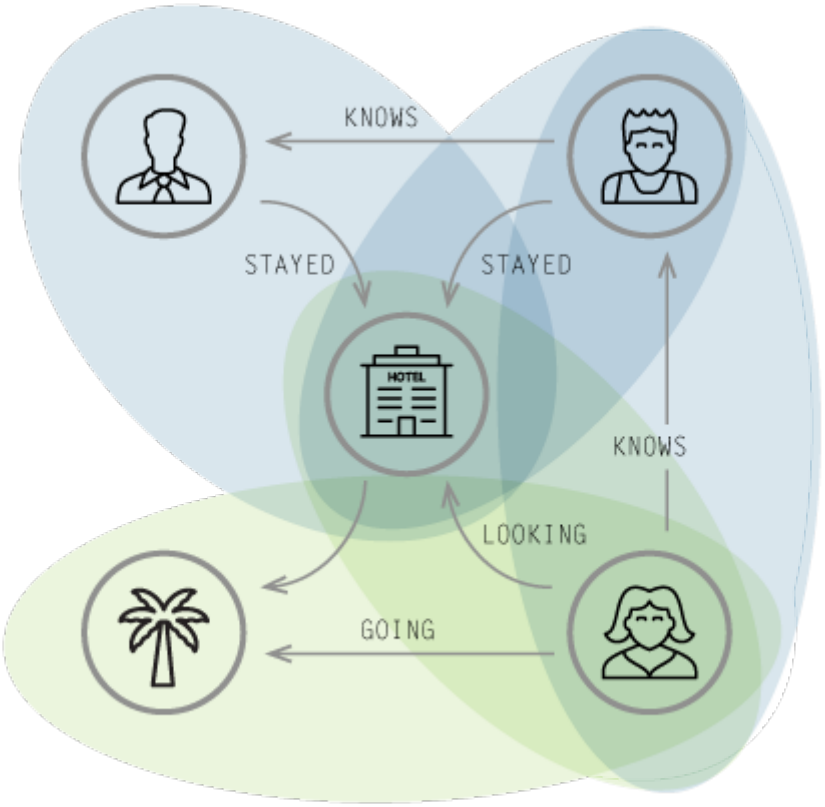
Fraud detection process has some critical issues to address such as complex link analysis to extract suspicious fraud patterns, detecting and preventing fraud ring using real-time link analysis on an interconnected dataset, detecting and discovering fraud patterns given the dynamic growth of fraud rings. A simple example of an E-Commerce fraud can be illustrated in the following figure where online transactions are being performed with the following identifiers: user IDs, IP addresses, geolocations, tracking cookies and credit card numbers. A suspicious pattern can emerge where the relationships between these identifiers are not one-to-one. In the following figure, A graph of a series of transactions from different IP addresses with a likely fraud event occurring from IP1, which has carried out multiple transactions with five different credit cards.




*Fraud detection pattern (Source: neo4j.com)*

## Real-Time Recommendation Engine

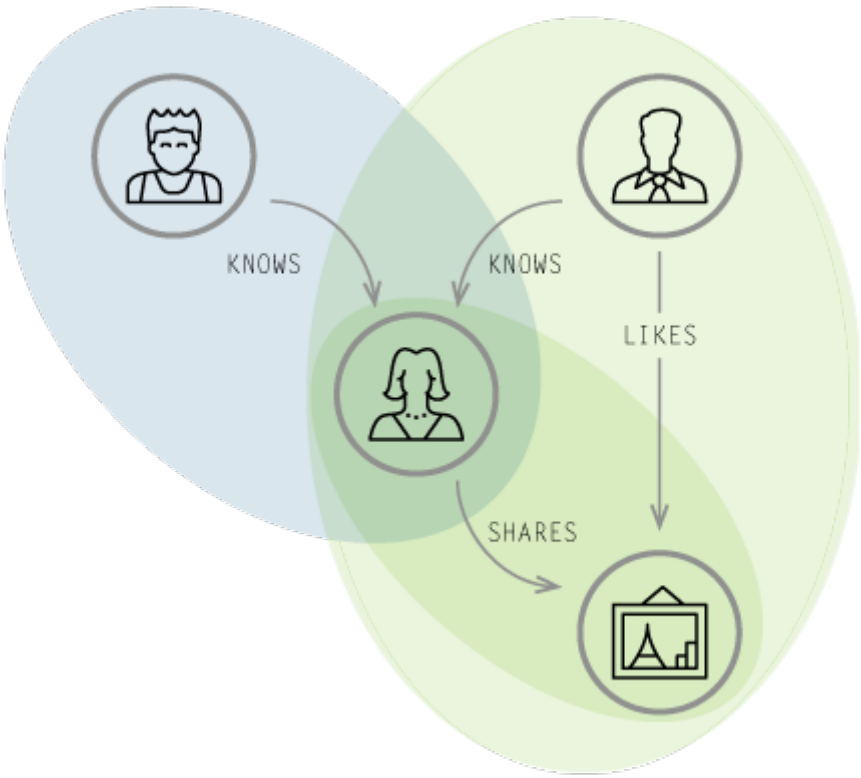
Whether your enterprise operates in the retail, social, services or media sectors, offering your users highly targeted, real-time recommendations (https://en.wikipedia.org/wiki/Recommender_system) are essential to maximizing customer value and staying competitive. Unlike other business data, recommendations must be inductive and contextual to be considered relevant by your end consumers. With a graph database, you are able to capture a customer's browsing behavior and demographics and combine those with their buying history to instantly analyze their current choices and then immediately provide relevant recommendations - all before a potential customer clicks to a competitor's website. The following figure depicts a sample real-time recommendation scenario given interrelationships among different elements.




*Graph Database Application: real-time recommendation engine (source: neo4j.com)*

## Social Media

Given the growing adoption of social media applications, more complex and sophisticated relationships among data items have emerged. This calls for a more efficient social data processing, management and storage approaches. Graph databases are one good fit for this application. The following figure depicts some simple interrelationships among different data items in a very small social media environment where some people know one another and share some topics.




*Graph Database Application: social media (source: neo4j.com)*

Other graph databases applications can be mentioned but not limited to: Web Browsing, Portfolio Analytics, Gene Sequencing, Mobile Social Applications, Network and IT Operations, Content Management and Access Control, Insurance Risk Analysis, Network Cell Analysis, and Bio-Informatics.

# Large-Scale Challenges

As mentioned earlier, interconnected data items have been generated from diverse sources like social networks, transportation networks, and so forth which are best represented using graph algorithms. However, in developing high-scale graph processing systems the following challenges should be addressed (https://www.researchgate.net/publication/308919426_BIG_GRAPH_TOOLS_TECHNIQUES_ISSUES_CHALLENGES_AND_FUTURE_DIRECTIONS):

- **High-degree vertex:** Graphs with high-degree vertices are computationally challenging, contribute heavily to communication and storage overhead and are difficult to partition.
- **Sparseness:** Splitting sparse graph demands more communication, computation, and synchronization.
- **Data-driven computations:** Graph computations are usually data-driven. The computations executed by a graph algorithm are dictated by the vertex and edge structure of the graph rather than being directly represented in code. Since the structure of computations of the algorithm is not known a priori, the parallelism based on the partitioning of computation can be challenging to represent.
- **Unstructured problems:** Irregular structure of graph data makes it difficult to extract parallelism by partitioning the problem data.
- **In-memory challenge:** Large-scale graphs usually do not fit in a single memory location. On the other hand, the graph data should reside in the RAM, rather than SDD or HDD, to minimize the

response time.

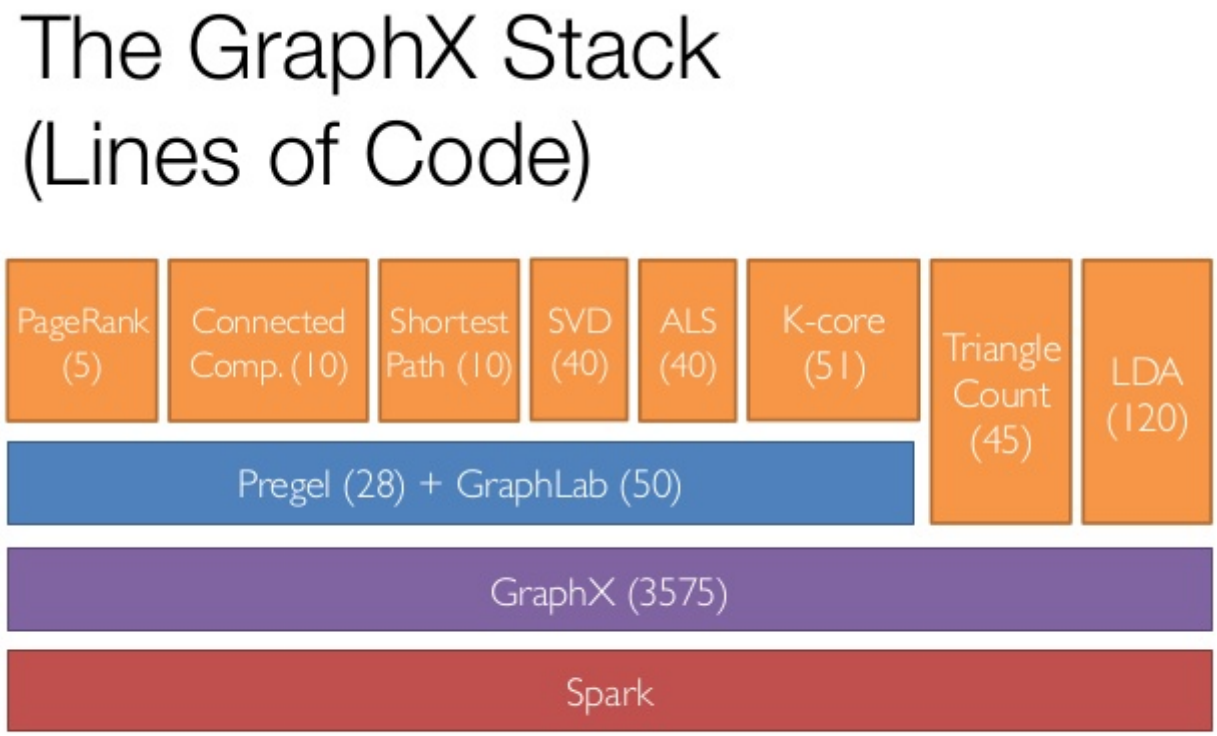
- **Poor locality:** Because graphs represent the relationships between entities and because these relationships may be irregular and unstructured, the computations and data access patterns tend not to have very much locality. Performance in contemporary processors is predicated upon exploiting locality. Thus, high performance can be hard to obtain for graph algorithms, even on serial machines.
- **Communication overhead:** As mentioned above, the existence of high-degree vertices forces communication overheads.
- **Load balancing:** We should pay extra attention to load balancing when processing some special power-law graphs.

To address the aforementioned challenges, some useful systems were introduced. In the following, we will review two main graph-based systems in this context: Apache GraphX along with Pregel, and Neo4j with its Cypher.

# 4.6.1
# Introduction to Spark GraphX

## Introduction

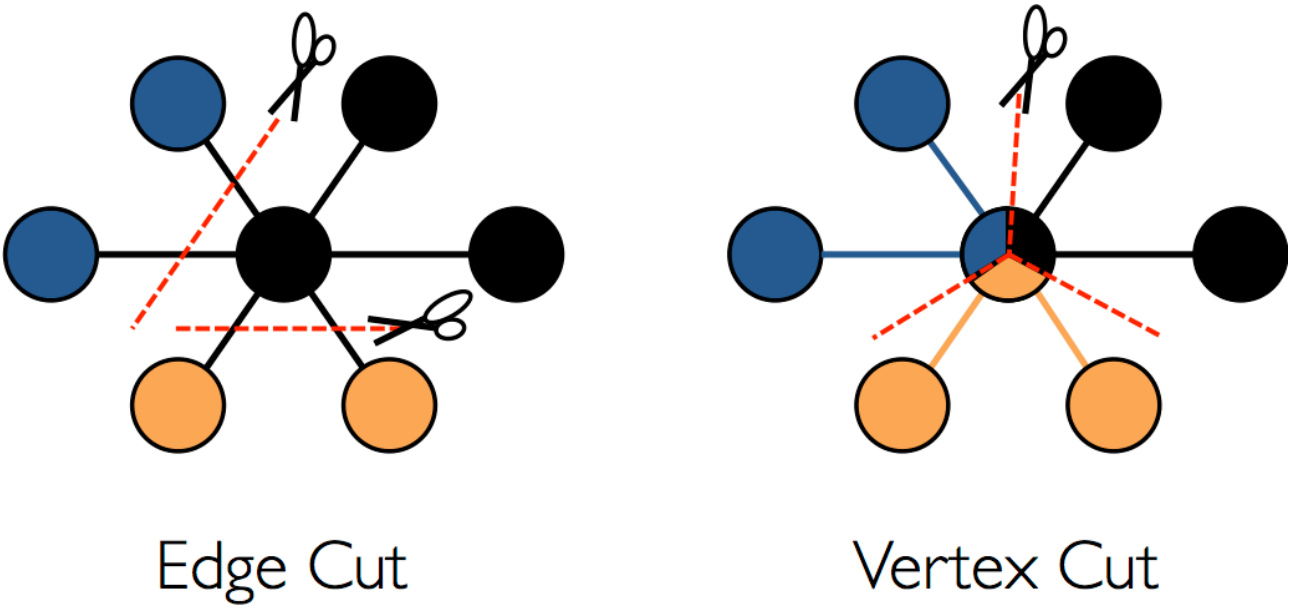
GraphX (https://spark.apache.org/graphx/) is Apache Spark's API for graphs (e.g., Web-Graphs and Social Networks) and graph-parallel computation (e.g., PageRank, Collaborative Filtering). Technically, GraphX extends Apache Spark's Resilient Distributed Dataset (RDD) with a Resilient Distributed Property Graph (RDPG). This means that GraphX proposed a new graph abstraction which is a directed multigraph with properties of vertices and edges are attached to them. The graph computation in GraphX is supported by introducing a set of operators (e.g., `subgraph, joinVertives, aggregateMessages`) and an optimized version of **Pregel API** (https://pregel.com.au/). Pregel is a programming model by Google which specifically targets large-scale graph problems. GraphX is a thin layer on top of the Apache Spark general-purpose dataflow framework. The following is a demonstration of the GraphX stack and its relation with Apache Spark components:




*Apache Spark GraphX and its components*

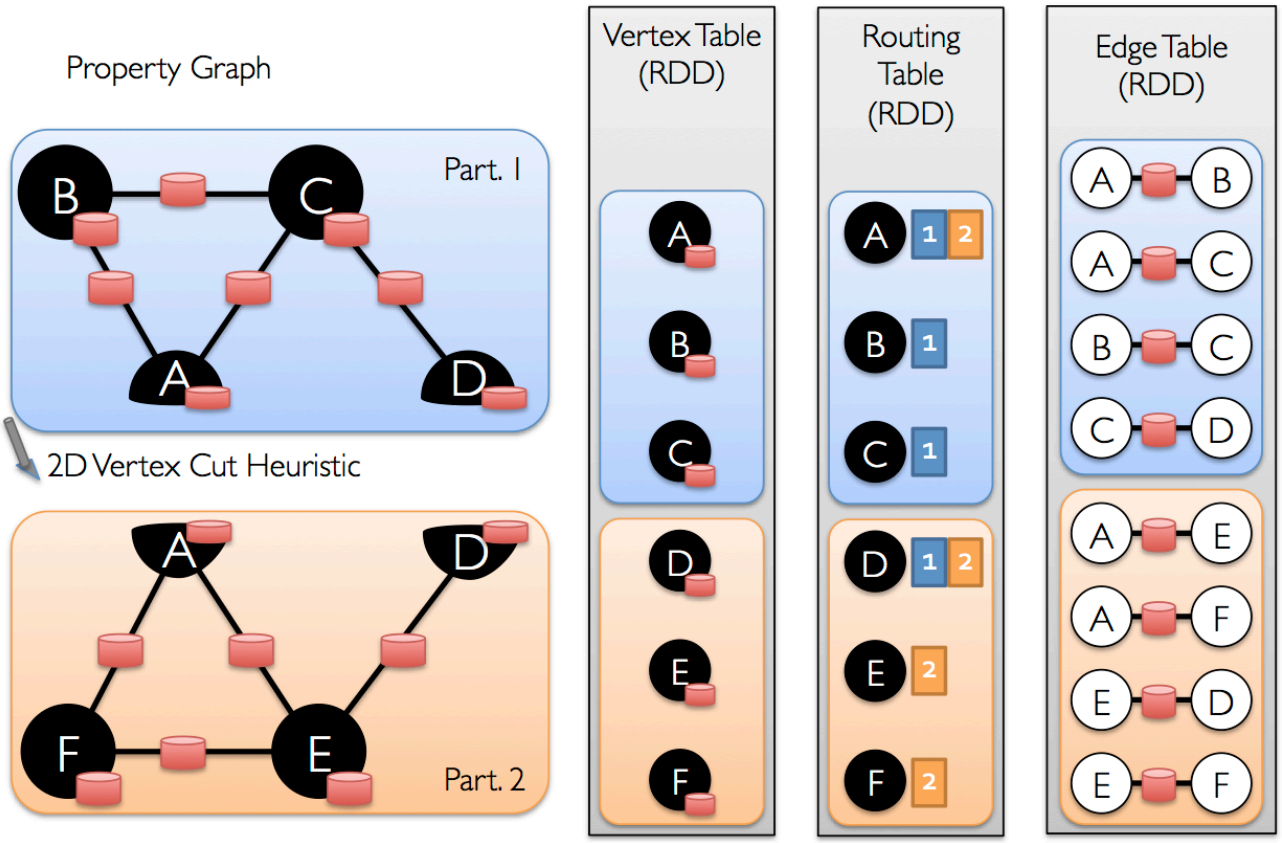
# Optimized Representations

Graphs, in general, tend to have skewed distributions. Therefore, proper **graph partitioning** approaches become critical. Graph partitioning (https://spark.apache.org/docs/latest/graphx-programming-guide.html#optimized-representation) algorithms fall into two key categories: **edge-cut** partitioning and **vertex-cut** partitioning. Vertex cut partitioning schemes have been observed to perform well on many large natural graphs (https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf). The following figure depicts the simple concept of the two methods:




*Graph Partitioning: edge cut vs. vertex cut (source: Apache Spark)*

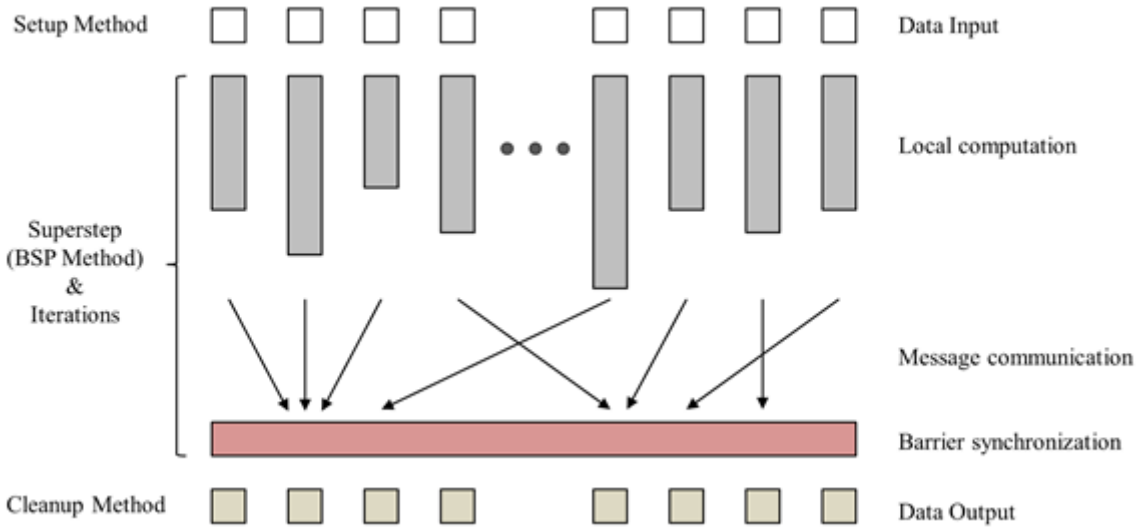
Apache GraphX adopts the vertex-edge partitioning approach which helps with even distribution of computation workload. It also decreases the overheads associated with the communication and storage tasks. Vertex-cut scheme divides high-degree vertices along with partitions and evenly associates edges to a machine. This way, the number of times each vertex is cut is minimized. To assign edges to machines, Apache GraphX provides the *PartitionStrategy* along with different heuristics. Developers are able to select which strategy they want to partition the graph using the `Graph.partitionBy` operator. The following displays a sample encoding process of a property graph as RDDs:




*Example: a property graph as RDDs (source: Apache Spark)*

## Bulk Synchronous Parallel

The Bulk Synchronous Parallel (https://en.wikipedia.org/wiki/Bulk_synchronous_parallel) (BSP) abstract computer is a parallel processing model for massive scientific and iterative algorithms. It was introduced by Leslie Valiant (https://people.seas.harvard.edu/~valiant/) at Harvard University during the 1980s. BSP is an easy and flexible programming model, as compared with traditional models of Message Passing. This computer consists of a collection of processors, each with its own memory. It is a distributed-memory computer. Algorithms designed for a BSP computer are portable; they can be run efficiently on many different parallel computers. The following figure illustrates a sample flow of tasks in one BSP computer.

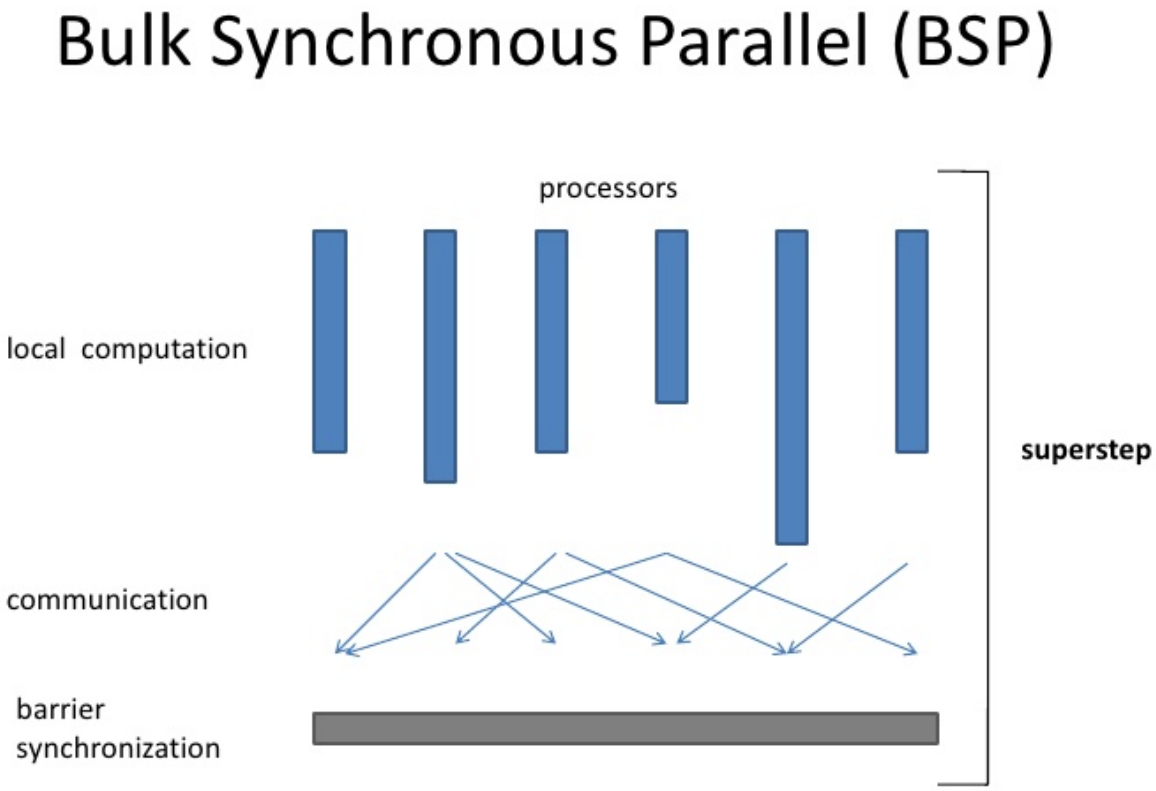


*Bulk Synchronous Parallel model*

Each BPS algorithm consists of a sequence of **super-steps**. The main components of BSP [are](https://en.wikipedia.org/wiki/Bulk_synchronous_parallel):

- **Concurrent computation** - each involved processor performs their local computations. Therefore, they get access to their local memory for the calculations. A computation super-step consists of many small steps, such as the floating-point operations (flops) addition, subtraction, multiplication, division.
- **Communication** - to get benefit from the remote data storage facilities, processes interchange data among themselves using the communication operations.
- **Barrier synchronization** - this element synchronizes the flow of tasks among all processes. All participating processes should be waiting here for others to arrive and unify the final output.

These elements are depicted in the following figure:




*Bulk Synchronous Parallel components*

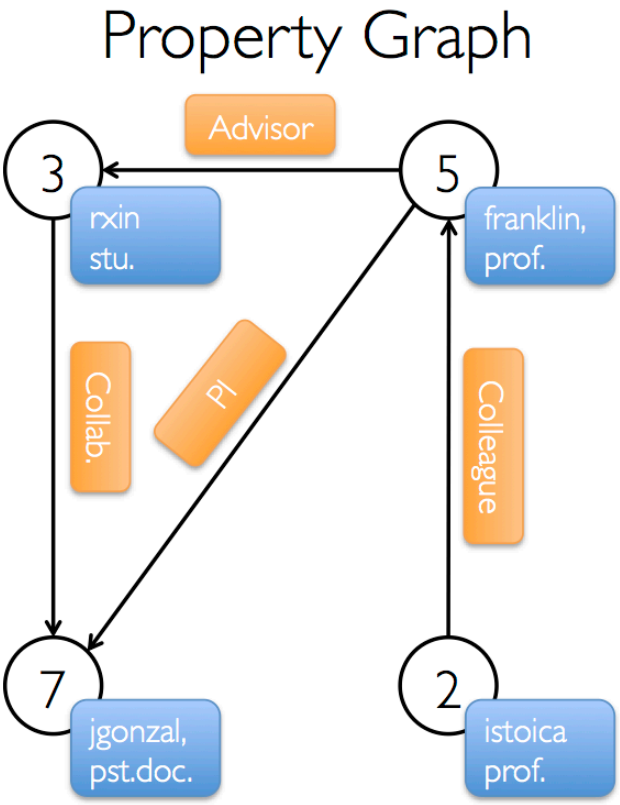
We will discuss GraphX more in the activities.

# 4.6.2
# Activity: GraphX Basics

Apache Spark GraphX supports a wide range of operations. In the following, we discuss some of the most frequently used operations in brief. In the next activity, we will use some of these operators to solve an analytical problem.

## A Property Graph Example

The following figure represents a sample property graph of several collaborators on a project. The nodes (vertices) contain *username* and *occupation* properties. This is only one example of countless applications of graphs. We use this example to discuss the main GraphX concepts and operations (this example has been widely used in GraphX [documentation](https://spark.apache.org/docs/1.1.1/graphx-programming-guide.html) and tutorials).




Vertex Table

| Id | Property (V) |
|---|---|
| 3 | (rxin, student) |
| 7 | (jgonzal, postdoc) |
| 5 | (franklin, professor) |
| 2 | (istoica, professor) |

Edge Table

| SrcId | DstId | Property (E) |
|---|---|---|
| 3 | 7 | Collaborator |
| 5 | 3 | Advisor |
| 2 | 5 | Colleague |
| 5 | 7 | PI |

## 1. Create a Graph

The first step in the graph analysis is to create one or more graphs. The simplest way to create a graph RDD is to create vertices and edges RDDs. Then, a Graph object can be easily created using those values. For example, we can create a simple graph representing the above example:

```
scala> import org.apache.spark.graphx._
```

```
import org.apache.spark.graphx._

scala> import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD

scala> val users: RDD[(VertexId, (String, String))] =
     |     sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal",
"postdoc")),
     |                           (5L, ("franklin", "prof")), (2L, ("istoica",
"prof"))))
users: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, (String,
String))] = ParallelCollectionRDD[14] at parallelize at <console>:32

scala> val relationships: RDD[Edge[String]] =
     |     sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),
     |                           Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
relationships: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
ParallelCollectionRDD[15] at parallelize at <console>:32

scala> val defaultUser = ("John Doe", "Missing")
defaultUser: (String, String) = (John Doe,Missing)

scala> val graph = Graph(users, relationships, defaultUser)
graph: org.apache.spark.graphx.Graph[(String, String),String] =
org.apache.spark.graphx.impl.GraphImpl@5db16956

scala> graph.vertices.collect()
res3: Array[(org.apache.spark.graphx.VertexId, (String, String))] =
Array((2,(istoica,prof)), (3,(rxin,student)), (7,(jgonzal,postdoc)),
(5,(franklin,prof)))

scala> graph.edges.collect()
res4: Array[org.apache.spark.graphx.Edge[String]] = Array(Edge(3,7,collab),
Edge(5,3,advisor), Edge(2,5,colleague), Edge(5,7,pi))
```

In the above example, `users` is an RDD containing `VertexId` (a number) and a tuple which itself contains two strings: users' name and occupation (see Vertex Table). We use this as our vertex RDD. The relationships is another RDD which stores an `Array` of edges information. Each element of this `Array` is an `Edge` containing two numbers (link source `VertexId` to destination `VertexId`) as well as a string indicating the relationship between the linked users (see Edge table). The `defaultUser`, as its name indicates, is the default node value which will be used in the case of the missing data. The `Graph` statement creates our graph, and finally, `graph.vertices.collect()` and `graph.edges.collect()` respectively return the vertices and edges of the created graph.

## 2. Querying with Filters

In the previous example, we queried all edges and vertices of the graph. In practice, we are interested in some parts of a massive graph. In such cases, we need to filter out the results of a query. For example, the following code depicts the usage of the `graph.vertices` and `graph.edges` filters in querying over vertices or edges of the graph. This way, we deconstruct the graph into vertex and edge views:

```
scala> graph.vertices.filter{ case (id, (name, pos)) => pos == "postdoc"
```

```
}.take(10)
res4: Array[(org.apache.spark.graphx.VertexId, (String, String))] =
Array((7,(jgonzal,postdoc)))
```

The above example returns the vertices with the position equal to postdoc. We could write this line even shorter:

```
scala> graph.vertices.filter{_._2._2 == "postdoc" }.take(10)
res5: Array[(org.apache.spark.graphx.VertexId, (String, String))] =
Array((7,(jgonzal,postdoc)))
```

Note that `_._2._2` in this example refers to the position of the user. We could write other statements to only return the name of those users or just count the number of users with postdos position:

```
scala> graph.vertices.filter {_._2._2 == "postdoc" }.take(10).map(_._2._1)
res7: Array[String] = Array(jgonzal)

scala> graph.vertices.filter {_._2._2 == "postdoc" }.count()
res8: Long = 1
```

We are also able to filter over edge contents. For example, let's count the number of relationships that the source user's Id is larger than the destination user's Id:

```
scala> graph.edges.filter(e => e.srcId > e.dstId).count
res9: Long = 1
```

Only one users! But who is it? We can find his/her name by joining the filtered results with the `users` table. For example:

```
scala> graph.edges.filter(e => e.srcId >
e.dstId).map(e=>(e.srcId,e.attr)).join(users).map(_._2._2._1).take(10)
res10: Array[String] = Array(franklin)
```

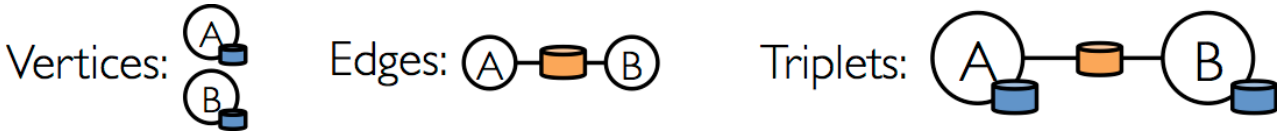
We will discuss the join operation further in the following parts.

## 3. Triplet View

GraphX also supports the triplet view. Triplet view performs a join over the vertex and edge properties to produce one `RDD[EdgeTriplet[VD, ED]]` with instances of `EdgeTriplet` class. In SQL expression, this can be illustrated as:

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

Or as the following figure:




Triplet view logically connects the Edge and Vertex properties. The connection is made by yielding the

RDD[EdgeTriplet[VD, ED]] that contains instances of the `EdgeTriplet` class. The `EdgeTriplet` class adds the given members containing the source and destination properties respectively and hence extends the `Edge` class. An edge triplet represents an edge, along with the vertex attributes of its neighboring vertices. It has the following properties:

- `srcId` The source vertex id.
- `srcAttr` The source vertex attribute.
- `dstId` The destination vertex id.
- `dstAttr` The destination vertex attribute.
- `attr` The edge attribute.

Use the triplets view to create an RDD of facts to find out all the relationship in the graph.

The usage of triplet view of a graph to produce a collection of users and their relations' strings is explained in the following code segment:

```
scala> graph.triplets.map(triplet => triplet.srcAttr._1 + " is the " +
triplet.attr + " of " + triplet.dstAttr._1).collect.foreach(println(_))
rxin is the collab of jgonzal
franklin is the advisor of rxin
istoica is the colleague of franklin
franklin is the pi of jgonzal
```

# Graph operators

In the previous section, we brought you a high-level overview of the GraphX operations. In the following, we provide more information about the GraphX's core and optimized operators (https://spark.apache.org/docs/latest/graphx-programming-guide.html#summary-list-of-operators) as defined in Graph (https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Graph) and GraphOps (https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.GraphOps). Main operators can be classified into the following categories:

- **View operators -** such as `vertices, edges,` and `triplets` provide access to graph parts as collections
- **Property operators -** which produce a new graph with vertex/edge properties. `mapVertices, mapEdges,` and `mapTriplets` are the most common property operators.
- **Structural operators -** that are focused on the operations regarding the generated graph's structure.
  - `reverse:` returns a new graph with all the edge directions reversed,
  - `subgraph:` takes vertex and edge predicates and returns the graph containing only the vertices that satisfy the vertex predicate and edges that satisfy the edge predicate and connect vertices that satisfy the vertex predicate.
  - `mask:` constructs a subgraph by returning a graph that contains the vertices and edges that are also found in the input graph.
  - `groupEdges:` merges parallel edges.
- **Join operators -** these operators are useful in combining data items from external collections (RDDs) with graphs.
  - `joinVertices:` joins the vertices with the input RDD and returns a new graph with the vertex properties obtained by applying a `map` to the result of the joined vertices.
  - `outerJoinVertices:` similar to `joinVertices` except that the `map` is applied to all vertices and can change the vertex property type.

- **Aggregation operators -** which are concerned with aggregating information about the neighborhood of each vertex.
  - `mapReduceTriplets:` takes a `map` which is applied to each triplet and can yield messages destined to either (none or both) vertices in the triplet.
  - `degrees, inDegrees` and `outDegrees:` compute degree information.
  - `collectNeighborIds` and `collectNeighbors:` collect neighboring vertices and their attributes at each vertex.
- **Pregel API -** to be used in the iterative process (explained bellow)

In addition to these operators, some basic yet popular graph algorithms are also implemented in GraphX package. For example:

- `pageRank:` returns the PageRank (https://en.wikipedia.org/wiki/PageRank) of all vertices in a directed graph.
- `connectedComponents:` returns all connected components (https://en.wikipedia.org/wiki/Connected_component_(graph_theory)).
- `triangleCount:` counts the number of triangles (https://en.wikipedia.org/wiki/Triangle_graph) in a given graph.
- `stronglyConnectedComponents:` finds and returns the strongly connected (https://en.wikipedia.org/wiki/Strongly_connected_component) parts of a graph.

# Pregel API

Prominent graph-based algorithms are iterative. This means vertex properties are related to their neighbor vertices' properties and so forth. Pregel is an iterative graph processing model (https://www.researchgate.net/profile/James_Dehnert/publication/221257383_Pregel_A_system_for_large-scale_graph_processing/links/00b7d537c615821fa4000000.pdf) for large-scale graph problems which was introduced by Google. It uses a sequence of iterations of messages passing between vertices in a graph. Pregel model is a high-level program organization based on Bulk Synchronous Parallel (BSP) model. BSP is explained later. GraphX provides a variant of Pregel API for graphs.

# 4.6.3 Activity: Spark GraphX (Flight Example)

In this activity (http://sparktutorials.net/analyzing-flight-data:-a-gentle-introduction-to-graphx-in-spark), we learn to load data from the disk into Spark, create a property graph, and perform indicative analytics in response to a set of Business Intelligence (BI) questions. For this example, we use a compressed cvs file called `flights_2008.csv.bz2`, comprising flight data information. The dataset is already stored in the provided VM in **`$HOME`**`/Documents/Datasets/flights_2008.csv.bz2`. In this example, **`$HOME`** is `/home/user`.

Note that our dataset consists of the following columns: `Year, Month, DayofMonth, DayOfWeek, DepTime, CRSDepTime, ArrTime, CRSArrTime, UniqueCarrier, FlightNum, TailNum, ActualElapsedTime, CRSElapsedTime, AirTime, ArrDelay, DepDelay, Origin, Dest, Distance, TaxiIn, TaxiOut, Cancelled, CancellationCode, Diverted, CarrierDelay, WeatherDelay, NASDelay, SecurityDelay,` and `LateAircraftDelay`.

## 1. Loading Libraries

We need to load a number of libraries to be able to complete this exercise:

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import scala.util.hashing.MurmurHash3
import org.apache.spark.SparkContext._
```

To make Spark less verbose, we can issue `sc.setLogLevel("OFF")`.

## 2. Reading Data

We can read and load the data using `spark.read.option`. Notice the provided dataset has a header that should be handled properly.

```
scala> val df =
spark.read.option("header","true").csv("/home/user/Documents/Datasets/flights_20
08.csv.bz2")
df: org.apache.spark.sql.DataFrame = [Year: string, Month: string ... 27 more
fields]
```

Let's see what are the column names and types:

```
scala> df.printSchema()
root
 |-- Year: string (nullable = true)
 |-- Month: string (nullable = true)
 |-- DayofMonth: string (nullable = true)
 |-- DayOfWeek: string (nullable = true)
 |-- DepTime: string (nullable = true)
 |-- CRSDepTime: string (nullable = true)
```

```
|-- ArrTime: string (nullable = true)
|-- CRSArrTime: string (nullable = true)
|-- UniqueCarrier: string (nullable = true)
|-- FlightNum: string (nullable = true)
|-- TailNum: string (nullable = true)
|-- ActualElapsedTime: string (nullable = true)
|-- CRSElapsedTime: string (nullable = true)
|-- AirTime: string (nullable = true)
|-- ArrDelay: string (nullable = true)
|-- DepDelay: string (nullable = true)
|-- Origin: string (nullable = true)
|-- Dest: string (nullable = true)
|-- Distance: string (nullable = true)
|-- TaxiIn: string (nullable = true)
|-- TaxiOut: string (nullable = true)
|-- Cancelled: string (nullable = true)
|-- CancellationCode: string (nullable = true)
|-- Diverted: string (nullable = true)
|-- CarrierDelay: string (nullable = true)
|-- WeatherDelay: string (nullable = true)
|-- NASDelay: string (nullable = true)
|-- SecurityDelay: string (nullable = true)
|-- LateAircraftDelay: string (nullable = true)
```

# 3. Creating a Graph

Before performing any analysis on our data, we need to convert our RDD to a Graph. Indeed, we should change our data to a format GraphX can understand. In other words, we need to define the graph vertices, edges and default vertices (in the case of missing values). In this example, the airports are the nodes and the flights are the edges.

## 3.1 Creating Vertices

Let's start with the nodes. Our objective is to find all the airports in the datasets, regardless of being an origin to a flight or just a destination. Since each item of our RDD has both origin and destination information, we firstly create lists of all origin and destination airports. Therefore, we will have a large number of tuples. We do not like tuples here! So, we flatMap these tuples to create a long list of all available airports. Of course, there might be a lot of redundancies.

```
val airportCodes = df.select($"Origin", $"Dest").flatMap(x =>
Iterable(x(0).toString, x(1).toString))
```

In the above statement, `Iterable(x(0).toString, x(1).toString)` converts the airport codes to strings. To clarify the above process, let's take a quick look of the airport codes before and after applying the aforementioned flatMap:

```
scala> df.select($"Origin", $"Dest").take(10)
res9: Array[org.apache.spark.sql.Row] = Array([IAD,TPA], [IAD,TPA], [IND,BWI],
[IND,BWI], [IND,BWI], [IND,JAX], [IND,LAS], [IND,LAS], [IND,MCI], [IND,MCI])
```

```
scala> df.select($"Origin", $"Dest").flatMap(x => Iterable(x(0).toString,
x(1).toString)).take(10)
res10: Array[String] = Array(IAD, TPA, IAD, TPA, IND, BWI, IND, BWI, IND, BWI)
```

As it is shown in the results, many of the airport codes are repeated. Indeed, we (almost!) have two airport codes for each flight, but in reality, we have much less number of airports. To have a better figure, let's compare the size of this RDD and the number of distinct values:

```
scala> airportCodes.count()
res11: Long = 14019456

scala> airportCodes.distinct.count()
res12: Long = 305
```

Now you know what I am talking about!

As mentioned before, we need to find the list of unique airport codes. In addition, we should convert these strings to numbers to be able to create our graph. MurmurHash helps us to perform this conversion.

```
scala> val airportVertices: RDD[(VertexId, String)] =
airportCodes.rdd.distinct().map(x => (MurmurHash3.stringHash(x), x))
airportVertices: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
String)] = MapPartitionsRDD[86] at map at <console>:37
```

Are you curious what this hashing looks like? So am I!

```
scala> airportVertices.take(10)
res14: Array[(org.apache.spark.graphx.VertexId, String)] =
Array((455752397,FWA), (-1288021106,SMX), (-413129232,SPS), (-1841167711,PIA),
(-2105183870,BMI), (1834448609,HLN), (891784594,SUN), (-1852287689,RIC),
(736678150,PSE), (1226329766,SLC))
```

The last thing remains is to define a default value for the missing airport codes:

```
scala> val defaultAirport = ("Missing")
defaultAirport: String = Missing
```

## 3.2 Creating Edges

Now, we have to create edges which are the flights. Finding the flights between two airports is very easy:

```
scala> val flightsFromTo = df.select($"Origin",$"Dest")
flightsFromTo: org.apache.spark.sql.DataFrame = [Origin: string, Dest: string]
```

I believe there could be multiple flights from a unique pair of origin-destination airports. Let's validate my hypothesis:

```
scala> flightsFromTo.count()
17/06/12 17:11:39 WARN Utils: Truncated the string representation of a plan
since it was too large. This behavior can be adjusted by setting
```

```
'spark.debug.maxToStringFields' in SparkEnv.conf.
res2: Long = 7009728

scala> flightsFromTo.distinct.count()
res3: Long = 5366
```

To create a sound graph, we have two options: 1) create one edge for each unique flight or 2) create only one edge for a unique pair of airports and use the number of similar flight as the edge weight. We choose the second option. As a result, we have to count the number of similar flights. Note that `Edge()` expects three values, the first two are the vertices and the last one is the weight. In this example, the vertices are the hashed airport codes and the weight is nothing but the number of flights from the origin (the first vertex) to the destination (the second vertex). Since we hashed the vertices, we should again use their hashed values in the `Edge()` function.

```
scala> val flightEdges = flightsFromTo.map(x =>
((MurmurHash3.stringHash(x(0).toString),MurmurHash3.stringHash(x(1).toString)),
1)).rdd.reduceByKey(_+_).map(x => Edge(x._1._1, x._1._2,x._2))
flightEdges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] =
MapPartitionsRDD[40] at map at <console>:39
```

Let's take a quick look at the results:

```
scala> flightEdges.take(10)
res4: Array[org.apache.spark.graphx.Edge[Int]] =
Array(Edge(1095539962,178004914,2205), Edge(1567376521,1710100676,925),
Edge(-1576863504,52301407,58), Edge(1917476200,813811672,3729),
Edge(-1928985797,-216730884,1379), Edge(-1020692585,1058030826,94),
Edge(-1787161043,178004914,509), Edge(1226329766,1146947515,2),
Edge(-1515187592,1710100676,3575), Edge(1226329766,1554094361,338))
```

## 3.3 Creating Graph

Now, we have our vertices and edges. Therefore, we can create our graph, finally!

```
scala> val graph = Graph(airportVertices, flightEdges, defaultAirport)
graph: org.apache.spark.graphx.Graph[String,Int] =
org.apache.spark.graphx.impl.GraphImpl@60ee753f
```

For a better performance, we can also issue `graph.persist()` to make sure that the resulting graph is always available.

# 4. Basic Analysis

OK! Now we explore our graph to answer some simple questions. Then, we can go further and perform some more complex analysis. Let's find the answers of the following questions:

## 4.1 How many airports we have?

We already answered this question (look above!) However, let's find its answer with basic graph operations. The number of airports is equivalent to the number of nodes:

```
scala> graph.numVertices
res6: Long = 305
```

## 4.2 How many unique routes we have?

Here, we are not interested in the number of flights between two distinct airports. Instead, we want to know how many different flights we have. Or simply, how many edges we got?

```
scala> graph.numEdges
res7: Long = 5366
```

## 4.3 What are the top N flights from Airport to Airport?

More specifically, we want to find the N busiest air routes. To find the answer, we can simply sort the edges based on their weights (or attributes in GraphX language), and print the first N largest values. To have a human readable output, we add some texts to the output. For now, let's fix N to 10.

```
scala> graph.triplets.sortBy(_.attr, ascending=false).map(triplet => "There are
" + triplet.attr.toString + " flights from " + triplet.srcAttr + " to " +
triplet.dstAttr + ".").take(N)
res8: Array[String] = Array(There are 13788 flights from SFO to LAX., There are
13390 flights from LAX to SFO., There are 12383 flights from OGG to HNL., There
are 12035 flights from LGA to BOS., There are 12029 flights from BOS to LGA.,
There are 12014 flights from HNL to OGG., There are 11773 flights from LAX to
LAS., There are 11729 flights from LAS to LAX., There are 11257 flights from LAX
to SAN., There are 11224 flights from SAN to LAX.)
```

Note that `attr`, `srcAttr` and `destAttr` refer to the edges' values, the source value, and the destination value, respectively. Among these values, only attr needs to be converted to a string. Can you guess how we can find the N less busiest routes?

## 4.4 What Airport is the most popular destination?

This question is a little bit tricky, but nothing that we should worry about! We only need to translate the original question to a graph question. Indeed, here we are interested to find the vertex with the largest number of input edges (recall that our graph is directed). The number of input edges is known as input degree and GraphX has a method to work with that: `inDegrees`.

```
scala> graph.inDegrees.take(10)
res0: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((-1717145497,15),
(1186398190,7), (-1149672775,3), (1290932502,1), (-1072205815,5),
(1599919368,33), (507908134,6), (267125246,2), (-1723027463,1), (1026648244,9))
```

Let's sort the vertices based on their `inDegrees` and take the first one. Note that the output of the above statement (that we want to sort in the next step) is an Array which each of its elements has two values:

airport code and input degree. Make sure that you sort this array based on the second value!

```
scala> graph.inDegrees.sortBy(_._2, ascending=false).take(1)
res6: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((1710100676,173))
```

The `_._2` in the above code indicates that we want to sort the array based on input degrees (the second item).

Is the problem solved? Yes and no! we have the airport code of the most popular destination, but who cares about the code? We need a human readable string (name of the airport). After all, data scientists are human, right?

To solve this issue, we can simply join `inDegrees` and `airportVertices` that we created previously. Remember that `airportVertices` contains both airport codes and name strings.

```
scala> graph.inDegrees.join(airportVertices).sortBy(_._2._1,
ascending=false).take(1).map(_._2._2)
res11: Array[String] = Array(ATL)
```

So, Hartsfield-Jackson Atlanta International Airport (ATL) is the most popular destination in our dataset. Note that `_._2._1` refers to the first value of the second item, because the first item is the airport hash number, and the second one is a tuple of input degree and airport name (e.g., `(1710100676,(173,ATL))`). Now, do you know what `.map(_._2._2)` does at the end of the last example?

Can you change the code to find the least popular destination? What about the most popular origin?

## 4.5 What airport is the most important one?

This question is very vague! There could be a number of factors to indicate the importance of an airport. For example, the input and output degrees in the previous example (i.e., ATL), or even the airports associated with the busiest route (i.e., SFO and LAX). Therefore, the answer is "*it depends!*".

Here, we chose to use PageRank (https://en.wikipedia.org/wiki/PageRank) algorithm and report the airport with the highest rank as the most important one. In short, the vertices which receive more input edges than output edges receive higher ranks. Note that PageRank does not count all input and output edges similarly. For example, receiving an input link from a high-ranked source plays a more important role than receiving an input link from a low-rank vertex.

PageRank algorithm is already implemented and available as a part of GraphX. We only need to set a convergence threshold as the stopping criterion.

```
scala> val stopAt = 0.0001
stopAt: Double = 1.0E-4
scala> val ranks = graph.pageRank(stopAt).vertices
ranks: org.apache.spark.graphx.VertexRDD[Double] = VertexRDDImpl[819] at RDD at
VertexRDD.scala:57

scala> ranks.join(airportVertices).sortBy(_._2._1,
ascending=false).map(_._2._2).take(10)
res3: Array[String] = Array(ATL, DFW, ORD, MSP, SLC, DEN, DTW, IAH, CVG, LAX)
```

In the above example, `_._2._1` and `_._2._2` refer to a vertex's rank and name, respectively.

Can you suggest a small change increase the speed of this example? ***Hint***: Think about reordering some parts of a statement!

# 4.6.4
# A Brief Introduction to Neo4j

## Introduction

Neo4J is a graph database widely used in data science world. According to the Neo4j official website (https://neo4j.com/), it is a highly scalable, native graph database to consider both data and its relationships. It delivers robust and steady real-time performance which is ACID-compliant. In short, Neo4j is an open source, schema-free, NoSQL graph database.

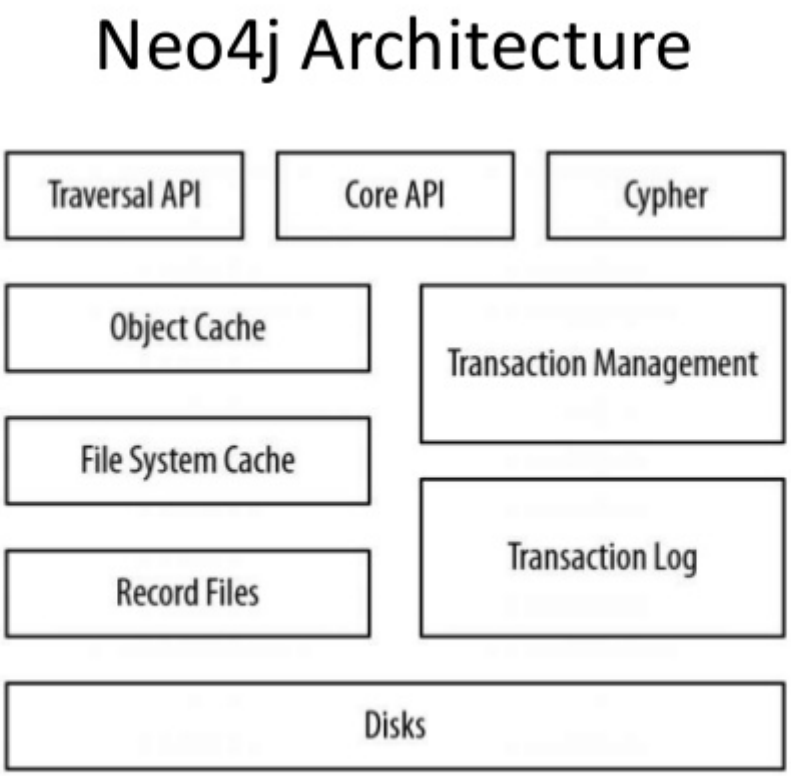
Neo4j (https://neo4j.com/product/#panel-compares-1), as a native graph database, can be compared to the traditional relational databases in many aspects, some of which are:

- **Data storage**
  - Relational DBMS - happens in fixed, pre-defined tables (row, columns). Connected data often disjointed between tables which hinder the query processing efficiency.
  - Graph DMBS - happens in graph structures which improve transaction performance in queries regarding relationships among data.
- **Data modelling**
  - Relational DBMS - models should be developed in logical model and then translated into the physical model. Any change in the data structure seems to be a nightmare!
  - Graph DMBS - data models are flexible! Changes are welcome anytime, anywhere!
- **Query performance**
  - Relational DBMS - low performance in interrelated data items such as deep JSONs.
  - Graph DMBS - given any depth of relationships, graph processing responds to queries in real-time.
- **Query language**
  - Relational DBMS - **SQL**
  - Graph DMBS - **Cypher** which is a native graph query language and is efficient in expressing relationship queries. We will explain Cypher later in this module.
- **Processing at Scale**
  - Relational DBMS - complex data relationships are even costly or impossible!
  - Graph DMBS - no problems with scale as graph databases are inherently scalable.

Note that Neo4j applications can be very different from the Apache Spark GraphX (that is discussed in the previous chapter) and its extension Apache Spark GraphFrame (https://graphframes.github.io/). In brief, GraphX and GraphFrame add high-level abstractions to Spark RDD and DataFrame which make Spark more suitable for processing graph data sets. In contrast, Neo4j, as well as OrientDB (http://orientdb.com/orientdb/), Titan (http://titan.thinkaurelius.com/), and ArangoDB (https://www.arangodb.com/) are graph database management systems that help us store and query the graph data sets.

## Architecture

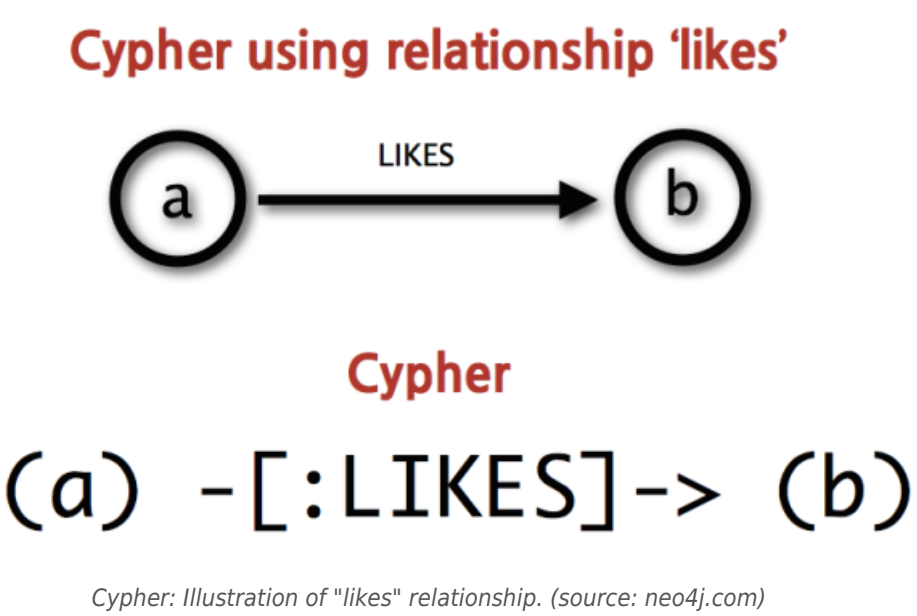
Neo4j comprises of several efficient components. One representation of Neo4j architecture (including its key components) is depicted in the following figure:




*Neo4j components*

## Cypher

Cypher is a declarative, relatively simple yet very powerful graph query language (https://neo4j.com/developer/cypher-query-language/) that allows for expressive and efficient querying and updating of the graph store. Very complicated database queries can easily be expressed through Cypher. It is a powerful language in the case that lets the developer to be concerned with ***what*** they want to do (select, insert, update, delete) without the need to express ***how*** to do it. The following graph simply illustrates the `likes`"relationship query between two nodes a and b:




*Cypher: Illustration of "likes" relationship. (source: neo4j.com)*

Nodes in Cypher are represented as **(node)**. Relationships between two nodes are expressed by arrows: **-->** Additional information for relationships are placed in square brackets. Some examples are:

- relationship-types like **-[:KNOWS|:LIKE]->**
- a variable name before the colon **-[rel:KNOWS]->**
- additional properties **-[{since:2010}]->**
- structural information for paths of variable length **-[:KNOWS*..4]->**

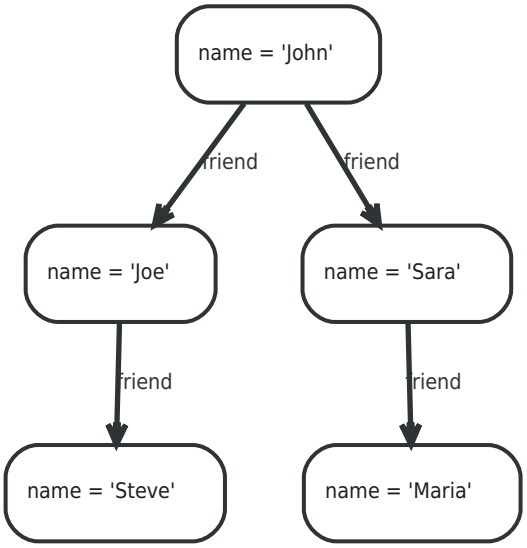
The general syntax of Cypher (https://neo4j.com/developer/cypher-query-language/) queries can be depicted as the following:

```
MATCH (n1:Label1)-[rel:TYPE]->(n2:Label2)
WHERE rel.property > {value}
RETURN rel.property, type(rel)
```

A complete list of operators and syntax clauses is elaborated here (http://neo4j.com/docs/developer-manual/current/cypher/).

## A Simple Example

Let us give one simple example of friendship relation among different people represented by the following graph:




*Neo4j example: friendship relation (source: neo4j.com)*

Given the above graph, a sample query (http://neo4j.com/docs/developer-manual/current/cypher/) to find a user named "John" and his friends-of-friends using the Cypher query language could be written as the following:

```
MATCH (john {name: 'John'})-[:friend]->()-[:friend]->(fof)
RETURN john.name, fof.name
```

And the result is displayed like this:

```
+----------------------+
| john.name | fof.name |
+----------------------+
| "John"    | "Maria"  |
| "John"    | "Steve"  |
+----------------------+
2 rows
```

Now, to make it more exciting, we write a query to retrieve all user names who have followers, names of whom start with "S":

```
MATCH (user)-[:friend]->(follower)
WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve'] AND follower.name
=~ 'S.*'
RETURN user.name, follower.name
```

And the output will be like the following:

```
+--------------------------+
| user.name | follower.name |
+--------------------------+
| "Joe"     | "Steve"       |
| "John"    | "Sara"        |
+--------------------------+
2 rows
```

The complete list of Cypher keywords in Clauses, Sub-clauses, Operators, Functions, Commands, Hints, and Expressions can be found here (http://neo4j.com/docs/developer-manual/current/cypher/keyword-glossary/#cypher-glossary).

## Learn more!

Interested readers can watch this webinar (https://info.neo4j.com/WBCODWBRIntroducingNeo4j3.0_LP-Video.html) about Neo4j 3.0, read this free ebook (https://neo4j.com/graph-databases-book/?ref=home) about the graph database concepts (as long as available) or just take a quick look at this whitepaper (http://info.neo4j.com/rs/773-GON-065/images/Neo4j_Top5_UseCases_Graph%20Databases.pdf) about some applications of graph databases

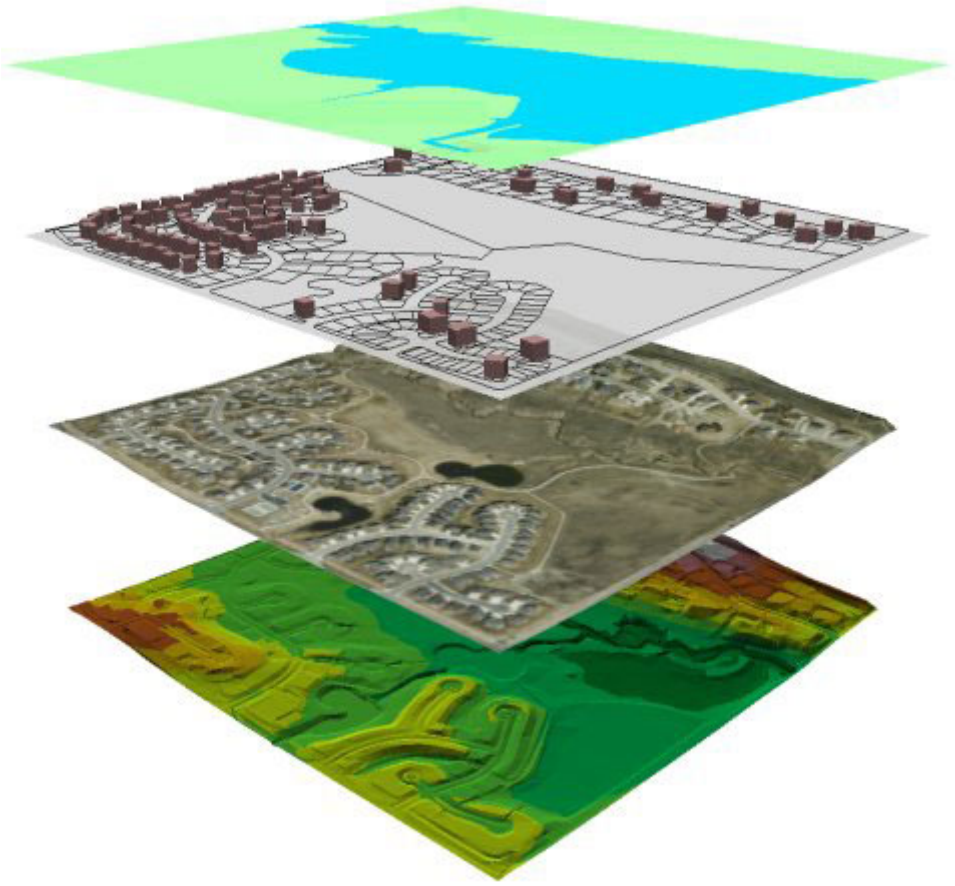
# 4.6.5
# Spatial Databases and Queries

Spatial Databases (http://dna.fernuni-hagen.de/papers/IntroSpatialDBMS.pdf) allow the storage of the geometries of records (such as points, lines, and polygons) inside a Database as well as providing functionality for querying and retrieving the records using these Geometries (https://en.wikipedia.org/wiki/Spatial_database#Geodatabase). The Open Geospatial Consortium (http://www.opengeospatial.org/) as an international non-profit organization is responsible for developing standards regarding the addition of spatial functionality to database systems. Spatial databases support spatial data types in their data models and query languages. They need to address large collections of relatively simple geometric objects. To improve database operations, spatial databases use spatial indices. Some well-known spatial index methods (https://en.wikipedia.org/wiki/Spatial_database#Geodatabase) are HHCode, Grid (spatial index), Z-order (curve), Quadtree, Octree, UB-tree, R-tree (https://en.wikipedia.org/wiki/R-tree), R+ tree, R* tree, Hilbert R-tree, X-tree, KD-tree, m-tree, and Binary space partitioning (BSP-Tree). Spatial objects are partitioned utilizing the minimum bounding rectangle (MBR (https://en.wikipedia.org/wiki/Minimum_bounding_rectangle)) algorithm.

Data entries in spatial databases can be of two types:

- **Spatial Information:** This information is in form of ***locations of objects*** which are individual points in space, and ***space occupied by objects*** such as lines and regions. The spatial information is usually referred to as spatial data types (***SDT***s).
- **Non-Spatial Information:** This information is not spatial but related to spatial objects. City population, postal codes, region names, road names, and speed limits are some examples of non-spatial information.

The following diagram demonstrates different layers of information available in a GIS application.



*Information layers in a GIS example (source: livabilitylane.org)*

Spatial queries (https://en.wikipedia.org/wiki/Spatial_query) are kinds of database queries which are delivered by spatial databases. The main differences between a spatial and non-spatial query are:

- Spatial queries support SDTs and geometry data types like nodes, lines, polygons.
- Spatial queries take into account the interrelationships among SDTs.

There are several systems (https://en.wikipedia.org/wiki/Spatial_database#Geodatabase) developed to support spatial databases. Some prominent examples of geodatabases are Oracle (https://www.oracle.com/database/spatial/index.html) Spatial, IBM DB2 (https://www.ibm.com/analytics/us/en/technology/db2/), SpatiaLite (http://www.gaia-gis.it/gaia-sins/), CouchDB (http://couchdb.apache.org/) with GeoCouch extension, PostgreSQL with PostGIS (http://postgis.net/) extension, Neo4j (https://neo4j.com/product/), AllegroGraph (http://franz.com/agraph/allegrograph/), Apache GeoMesa (http://www.geomesa.org/), and so on. We further discuss the Neo4j in this module to show its capabilities in handling graph-based data types.

Spatial database stores spatial objects (SDTs with coordinates) as well as spatial relationships between the objects. In this case, one can search for an object that spatially intersects another. It is evident that we can store SDTs in a usual database, however, its performance would not be desirable especially for huge datasets. A spatial database is armed with effective facilities (https://geography.wisc.edu/gisdev/index.php/2015/12/15/4-reasons-you-should-use-geospatial-databases/) to efficiently manage and store spatial objects and relationships (http://www.cubrid.org/blog/dev-platform/20-minutes-to-understanding-spatial-database/).

# 5
# Data Streaming and Dynamic Visualization

## Agenda

The main topics that we discuss in this module are related to data streams (e.g., data stream sources, streaming algorithms, and Spark Streaming packages) and Big Data visualization. We will learn key principles, benefits, and limitation in stream processing and compare stream processing with batch processing. Then, we will discuss Spark streaming fundamentals, algorithms, and tools. Finally, we practice dynamic data visualization.

### Learning Objectives

At the end of this module, you should be able to:

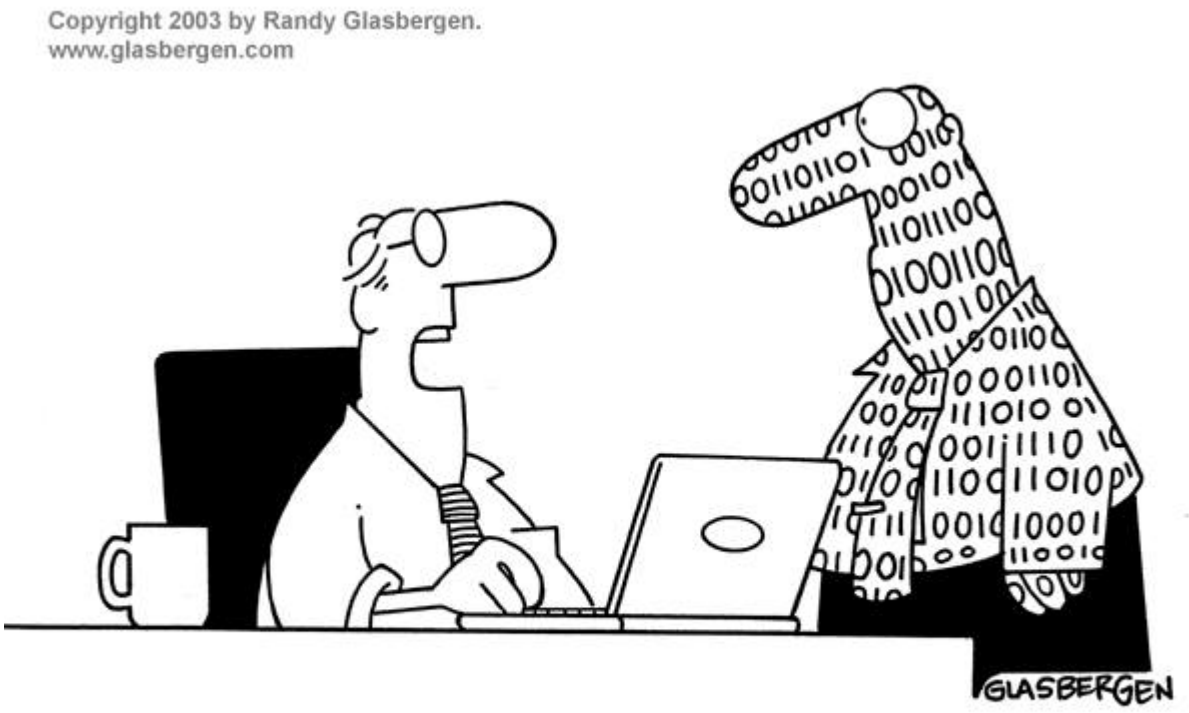
- Demonstrate your understanding of data streaming, dynamic datasets, and commonly used streaming algorithms.
- Demonstrate your understanding of Apache Spark Streaming module components such as streaming context and DStreams.

### References

As usual, the main references for this modules are mainly from the Apache projects documentations such as:

- Apache Spark (2.1) Streaming Programming Guide (https://spark.apache.org/docs/latest/streaming-programming-guide.html)
- Apache Spark (2.1) Streaming + Kafka Integration Guide (https://spark.apache.org/docs/2.1.0/streaming-kafka-integration.html)
- Apache Spark (2.1) Streaming + Flume Integration Guide (https://spark.apache.org/docs/2.1.0/streaming-flume-integration.html)
- Apache Spark (2.1) Streaming + Kinesis Integration (https://spark.apache.org/docs/2.1.0/streaming-kinesis-integration.html)
- Apache Spark (2.1) Clustering - RDD-based API (https://spark.apache.org/docs/2.1.0/mllib-clustering.html)

# 5.1
# An Introduction to Stream Processing




"Watch where you're going, Larry — you walked right through my wireless data stream!"

Data is generated from several sources at unprecedented rates and is flowing everywhere around us. Smart devices, online transactions (including social media activities, credit card purchases), Internet of Things (IoT) devices generated data (such as different sensors, smart home appliances), and many other examples fall into this category (https://aws.amazon.com/streaming-data/). In these applications, it is not feasible to load the arriving data into a traditional DBMS which are not designed to directly support the continuous queries required by these applications. This situation calls for efficient approaches to collect, process, disseminate and update the flowing current of data. The process to address the mentioned requirement is called stream processing (https://en.wikipedia.org/wiki/Stream_processing).

> **Quick Quiz:** Can you name a number of data stream examples and application that you have dealt with? What are the challenges of storing, querying, and processing such class of dynamic data?

## Data Streams

Prior to discussing on data streaming systems, it is critical to understand different **real-time systems**'

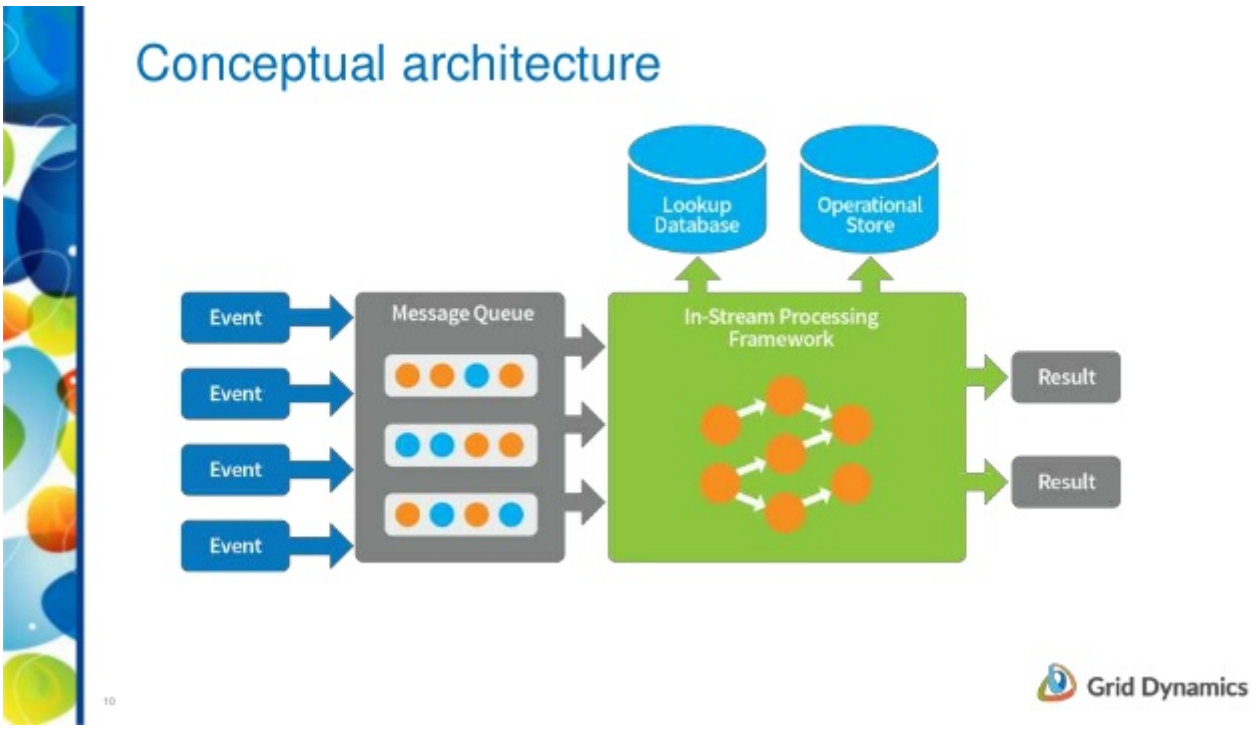
categories. According to Streaming Data book by Manning (https://www.manning.com/books/streaming-data), there are three main classes of real-time systems:

- **Hard Real-time Systems** - such as heart pacemaker or anti-lock brakes. Hard real-time systems produce data in microseconds and need to be processed within milliseconds. They do not tolerate any delay as any processing lag may cause total system failure or even worse which may cause potential life loss. Hard real-time systems can be identified in most of the embedded systems with strict time requirements.
- **Soft Real-time Systems** - like the airline reservation systems, voice over IP (VoIP) services in multimedia applications such as Skype and Viber, and online stock quotes. The data is produced within milliseconds in such systems and needs to be processed in seconds. Their tolerance to delays is low.
- **Near Real-time Systems** - such as online video streaming which produces data in seconds and needs to be processed within a minute. As any delay in such systems is not of great concern, their tolerance is allegedly high regarding any processing lag.

Given different real-time system categories, a ***streaming data system*** is defined as a non-hard real-time system with clients that consume data whenever they need it.

## A Conceptual Architecture

A sample conceptual streaming data architecture is illustrated in the following figure where streaming data elements are fed into the architecture from the *event* nodes and are being transferred to the *stream processing framework* through the reliable and efficient *message queue* component. The stream processing framework performs several computations on the streaming data utilizing robust *data store* components for storage and retrieval of the results. The deliverable output will be disseminated to the target applications/users through the *result* nodes.




*Conceptual Streaming Data Architecture*

## Applications

There are several application scenarios for data streaming systems. To name a few:

- **Healthcare** - monitoring patients' vital health status continuously and proactively identify at-risk patients.
- **Transportation** - dynamic rerouting of traffic or vehicle fleet.
- **Retail** - real-time in-store advertisements, recommendations, dynamic inventory management.
- **Security and Surveillance** - identifying suspicious behaviors and potential threats and making real-time interventions.
- **Digital Marketing and Advertising** - personalizing and optimizing contents based on real-time information.
- **Manufacturing** - performing proactive maintenance, dynamic identification of equipment failures and instant reactions when required.
- **Credit Related Applications** - identifying suspicious financial behaviors, detecting fraudulent transactions soon after they occur.

# Big Data Stream Ecosystem

To address the increasing demand for real-time processing and data streaming, multiple stream processing systems have been proposed. Those systems mostly comprise of the following five components (http://ingest.tips/2015/06/24/real-time-analytics-with-kafka-and-spark-streaming/):

- **Data Sources** - the raw data streams which are being generated from diverse sources. Examples are sensors, smartphone apps, the web and online interactions, server logs. Data types produced from these sources are typically *semi-structured* and *schema-less*. Datasets of these types are commonly referred to as ***Dynamic Datasets***.
- **Message Bus** - an efficient messaging system to transfer the generated streaming data to proper analytical units. This system should be reliable, robust, high-throughput and should have a low latency. Some examples are Apache Kafka (https://kafka.apache.org/) and Apache Flume (https://flume.apache.org/) which will be elaborated later in this module.
- **Stream Processing System** - the main computational engine for the streaming data. The steam processing system is an analytical framework that is capable of performing different computations on streaming data. Apache Spark Streaming (https://spark.apache.org/streaming/) is one example.
- **NoSQL Data Storage Components** - given that data types are mostly semi-structured and schema-less (mentioned in the **data sources** above), a non-relational DBMS that is able to store/retrieve a huge amount of streaming data. Apache HBase (http://hbase.apache.org/), Apache Cassandra (http://cassandra.apache.org/), and MongoDB (https://www.mongodb.org/) are among such popular NoSQL DBMS.
- **Target Applications** - the end applications which are the final users of the processed streaming data.

The following figure depicts the afore-mentioned components in one canonical stream processing architecture:

*Data Stream Processing Ecosystem (Source: Cloudera)*

# Potentials and Pitfalls

Like any other technology, stream processing has its own advantages and disadvantages. In this part, we discuss some of the benefits as well as challenges of utilizing big data stream processing technologies. Then, we compare it with traditional relational databases and conventional batch processing techniques.

## Benefits of Streaming Data

Some of the main benefits of analyzing streaming data are listed as the following (https://aws.amazon.com/streaming-data/):

- **Instant detection of issues within the organization** - real-time error identification helps enterprises intervene and react to them to reduce their impact. This way, organizations can prevent from being failed by responding to errors effectively and soon.
- **Identifying state-of-the-art strategies** - utilizing proper streaming processing analytics makes organizations remain competitive and in some cases, one step ahead of their opponents.
- **Dynamic service improvements** - monitoring and processing dynamic customers' behaviors (such as data gathered from real-time sensors) gives the power of reacting instantly to customers' requests and increase revenue in exchange.
- **Detection of suspicious behaviors instantly** - frauds in financial transactions or attempts to hack into the systems can be notified the moment they happen by analyzing the generated streaming data.
- **Reduced costs** - streaming processing provides the pay-as-you-go scheme for organizations which frees heavy costs of leasing cloud infrastructures, software, platforms, and services. Also, businesses get rid of new hardware installations and purchasing new software packages for their big streaming data processing tasks with the help of quick and efficient services provided by stream processing systems.
- **Scalability** - provisioning of new storage, memory, and processing capacity for organizations' growth can influence their budget and competition dramatically. Stream processing systems are capable of providing a scalable environment for businesses with high performance, capacity and dynamism to accommodate their growth in nearly unlimited amounts. This way, organizations are free to produce and analyze big data and customer demand as possible without being worried about their ability to process the tasks.

# Challenges of Streaming Data

Key challenges of stream processing can be listed as the following (https://aws.amazon.com/streaming-data/):

- **Proper coordination between the storage and processing components** - storage unit in stream processing systems should provide strong consistency for storage/retrieval of streaming data in its reads/writes which are being used by the processing unit. The processing unit will use the data provided by the storage unit, perform requested computations on it and generates the results to be disseminated to identified targets. Furthermore, the processing unit needs to notify the storage unit to remove unwanted data elements. This reciprocal relationship between the storage and processing units calls for an accurate and efficient communication means.
- **Security issues** - stream processing allow organizations to analyze internal and external threats affecting their business. There is a risk of identifying sensitive information which is not adequately protected.
- **Real-time insights need a different way of working with companies** - receiving the generated insights out of the streaming data in seconds is way different than getting them once a

week! Therefore, we need to come up with new approaches to working because insights require actions to be taken. This makes companies more of an information-centric business.

## Data Streaming vs Relational Databases

Main differences between data streams and the conventional relational data storage models are listed in the following:

- In data stream, data elements emerge online with high speed.
- The underlying system does not have any control on the sequence(s) of arriving data elements, within a data stream, or even across data streams to be processed.
- The volume of data streams could be unlimited.
- The moment the data element processing finishes, the system gets rid of that element (sometimes it is archived) which makes its retrieval almost impossible. In some cases, particular data elements are stored in memory which is a small fraction of the whole arriving data streams.

## Stream Processing vs. Batch Processing

Batch processing (https://en.wikipedia.org/wiki/Batch_processing) refers to the execution of a sequence of tasks in one application/system in a non-interactive way (offline). In batch processing, the input is a group (batch) of jobs instead of just one in regular processing. In such batch systems, the output is produced based on all the data which was given to the system as input (batch input). Having all the data elements, batch systems can perform deep analytical processing. Examples are MapReduce-oriented systems such as Amazon EMR. We discussed MapReduce programming paradigm in the previous chapters.

Stream processing, on the other hand, just ingests a sequence of data and updates its statistics and reports, incrementally, on getting new data elements (data streams). The following table depicts the major differences between batch and stream processing:

| | **Batch processing** | **Stream processing** |
|---|---|---|
| **Data scope** | Queries or processing over all or most of the data in the dataset. | Queries or processing over data within a rolling time window, or on just the most recent data record. |
| **Data size** | Large batches of data. | Individual records or micro batches consisting of a few records. |
| **Performance** | Latencies in minutes to hours. | Requires latency in the order of seconds or milliseconds. |
| **Analyses** | Complex analytics. | Simple response functions, aggregates, and rolling metrics. |

# 5.2
# Streaming Algorithms

## Introduction

To process the data streams generated from diverse sources, several streaming algorithms (https://en.wikipedia.org/wiki/Streaming_algorithm) have been proposed. Sequences of data streams are fed into streaming algorithms with few processing and memory units. Therefore, the output will be an approximation response based on the loaded limited data stream in the memory - we call that portion of data ***sketch*** of data streams. Given the mentioned limitations, streaming algorithms provide estimated results based on the portion of streaming data (sketch) loaded into the designated memory. Moreover, the performance of such algorithms is measured by taking into consideration the following factors (https://en.wikipedia.org/wiki/Streaming_algorithm):

- the number of passes the algorithm makes through the streaming data,
- the amount of allocated memory for processing the sketch of the data stream,
- the time complexity of the algorithm itself.

Streaming algorithms should take these important notes into consideration:

- the streaming data should be read just once,
- a small portion of the data stream (sketch) will be preserved and the system should discard the rest of the stream data,
- they should process the requested queries utilizing the sketch of stream data, approximately.

The following table summarizes the key differences between traditional and stream data processing (https://www.researchgate.net/publication/236007656_Data_Stream_Processing):

| | **Traditional Data Processing** | **Stream Data Processing** |
|---|---|---|
| **Number of Passes** | Multiple | Single |
| **Processing Time** | Unlimited | Restricted |
| **Memory Usage** | Unlimited | Restricted |
| **Type of Result** | Accurate | Approximate |
| **Distributed** | No | Yes |

## Algorithms

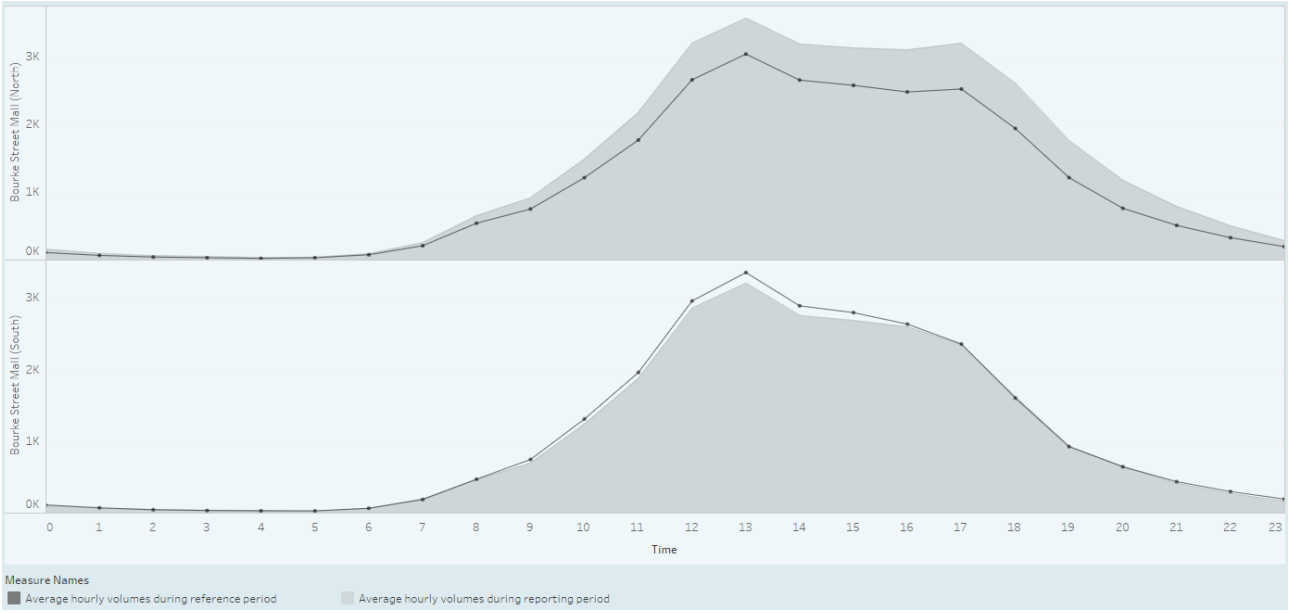
Suppose each data stream as a long sequence of elements arriving rapidly which are represented as $I_1, I_2, \ldots, I_t, \ldots, I_m$ where $I_t$ is the $t^{\text{th}}$ element and the size of the data stream is denoted as $m$.

It is important to understand different **data stream models** before proceeding with the streaming algorithms. According to the literature, there are three key data stream models mentioned in the following - given the aforementioned sequence of data streams with length $m$, the $I_t$ element can be represented in three different ways:

## 1. Time Series Data Stream Models

These models are best adapted to model the time-series data such as monitoring network nodes' traffic in every given time (from seconds to minutes), changes in the exchange market trades every minute, and so forth. Time Series are sequences of measurements that follow non-random orders. In the Time Series models, $I_t$ is modeled as $a_t$ where $a_t \in \{a_1, a_2, \ldots, a_n\}$. This way, the data stream is represented as a sequence of elements where each item belongs to $\{a_1, a_2, \ldots, a_n\}$. Therefore, the domain is limited to $n$ possible different values.

For example, the following figure depicted from the City of Melbourne website (https://data.melbourne.vic.gov.au/Transport-Movement/Pedestrian-volume-updated-monthly-/b2ak-trbp) illustrates the Melbourne Pedestrian volume for a given day (24 hours time span) for the Bourke Street Mall (north and south):




*City of Melbourne Pedestrian volume - 2016*

As one can realize, the data ranges from 0 to 3k people for each given time.

## 2. Cash Register Data Stream Models

Cash Register models are the most popular models of data streams. They are applicable in a diverse range of systems such as observing a sequence of IP addresses that access a web server. In the Cash Register models, the $I_t$ element is represented as a pair of $(j, c_t), a_i(t)$ and is calculated as the following given that $\langle a_1(t), a_2(t), \ldots, a_n(t) \rangle$ is the state of the data streams at time $t$:

$$a_i(t) = \begin{cases} a_i(t-1) + c_t & \text{if } i = j \\ a_i(t-1) & \text{otherwise} \end{cases}$$

Where $c_t \geq 1$.

In the cash register model, the items that arrive over time are domain values in no particular order, and

the function is represented by implicitly aggregating the number of items with a particular domain value. For example, in the telephone calls case, the stream could be:

```
(8008001111, 10), (8008002222, 15), (8008003333, 13), (8008001111, 23),
(8008001111, 3), ...
```

The underlying signal, namely `(8008001111, 36), (8008002222, 15), (8008003333, 13)` has to be constructed by aggregating the total number of minutes outgoing from numbers `8008001111, 8008002222, 8008003333` etc.

## 3. Turnstile Data Stream Models

These models are similar to Cash Register models where $|c_t| \geq 1$ that accepts both negative and positive values for $c_t$. Turnstile models are useful in fully dynamic situations where we have both insertions and deletions such as monitoring number of people passing through train stations' turnstiles.

To process the streaming data elements, several methods have been proposed the most well-known of which can be categorized (http://users.monash.edu/~mgaber/Muthu-Survey.pdf) in the following list (http://drops.dagstuhl.de/opus/volltexte/2013/4296/pdf/ch09-ikonomovska.pdf): sampling, sketching, and distinct sketching.

## Sampling Techniques

Sampling is a general technique to tackle huge amounts of data streams. It refers to the process of selecting a subset of data items to be processed among huge amounts of streaming data. Put simply, it is the process of selecting a smaller number of items from a larger group according to some rules (e.g., randomized or deterministic methods).

Sampling is basically a statistical probabilistic technique. One issue with Sampling in data streams is that the size of incoming data is unknown. Another issue with Sampling is that it cannot handle data stream rate fluctuations. In general, Sampling approaches lose information because they just select a subset of arriving tuples while skipping the rest. One example could be computing the median packet size of some selected IP packets where just a fraction of packets is selected and the median of the selected packets is calculated which will be an estimate for the real median. Several Sampling techniques have been proposed to answer queries over the streaming data. Three of the most well-known Sampling techniques are listed in the following:

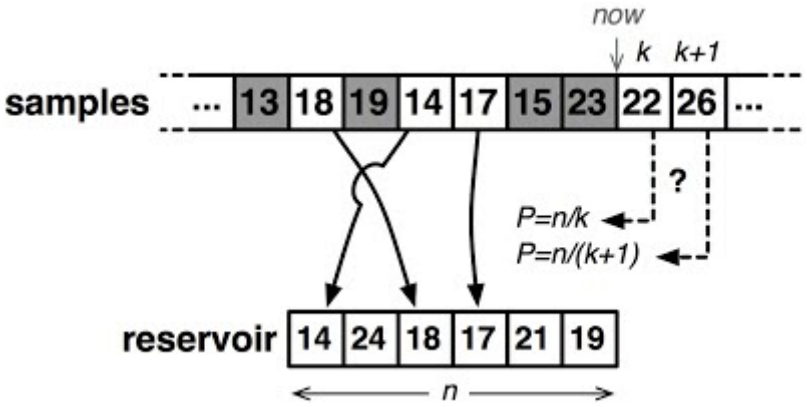
### 1. Reservoir Sampling

This technique is the most popular sampling scheme and maintains online random samples and is a classic Sampling algorithm. The idea is to preserve a sample of size $m$ (called the **reservoir**), data items of which can be replaced by new data elements with a certain probability. In Reservoir Sampling, a single data item is randomly selected as sample item from the streaming data without having the knowledge of its size. The selection process is performed uniformly. The algorithm has a simple process:

1. select the first $m$ data items (from the total streaming data) as samples,
2. choose to sample $t^{\text{th}}$ item with the uniform probability of $m/t$, and
3. if sampled, randomly replace a previously sampled data item with any data item in the sample with the same probability (uniformly).

The big challenge of the Reservoir Sampling algorithm is that it is very hard to parallelize. However, this

technique is useful in applications where observing the whole streaming data is expensive or impossible. One sample illustration of the Reservoir Sampling algorithm is depicted in the following figure:




*Reservoir Sampling (credit: Kapildalwani)*

## 2. AMS Sampling

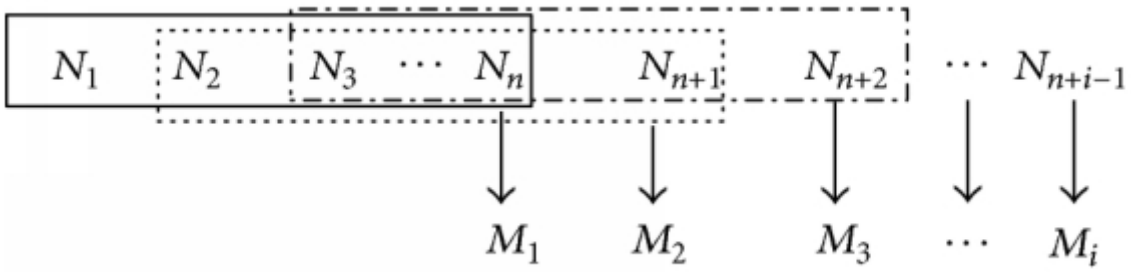
AMS sampling is an algorithm to estimate frequency moments using just sublinear space with the desired accuracy. It is a sample and count approach where a sample is preserved with additional data. The main idea is to pick a data item uniformly at random from the set of all streaming data items and compute their frequencies inside the sample. Then, more complicated analytical processes can be performed on the sample data such as average, min/max values, median and other statistical quantities along with most frequent data items and represent the results in histograms.

> **Quick Quiz:** What do you expect to happen if the domain (the possible unique values) of the data being sampled is very wide? How can we deal with such phenomenon? What would be the caveat?

## 3. Sliding Window Sampling

This method is useful in applications where, unlike other two above-mentioned methods, recent data items are more significant than the previous ones. The sliding window approach then predicts the future behavior of the system based on its current status where the recent data items become important. One example application could be random early detection protocol utilized by Internet routers to extrapolate the traffic limitations by preserving recent data items' statistics. It is applicable in situations where we need to give preference to recent data.

The main idea is to define a stream's location such that all stream data items greater than that location are assigned uniform (equal) significance, while data items less than the defined location are completely skipped! If we want to extend our sliding window size to $w$, we will increase the defined location on each incoming data item until we collect $w$ recent data items. In other words, if we are interested in the recent $w$ items and the streaming data items are represented as a sequence of $a_1, a_2, \ldots, a_t$ where $a_t$ is the most recent data item arrived, we need to just observe data items $a_{t-w+1}, a_{t-w+2}, \ldots, a_t$. A very simple representation of this algorithm is depicted in the following figure:



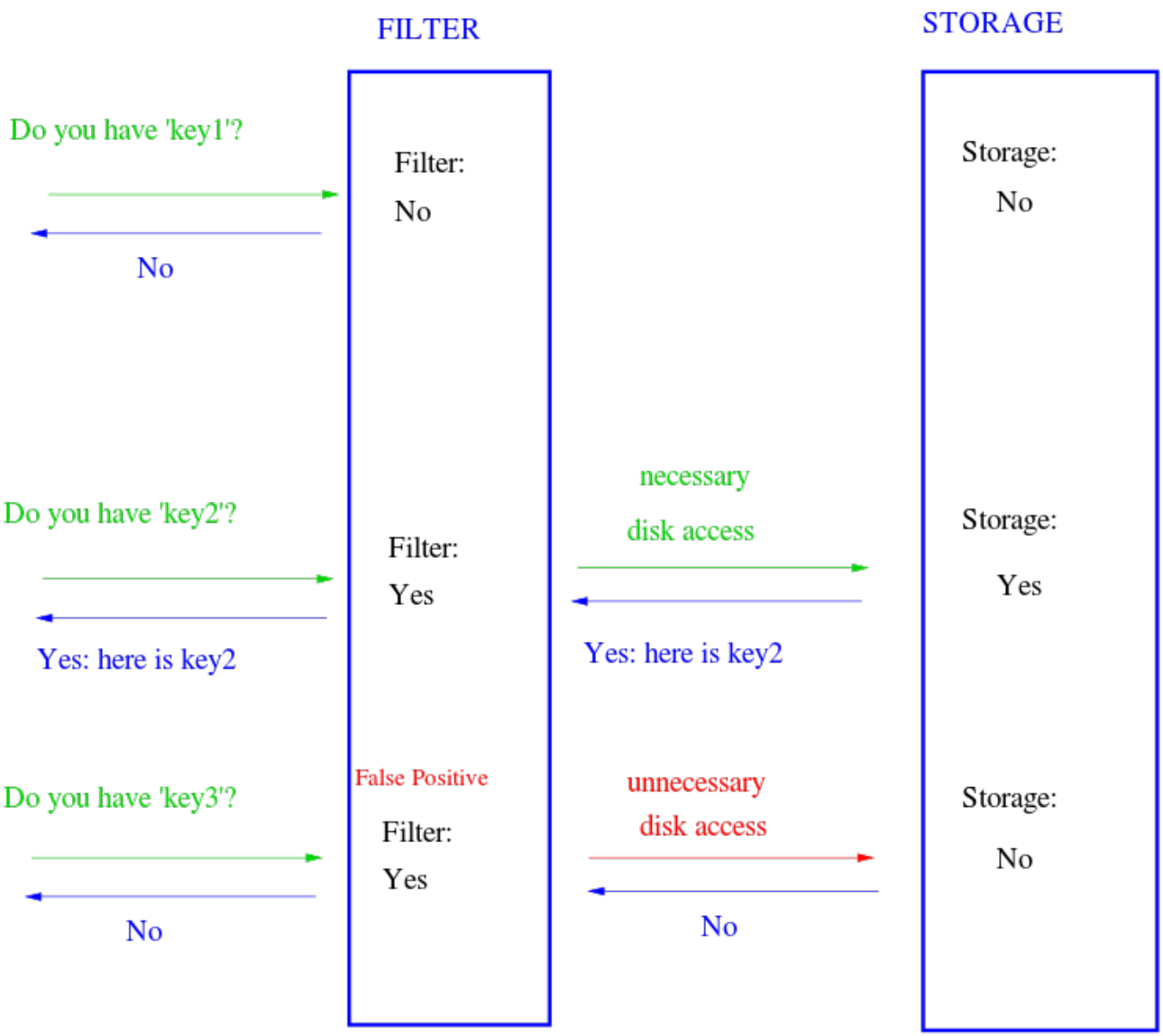

*Sliding Window Sampling (source: doi.org/10.1155/2014/831839)*

## Sketching Techniques

Sketching is the process of projecting a subset of the features randomly which translates into a vertical sampling of arriving data streams. It is most useful in comparing several data streams as well as responding to aggregate queries. Put simply, it provides a space efficient summary of the input data streams. Sketching technique suffers from one big challenge: accuracy which hinders its usage in data stream mining. In Sketching techniques, a linear projection will be applied on-the-fly over a high dimensional streaming data to reduce its dimension to a smaller extent. Therefore, Sketching is an approach used for ***dimensionality reduction***. A list of prominent Sketching algorithms over the streaming data is noted in the following:

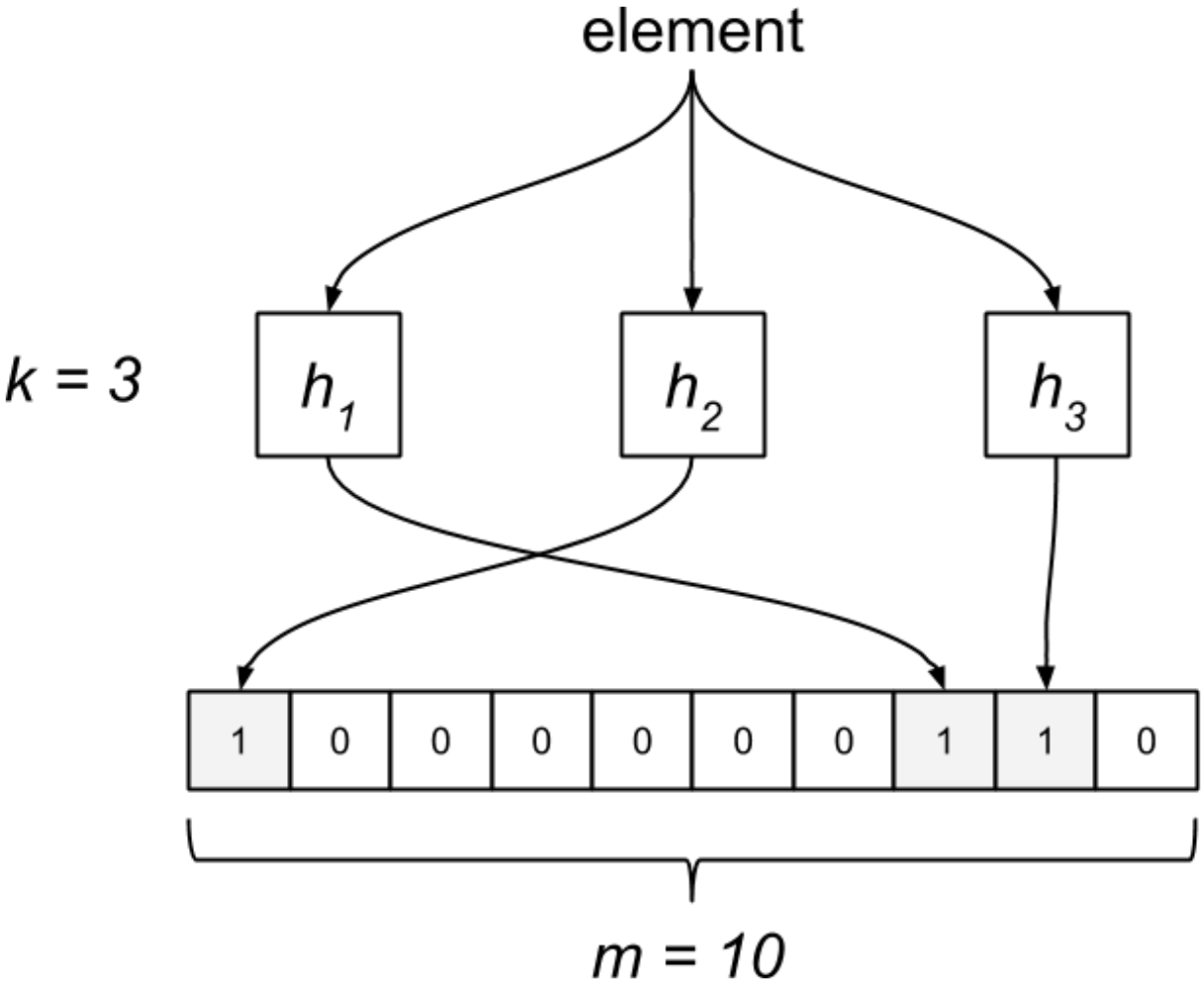
### 1. Bloom Filtering

This method is a common and simple probabilistic data structure by answering the question: *is the newly arrived data item a member of a set?* Bloom filtering method does not store actual data items. Instead, it preserves their memberships. In general, this approach efficiently responses whether a recent data item is in a large set of data items by utilizing specific hash functions. It is a probabilistic approach that answers questions about data items' membership to a certain set. For example, it can respond to queries regarding certain databases. On arrival of a query for a piece of data, if the filter does not have the data, the database is completely bypassed, but if it has the data, the query will be sent to the storage to retrieve the requested element. The following figure illustrates a simple scenario in different queries with Filter and Storage components. Please note that the data items use the hash function to calculate their keys and then ask for the data.




*Bloom Filtering (source: bravenewgeek.com)*

The Bloom Filtering algorithm comprises a bit array of size $m$ and $k$ different hash functions to map the incoming streaming data into their corresponding cell in the bit array. By default, all bits in the bit array is set to `0` (unset). On arrival of each data item, it goes through all hash functions to produce $k$ different indices in the bit array. Those indices are set to 1. Now, to check each data item's membership, we apply the hash functions on it and check whether all its corresponding bits in the bit array are set to 1. One simple membership scenario of the bit array of size `10` with 3 different hash functions is represented in the following figure:




*Bloom Filtering Example (source: bravenewgeek.com)*

## 2. Count-Min Sketch

Count-Min (CM) Sketching is another data stream summarization technique. It provides a frequency table of data items which can be accessed through a hash function. The hash function maps each data item to its corresponding frequency using only sublinear space. The CM sketching method is similar to the Bloom filtering method in their data structures, but it has a sublinear number of table cells based on the sketch quality approximation, where the Bloom filtering just calculates each data item's frequency in a given set (their membership).

## 3. Universal Hash Functions

This method is a generalization in hash functions used in the standard sketch algorithms (http://drops.dagstuhl.de/opus/volltexte/2013/4296/pdf/ch09-ikonomovska.pdf). We normally utilize pseudo-random hash functions in sketches because of the memory limitation. However, if we were to devise a completely random hash function, we needed to store any function over the universe. The universal hash functions are suitable pseudo-random hash functions to produce random-like functions with smaller memory.

## Distinct Elements Sketch

This is a problem which is defined as: given a set of data streams $S$ each of which is from the domain $D$ (in a simple view, they can be integers in the range $[0 - D]$), calculate the number of distinct elements in S (http://www.cse.cuhk.edu.hk/~taoyf/course/wst501/notes/lec4.pdf). For example, if we have data streams of type integer and the set *S* includes data items $\{2, 55, 10, 2, 3, 10, 11, 55, 10\}$, then the response will be $5$ which means we have $5$ distinct data items in arriving set $S$. One sample application could be finding the number of distinct users who logged in to a particular web-site given a stream of logins to that web-site. The **Flajolet-Martin** (FM) algorithm is proposed to estimate the number of such distinct elements (https://en.wikipedia.org/wiki/Flajolet%E2%80%93Martin_algorithm) with a probabilistically guaranteed accuracy.

### 1. FM Sketch

The FM Sketch algorithm observes each data item in an arriving data stream set just once and produces the approximate answer on the number of distinct elements with reasonable precision (https://en.wikipedia.org/wiki/Flajolet%E2%80%93Martin_algorithm). Assume that we have a hash function $h(w)$ that maps each data item to integers in the range of $[0, 2^w - 1]$. Then, the algorithm works in three simple steps:

1. for each data item $w$ in the set of arriving data stream $S$, calculate $h(w)$,
2. find the maximum among the computed hash functions and store it in a variable ($K$ for instance), and
3. return the number of distinct data items as $2^k$.

The advantage of the FM Sketch algorithm is its ability to approximate distinct elements with much smaller storage which is logarithmic to the number of distinct data items.

# 5.2.1
# Activity 1: Stream Sampling

The objective of this activity is to implement the Reservoir Sampling algorithm in Scala. Then, we use this algorithm to analyze a publically available Wikipedia dataset.

## 1. Reservoir Sampling

### 1.1 Simulate a stream

In the first part, we intend to implement a simple reservoir sampling. To do so, we need to simulate a streaming source. It's simple! Let's assume we have a stream of size `N` and implement the stream as a Scala `Array`.

```
scala> // Create an Array of 1 to 100 Int
scala> var stream = Array.range(1, 101)
stream: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99, 100)
scala> // Find and store the number of values in the stream
scala> val N = stream.length
N: Int = 100
```

### 1.2 Create the reservoir memory

Now, let's create a reservoir of length n. The length of the reservoir should be manageable. For example, `n=10`

```
scala> // Set the length of the reservoir
scala> val n = 10
n: Int = 10
scala> // Book some space for the receiving stream
scala> val reservoir = new Array[Int](n)
reservoir: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

### 1.3 Perform sampling

Before proceeding any further, let's create a random number generator. We need it in the next steps.

```
scala> // Create a random number generator
scala> val myrandnum = scala.util.Random
myrandnum: util.Random.type = scala.util.Random$@21be80f6
scala> // Set a random seed (can be any number)
scala> myrandnum.setSeed(5202)
```

In the above example, we set our random seed to `5202.` You can choose any other number. Also, one can pick a dynamic number such as `System.currentTimeMillis/1000.` Obviously, tracing a program with dynamic initial seed is problematic unless we record the seed to be able to rerun the experiment with exact same initial seed.

As you recall, reservoir sampling records all receiving stream as long as its capacity allows. Therefore, for the first n values, we can store them one after another:

```
// Store the first k numbers
for (k <- 0 until n) reservoir(k) = stream(k)
```

When we run out of space (from n+1 receiving value onward), we should choose a random number `t` in range 1 to k where k in the number of received values so far. If `t` is a valid index (`t<n`), then we remove the $t^{th}$ element from the reservoir and store the $k^{th}$value in its spot:

```
for (k <- n until N) {
    // Pick a random index.
    val t = myrandnum.nextInt(k)
    // replace t-th element if t is a valid index
    if (t < n){reservoir(t) = stream(k)}
}
```

Let's put both parts together in one `for` loop block.

```
scala> for (k <- 0 until N) {
     |     // Store the first k numbers
     |     if (k < n) {
     |         reservoir(k) = stream(k)
     |     }
     |     else {
     |         // Pick a random index.
     |         val t = myrandnum.nextInt(k)
     |         // replace t-th element if t is a valid index
     |         if (t < n) {
     |             reservoir(t) = stream(k)
     |         }
     |     }
     | }
```

## 1.4 Print the output

And now we can print te results of our simulation:

```
scala> println("These "+n.toString+" values: ")
These 10 values:
scala> println(reservoir.mkString(" "))
1 81 3 99 77 27 64 70 9 53
scala> println("are randomly selected from these "+N.toString+" values: ")
are randomly selected from these 100 values:
scala> println(stream.mkString(" "))
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
```

```
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

To have a better formatting, you can create a Scala file, combine all the above codes and run them as a Scala script.

# 2. Read a Compressed File

The Wikipedia file that we want to use as the source stream is compressed. In this part, we want to write a basic Scala program to read such file. Then, in the next step, we will combine this code with the sampling algorithm that we developed in the previous example. The result would be a complete Scala script that samples some titles from a compressed file. For now, let's write the file reading script:

```
//Import packages
import scala.io.Source
import java.io._
import java.util.zip._
import scala.collection.mutable.ArrayBuffer
// Define a function that converts GZIP to a stram
def gis(s: String) = new GZIPInputStream(new FileInputStream(s))
// Define an Array of titles (String)
var sample_titles = ArrayBuffer[String]()
// Define an Array of indices (Int)
var sample_indices = ArrayBuffer[Int]()
// Add all titles and indices to the arrays
for ((line, index) <- Source.fromInputStream(
    gis(args(0))).getLines().zipWithIndex) {
    sample_titles += line
    sample_indices += index
}
// Print all data
for(i <- sample_titles.indices)
    println(s"sample_titles("+i+") -> ["+sample_indices(i)+"]"+sample_titles(i))
```

Assume we save the above script in a file called `ReadZipWiki.scala` and our input file is `test.txt.gz` (you can find this file in `/Documents/scala-reservoir-sampling/Reservoir-sampling-wikipedia`). To run it, we should go to the command-line (i.e., bash) and issue the following:

```
scala ReadZipWiki.scala test.txt.gz
```

In the above example, the name of the source file (`test.txt.gz` in this case) is passed to `ReadZipWiki.scala` as `args(0)`. Therefore, if we issue the above command with a different filename, the value of `args(0)` would be different.

When you run this script, you will observe that `test.txt.gz` has only two records. Now, you can rerun the same script with a larger file. For example, `enwiki-latest-all-titles-in-ns0.gz` that is accessible in the same folder. What do you observe?

As you can see, the actual file is too large for our script to process it completely. One can pass another argument (i.e., `args(1)`) to the script to only read and print the first n items (***Hint:*** *Think about using a break statement!*). However, this limitation brings us to think about the proper solution, which is nothing but sampling!

# 3. Complete Solution

Now, we can combine the above sections to develop a complete solution that reads the first n items of the file and samples the other items based on the reservoir sampling algorithm. Try to do it yourself before referring to the final solution that is provided bellow:

```
//Import packages
import scala.io.Source
import java.io._
import java.util.zip._
import scala.collection.mutable.ArrayBuffer
val myrandnum = scala.util.Random
myrandnum.setSeed(5202)
// Define a function that converts GZIP to a stram
def gis(s: String) = new GZIPInputStream(new FileInputStream(s))
val n = args(1).toInt
// Define an Array of titles (String)
var sample_titles = ArrayBuffer[String]()
// Define an Array of indices (Int)
var sample_indices = ArrayBuffer[Int]()
// Add all titles and indices to the arrays
for ((line, index) <- Source.fromInputStream(
    gis(args(0))).getLines().zipWithIndex) {
    if (index < n) {
        sample_titles += line
        sample_indices += index
    } else {
        val t = myrandnum.nextInt(index)
        if (t < n) {
            sample_titles(t) = line
            sample_indices(t) = index
        }
    }
}
// Print all data
for(i <- sample_titles.indices)
    println(s"sample_titles("+i+") -> ["+sample_indices(i)+"]"+sample_titles(i))
```

Assuming the above script is stored in a file called `WikiReservoir.scala`, you can apply it on the large Wikipedia file as:

```
scala WikiReservoir.scala en  enwiki-latest-all-titles-in-ns0.gz 10
```

And the output would be something like:

```
sample_titles(0) -> [8713096]Open_Web_Interface_for_.NET
sample_titles(1) -> [11502792]The_Man_They_Could_Not_Hang_(book)
sample_titles(2) -> [9173034]Pieris_tithoreides
sample_titles(3) -> [8622367]Oder_Dam
sample_titles(4) -> [1549860]Barchaniella_sacara
sample_titles(5) -> [3523112]Dracula_(1931_film)
sample_titles(6) ->
```

```
[8278027]National_Register_of_Historic_Places_in_West_Side_Chicago
sample_titles(7) -> [6948565]List_of_Konoha_Genin
sample_titles(8) -> [257547]2003-04_NCAA_football_bowl_games
sample_titles(9) -> [2085615]Buffer_Overflow
```

Note that `10` in the last example denotes the capacity of our reservoir. Since our script expects the second argument (i.e., `args(1)`), missing it will result in a crashed program. Can you fix this problem by adding a default value to be used when the second argument is missing?

# 5.3
# Spark Streaming Fundamentals

## Introduction

Spark Streaming is an extension of Apache Spark's API for stream processing which provides fault tolerant, scalable and high throughput stream calculations (https://spark.apache.org/streaming/). It extends the Spark's batch infrastructure to address the need for real-time analysis of streaming data. Spark Streaming simply **receives** streaming data from diverse data sources such as sensors, IoT devices, and live logs, **processes** them in parallel, and generates the results (**output**) to target systems e.g. HBase, Kafka, and Cassandra. A high-level stream processing architecture of Spark Streaming is illustrated in the following figure:


*Spark Streaming*

According to the figure, Spark Streaming simply breaks down the arriving data streams into several batches efficiently and treats each batch as RDDs which can be processed with RDD operations. Finally, the computed results are disseminated to target databases, filesystems, or live dashboards in batches.

Spark Streaming supports a high-level abstraction of a continuous stream of data which is called **Discretised Streams** or **DStreams**, for short. DStreams as a sequence of RDDs can be created from the received sources such as Kafka, Flume, and Kinesis, or from the processed data streams generated by transforming the input stream which means from applied operations on other DStreams (https://spark.apache.org/docs/latest/streaming-programming-guide.html). DStreams support the same operations applicable in RDDs' transformation and actions such as **map, flatMap, filter, count, reduce, countByValue, reduceByKey, join, updateStateByKey**. Next, we review some basic concepts related to Spark Streaming.

## StreamingContext

StreamingContext (https://spark.apache.org/docs/latest/streaming-programming-guide.html#initializing-streamingcontext) is the main entry point for all streaming functionality. Creating a StreamingContext object instance is the first

step in starting any Spark Streaming program. The following code snippet demonstrates how to create and initialize one StreamingContext object in Scala programming language. In the following code, appName is the name of the application to be displayed in the cluster user interface, the master is the Spark or YARN cluster URL, and one local instance of a StreamingContext object is created which has the batch interval of 1 second.

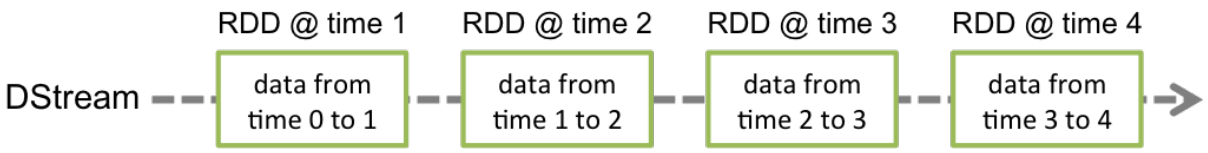
```
import org.apache.spark._
import org.apache.spark.streaming._
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

After creating a StreamingContext instance, we need to create input DStreams to start receiving input from different sources (by calling `streamingContext.start()`) and define particular computations using streaming transformation/output operations over the created DStreams and wait for the process to be finished.

> **Note 1:** once you started the context new streaming operations cannot be defined/added.
> **Note 2:** once a context has been stopped, it cannot be re-started.
>
> For more on these, please refer to the Spark Streaming documentation. (https://spark.apache.org/docs/latest/streaming-programming-guide.html#points-to-remember)

## DStreams

As mentioned earlier, Spark Streaming supports a fundamental data abstraction named Discretized Streams (DStreams (https://spark.apache.org/docs/latest/streaming-programming-guide.html#discretized-streams-dstreams)) to represent a stream of data and is implemented as a sequence of RDDs. Each RDD in a given DStream includes streaming data items from one particular interval. The following figure depicts a DStream with four RDDs each of which contain data items from different intervals:




*Spark Streaming DStream (source apache.org)*

DStreams API is quite similar to the Spark RDD API which creates input DStreams from different input sources and supports parallel operations over the formed sequences of RDDs. We will be discussing on DStreams operations later in this module.

## Streaming Data Sources

Spark Streaming supports two classes of built-in streaming sources based on the availability of input sources within the StreamingContext API (https://spark.apache.org/docs/latest/streaming-programming-guide.html#input-dstreams-and-receivers) each of which represents the stream of input data received from streaming sources. It also supports getting input data from any custom sources which needs extra code implementation of the receiver class.

### 1. Basic

These input sources (https://spark.apache.org/docs/latest/streaming-programming-guide.html#basic-sources) are directly available through the StreamingContext API such as file streams, streams based on custom receivers, and a queue of RDDs as a stream. The following displays a sample of creation of a DStream as file stream to read the data from a filesystem compatible with the HDFS API:

```
streamingContext.fileStream[KeyClass, ValueClass,
InputFormatClass](dataDirectory)
```

To define an input that reads from a queue of RDDs, you can use the `streamingContext.queueStream(queueOfRDDs)` method.

### 2. Advanced Sources

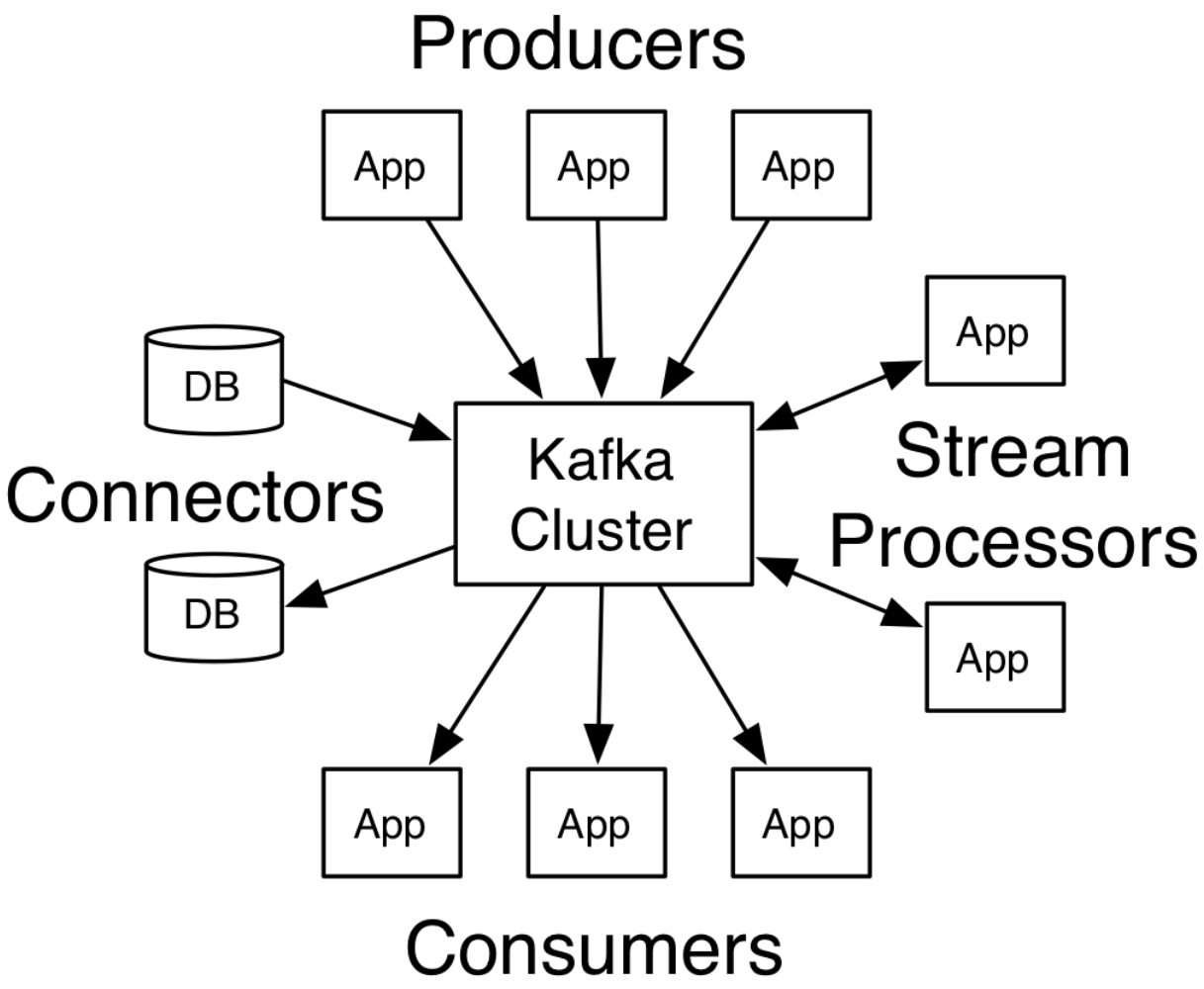
These input sources (https://spark.apache.org/docs/latest/streaming-programming-guide.html#advanced-sources) are not directly available in the StreamingContext API and can be accessed through additional utility classes.

Input data is ingested from sources like Kafka, Flume, and Kinesis the dependencies of which should be integrated (https://spark.apache.org/docs/latest/streaming-programming-guide.html#linking).

**2.1 Apache Kafka** (https://kafka.apache.org/) is a distributed, fault-tolerant, and scalable streaming platform which lets applications publish and subscribe to streams of records, store streams of records (in categories called ***topics***) efficiently, and process streams of records as they arrive (real-time). It has four core APIs:

1. **Producer API** (https://kafka.apache.org/documentation.html#producerapi) to publish a stream of record in one or more Kafka topics,
2. **Consumer API** **(https://kafka.apache.org/documentation.html#consumerapi)** that subscribes to one or more Kafka topics and computes the stream of records to them,
3. **Stream API** (https://kafka.apache.org/documentation/streams) or the stream processor which consumes the input stream from Kafka topics and produces an output stream to one or more Kafka output topics (transformation),
4. **Connector API** (https://kafka.apache.org/documentation.html#connect) which builds reusable producers/consumers to connect current Kafka topics to existing applications/data systems.

The following figure illustrates a sample interrelationship among these four Kafka APIs:

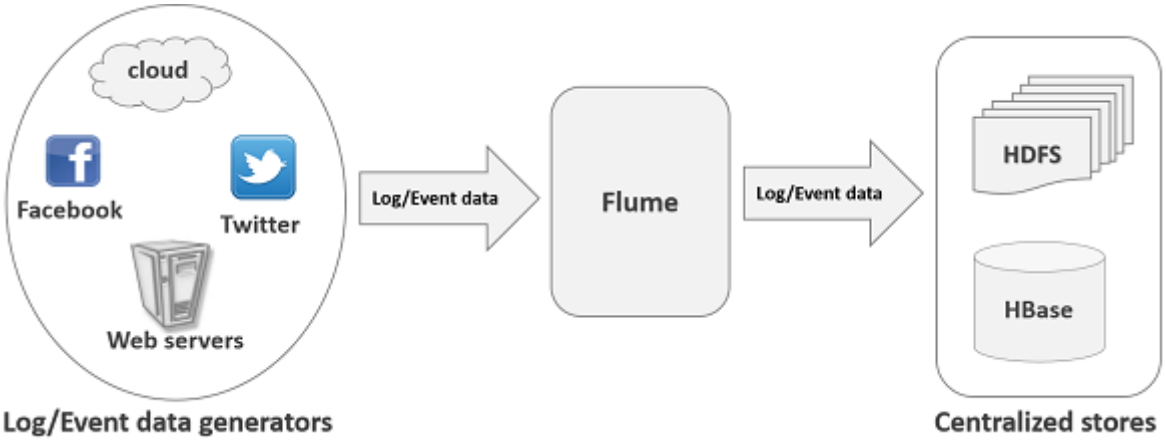



*Kafka APIs (source apache.org)*

This link (https://spark.apache.org/docs/latest/streaming-kafka-integration.html) provides further information in linking

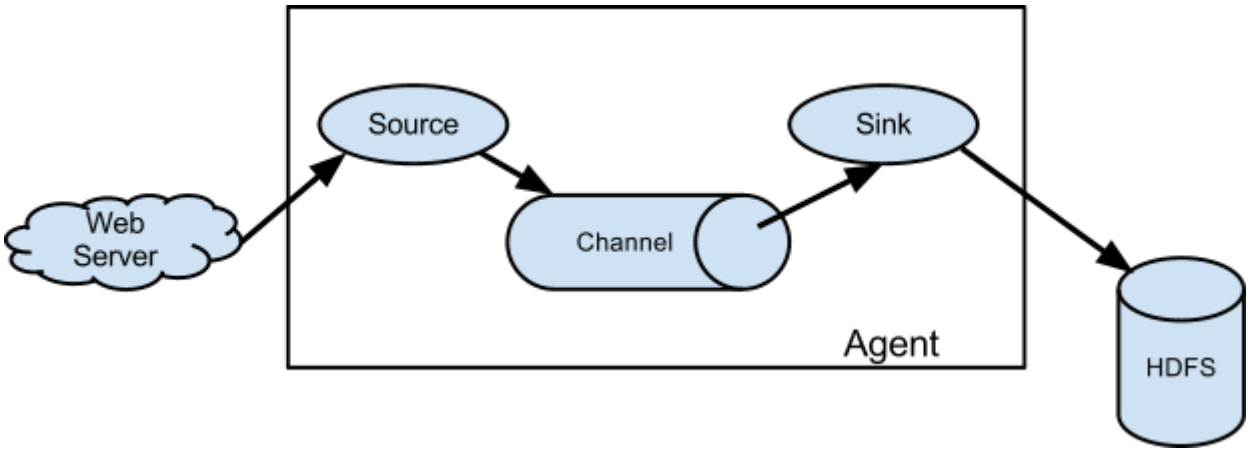
Spark Streaming with Apache Kafka.

**2.2 Apache Flume** (https://flume.apache.org/) is a distributed, robust, fault-tolerant, reliable, and available service for collecting, aggregating and moving a large amount of log data efficiently. The following figure displays the relation between Flume input and output:




*Apache Flume Input and Output (source: tutorialspoint.com)*

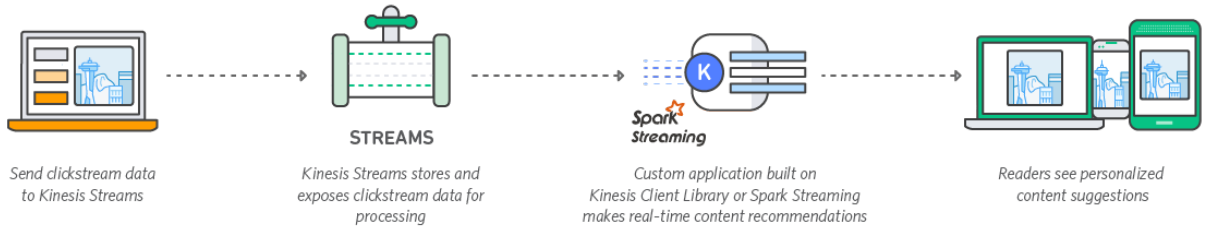
Apache Flume is not restricted to log data aggregation and is capable of transporting massive amount of streaming data such as network traffic data, social media-generated data, email messages and so forth. A Flume event is a unit of a data stream with a byte payload and optional set of string attributes. A Flume agent is a JVM process which includes hosts and components where event streams from external sources flow to the next target or hop. The arrived data streams (events) are stored in one or more channels till they are used by Flume sink. Flume sink removes the event from the channel and puts it into an external repository such as HDFS. The following figure illustrates a simple data flow model in Apache Flume:




*Apache Flume Data Flow (source: apache.org)*

This link (https://spark.apache.org/docs/latest/streaming-flume-integration.html) provides further information in linking Spark Streaming with Apache Flume.

**2.3 Amazon Kinesis** (https://aws.amazon.com/kinesis/) is a powerful platform to process (load, analyze) streaming data on Amazon web services (AWS). Amazon Kinesis provides several useful components such as Amazon Kinesis Firehose (https://aws.amazon.com/kinesis/firehose/), Amazon Kinesis Analytics (https://aws.amazon.com/kinesis/analytics/), and Amazon Kinesis Streams (https://aws.amazon.com/kinesis/streams/). The following figure depicts a high-level workflow entailing Amazon Kinesis Streams and Apache Streaming:




*Kinesis Streams (source: awsstatic.com)*

This link (https://spark.apache.org/docs/latest/streaming-kinesis-integration.html) provides further information in linking Spark Streaming with Amazon Kinesis.

### 3. Custom Sources

It is possible for the DStream data input object to be created from custom data sources (https://spark.apache.org/docs/latest/streaming-programming-guide.html#custom-sources) by implementing a user-defined receiver class which is able to receive data from any custom sources to be pushed into Spark.

## DStreams Transformations

As mentioned earlier, DStreams transformations (https://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams) enable us to modify the arrived data item (input DStream). DStreams support several core Spark RDDs transformations. You are already familiar with many of these transformations as they basically perform almost the same operations as their counterpart RDD transformations do. A list of common transformations is noted in the following table:

| Transformation | Meaning |
|---|---|
| **map**(*func*) | Returns a new DStream by passing each element of the source DStream through a function *func*. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to zero or more output items. |
| **filter**(*func*) | Return a new DStream by selecting only the records of the source DStream on which *func* returns true. |
| **repartition**(*numPartitions*) | Changes the level of parallelism in this DStream by creating more or fewer partitions. |
| **union**(*otherStream*) | Return a new DStream that contains the union of the elements in the source DStream and otherDStream. |
| **count**() | Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream. |
| **reduce**(*func*) | Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel. |
| **countByValue**() | When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream. |

| | |
|---|---|
| **reduceByKey**(*func*, [*numTasks*]) | When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. |
| ***join****(otherStream, [numTasks])* | When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key. |
| ***cogroup****(otherStream, [numTasks])* | When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples. |
| **transform**(*func*) | Return a new DStream by applying an RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream. |
| **updateStateByKey**(*func*) | Return a new "state" DStream where the state for each key is updated by applying the given function to the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key. |

The following displays a sample transform operation code segment where a real-time data cleaning is performed by joining the input data stream with precomputed spam information and filtering based on this:

```
 // RDD containing spam information:
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...)
// join data stream with spam information to do data cleaning
val cleanedDStream = wordCounts.transform { rdd =>
  rdd.join(spamInfoRDD).filter(...)
  ...
}
```

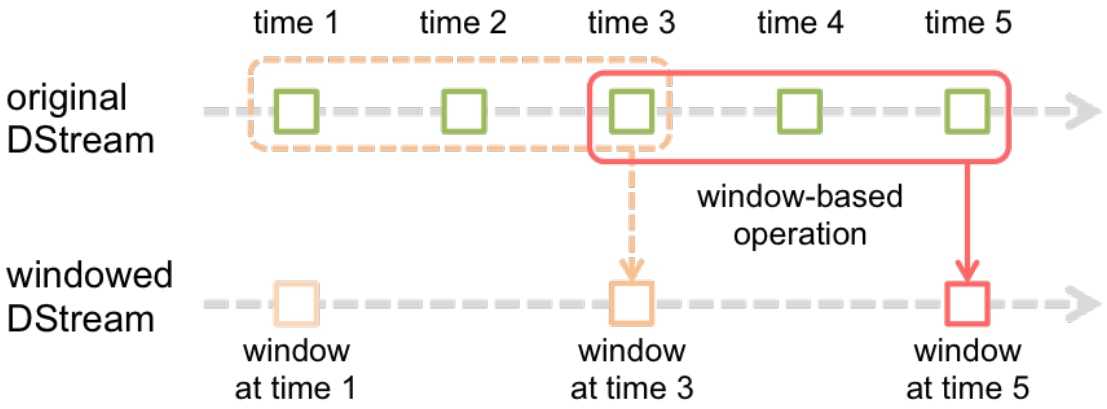
In the above code, ssc is a Spark Streaming Context. We will see more examples in the next activity section.

## DStreams Operations

There are also several DStream operations supported by Spark Streaming three of which are elaborated in the following.

### 1. Window Operations

Window [operations](https://spark.apache.org/docs/latest/streaming-programming-guide.html#window-operations) (https://spark.apache.org/docs/latest/streaming-programming-guide.html#window-operations) apply transformations over a sliding window of streaming data and are called **windowed computations**. This is done by defining a sliding window over the streaming data in DStream source. Within each window slide, the source DStream RDDs are clustered inside a window and the desired operation is performed on all RDDs within each window slide. Prior to utilizing window operations, **window length** (the duration of the window) and **sliding interval** (the interval at which the window operation is performed) parameters should be defined. The following figure represents a sample sliding window of length 3 with 2 intervals each:




*DStream Window (source: apache.org)*

The following table summarizes a list of common window operations:

| Window Operations | Meaning |
|---|---|
| **window** (*windowLength*, *slideInterval*) | Return a new DStream which is computed based on windowed batches of the source DStream. |
| **countByWindow** (*windowLength*, *slideInterval*) | Return a sliding window count of elements in the stream. |
| **reduceByWindow** (*func*, *windowLength*, *slideInterval*) | Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel. |
| **reduceByKeyAndWindow** (*func*, *windowLength*, *slideInterval*, [*numTasks*]) | When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window. |
| **reduceByKeyAndWindow** (*func*, *invFunc*, *windowLength*, *slideInterval*, [*numTasks*]) | A more efficient version of reduceByKeyAndWindow() where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enters the sliding window, and "inverse reducing" the old data that leaves the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable only to "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter *invFunc*). Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument. Note that checkpointing (https://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing) must be enabled for using this operation. |
| **countByValueAndWindow** (*windowLength*, *slideInterval*, [*numTasks*]) | When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument. |

## 2. Join Operations

Spark Streaming supports different kinds of joins (https://spark.apache.org/docs/latest/streaming-programming-guide.html#join-operations) such as stream-stream joins and stream-dataset joins. The following code segment displays one stream-stream join example:

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```

in which the RDDs generated in `stream1` and `stream2` are joined together. Spark Streaming also supports other kinds of joins like `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`. The following code snippet shows an example of stream-dataset join:

```
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }
```

## 3. Output Operations

These operators are concerned with pushing (https://spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations-on-dstreams) out the transformed DStream's data to external systems such as databases or file systems. The following table depicts a list of common output operations:

| Output Operation | Meaning |
|---|---|
| **print**() | Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. |
| **saveAsTextFiles**(*prefix*, [*suffix*]) | Save this DStream's contents as text files. The file name at each batch interval is generated based on `prefix` and `suffix: "prefix-TIME_IN_MS[.suffix]".` |
| **saveAsObjectFiles**(*prefix*, [*suffix*]) | Save this DStream's contents as SequenceFiles of serialized Java objects. The file name at each batch interval is generated based on *`prefix`* and `suffix: "prefix-TIME_IN_MS[.suffix]"`. |
| **saveAsHadoopFiles**(*prefix*, [*suffix*]) | Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on `prefix` and `suffix: "prefix-TIME_IN_MS[.suffix]"`. |
| **foreachRDD**(*func*) | The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs. |

### MLlib Operations

This is an amazing feature (https://spark.apache.org/docs/latest/streaming-programming-guide.html#mllib-operations) of Spark Streaming where you can use several machine learning (ML) algorithms supported by Spark MLlib (https://spark.apache.org/mllib/). MLlib is Apache Spark scalable ML library which supports both streaming and offline ML algorithms. MLlib can learn from the streaming data using streaming ML methods like Streaming Linear Regression or Streaming KMeans algorithms. It also enables applications to learn from offline models by using the historical data and apply the model to the streaming data online. MLlib supports a vast majority of ML algorithms like different classification, clustering, regression and collaborative filtering techniques.

# 5.3.1
# Activity 2: Counting Streaming Words

## Word Count Example

In this example, we want to modify the word count example that we developed for static data (e.g., a file that stored on HDFS) to work on streaming data. To create a streaming environment, we need to have two bash windows (simply connect to your VM twice). We use one of these connections as the sender of the text stream and the other one as the receiver. Indeed, we send data from the first one using the `nc` command and receive the data in the second one using our Scala program (Spark Streaming handles the listening part).

In the sender terminal issue one of the followings:

```
nc -lk 9999
cat input.txt | nc -lk 9999
```

In the above examples, 9999 is the port number (you can use any other available network port) that we send and receive the data through. The main difference between these two approaches is that the first one waits for our input (we should type whatever we want to send) while the second one automatically sends the content of the `input.text` and then waits for more input texts (if there is any). To terminate these senders, press `ctrl+d` or `ctrl+c`. If you like to see what you are sending before testing your Scala code, you can issue one of the followings in the receiver terminal:

```
nc localhost 9999
nc localhost 9999 | cat received.txt
```

As you probably guessed, the second example stores the received stream in a file. Note that the port number for the sender and receiver should be exactly the same.

The following is the first part of code example for our simple word count problem:

```
// Import the packages
import org.apache.spark._
import org.apache.spark.streaming._

// Force Spark to talk less and work more!
sc.setLogLevel("OFF")
sc.stop()

// Create a local StreamingContext with 2 working threads
// The master requires 2 cores to prevent from a starvation scenario.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
// Set batch interval of 1 second.
val ssc = new StreamingContext(conf, Seconds(1))
```

The above code loads the necessary packages and creates a new SparkContext. Nothing more!

Now, we need to create our DStreams that can connect to the server (the `nc` command). Note that the

sender is running in different terminal but still on the same computer. Therefore, the hostname should be `localhost`:

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

Note that we use the same port number as before.

The `lines` DStream represents the stream of received records. Each record of this DStream is a line of text and we need to break them down to lists of words. Which mapping should be used?

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

As we know, `flatMap` is a one-to-many DStream operator that creates new DStreams of multiple records per record of input DStream. Therefore, each line of the stream can be mapped to a list of zero or more words.

The rest of the task is simple. we need to count the words and print the statistics:

```
// Count each word in each batch
val wordCounts = words.map(word => (word, 1)).reduceByKey(_ + _)
// Print the results
wordCounts.print()
```

Our word count code seems to be ready. Now, we have to tell the Spark Streaming Context to start listening to the sender and count the words:

```
// Start the computation
ssc.start()
// Also, wait for 10 seconds or termination signal
ssc.awaitTerminationOrTimeout(1000)
ssc.stop()
```

## Windowing

In the above example, we count the frequency of all words in the stream. In many cases, we cannot process the data as fast as it is generated. Therefore, we need to perform our analysis on a selected sample of the data rather than the complete set. With the facilities that Spark Streaming library provides, this is not a complicated task. For example, if we can easily modify the word count example to only count the words in a window. To do so, we can replace the `reduceByKey` by `reduceByKeyAndWindow`. For instance, the following code every 10 seconds counts the words in a window with 30 seconds length.

```
// Count words in 30 sec windows
val wordCounts = words.map(word => (word, 1)).reduceByKeyAndWindow(_+_,
Seconds(30), Seconds(10))
// Print the results
wordCounts.print()
```

Before running the above code, make sure your stream is long enough to see the windowing result (or reduce the length of the window).

**Quick Quiz:** How do the window length and frequency (30 and 10 seconds in the last example) affect the accuracy and speed of an application? Can you practically demonstrate your answer?

# 5.4
# Streaming Pattern Analysis

## Introduction

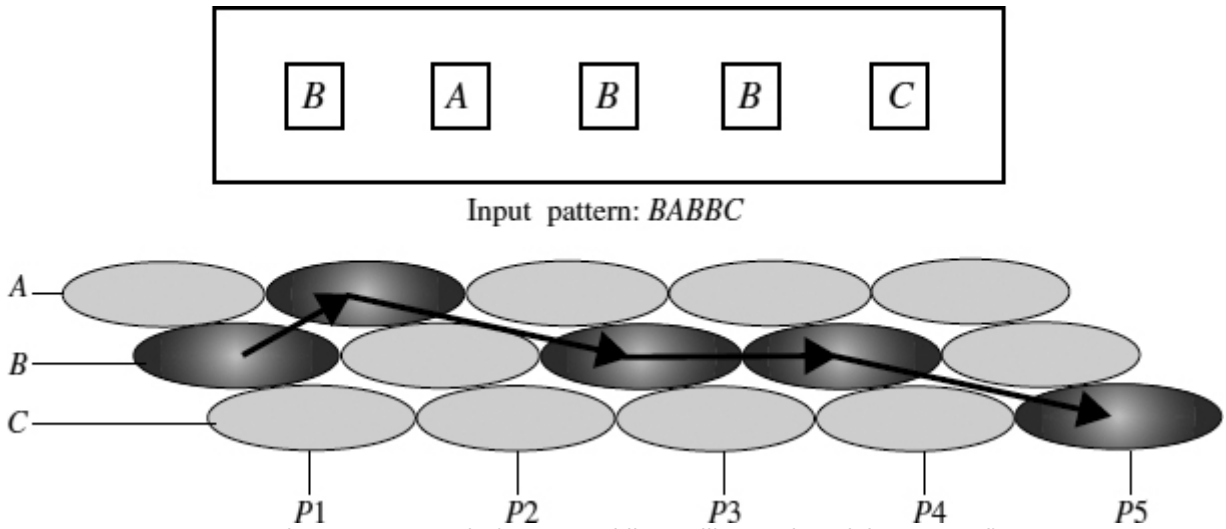
We have discussed on several streaming data processing algorithms so far. Now, we will elaborate on one particular branch of streaming data analysis which is called ***streaming pattern analysis*** or data stream mining. It refers to the process of eliciting insight and detecting patterns from rapidly streaming data records (https://en.wikipedia.org/wiki/Data_stream_mining). In most cases, relevant machine learning techniques are utilized to predict the class/value of arriving data items based on the acquired knowledge on the class membership/values of previously observed and processed data records. One way to cope with the issue of increasing computational overhead of pattern recognition of streaming data records is adopting the graph matching algorithms. Graphs, in general, support universal representation formalism and are helpful in processing and visualizing emerging patterns among data items. However, with the growth on arriving streaming data records and patterns size, even traditional graph-based algorithms cannot handle the issue.

## Graph Neuron

To address this challenge, a new concept is defined: **graph neuron (GN)** as a novel approach that uses graph-based representations of patterns to achieve one-shot learning. A highly-scalable associative memory device is thus created, which is capable of handling multiple streams of input that are processed and matched with the historical data stored within the network. The method uses parallel in-network processing to circumvent the pattern database scalability limitation associated with graph-based techniques (http://ieeexplore.ieee.org/abstract/document/4359217/).

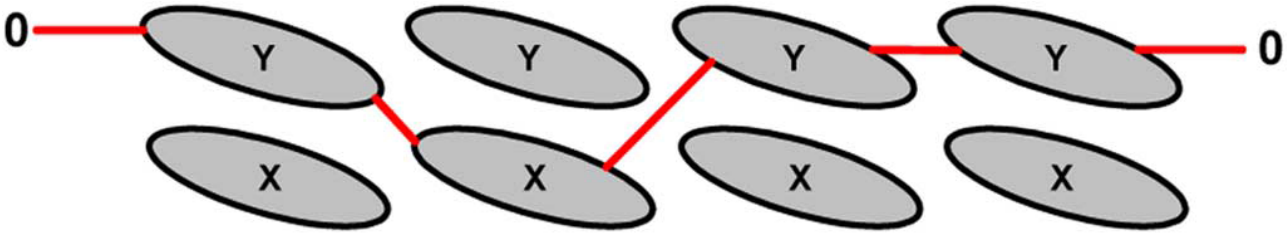
The GN is a finely distributed in-network pattern-recognition algorithm that preserves the data relationships in a graph-like memory structure. The GN structure and its data representations are analogous to a directed graph, the processing nodes of the GN array are mapped as the vertex set $V$ of the graph, and the inter-node connections (i.e., the communication channels) belong to the set of edges, $E$. The communications are restricted to the adjacent nodes (of the array), hence there is no increase in the communication overheads with corresponding increases in the number of nodes in the network. The information presented to each of the nodes is in the form of a (value, position) pair. Each of these pairs, in its simplest form, represents a data point in a two-dimensional reference pattern space. Hence, the GN array converts spatial/temporal patterns into a graph-like structural representation and then compares the edges of the graph with subsequent inputs for memorization or recall.

The following figure represents the simple GN with input pattern '*BABBC*' and how it is stored and represented within a GN:




*Graph Neuron Example (source Mobile Intelligence by Srinivasan et al)*

In the following figure (http://ieeexplore.ieee.org/abstract/document/4359217/), a sample pattern string '*YXYY*' is represented in a 2-D GN which has 2 rows and 4 columns:




*Logical connectivity within the GN array representing an input pattern of YXYY (credit: Nasution and Khan)*

The red lines show the logical connectivity within the GN array to represent the pattern.

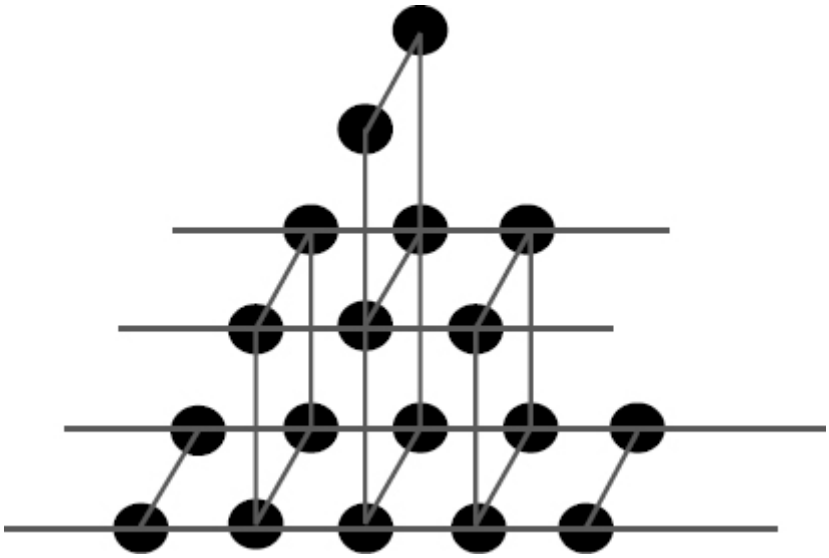
# Associative Memory

Before proceeding with the GN, we need to elaborate on another concept: **associative memory (AM)**. AM is derived from the neural network model and has been applied in many different application areas. A widely used unsupervised learning technique is the Hopfield network (https://en.wikipedia.org/wiki/Hopfield_network). This network has been widely used for implementing associative (or content-addressable) memory in pattern analysis and optimization. A study of Hopfield memory model shows that the model is not scalable and is limited by the number of processing/storage nodes in the network. The backpropagation network provides fast recalls, but the training cost becomes excessive for adding newer patterns. Ideally, an associative memory device should include simple one-shot training in the case of discrete time processing and fast retrieval. The GN is an approach that aims to overcome the scalability issues and reduce the training overheads in associative memory devices.

# Hierarchical Graph Neuron (HGN)

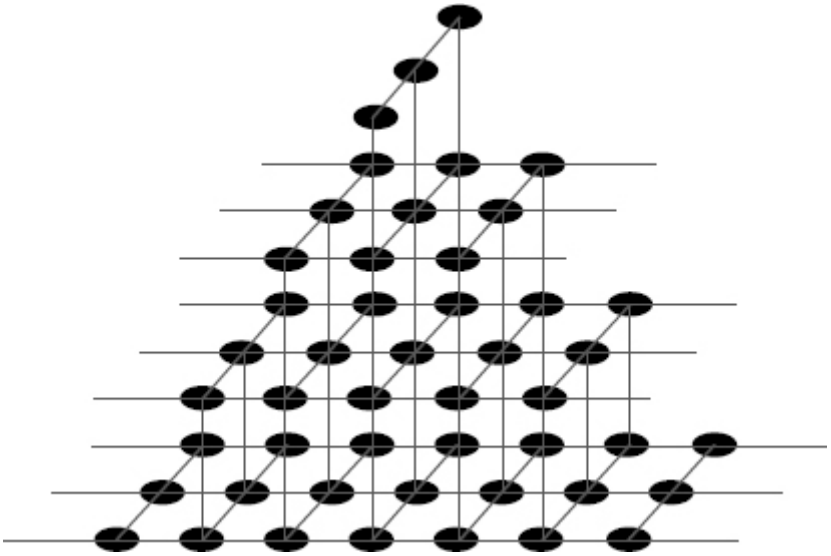
As an improvement to GN to handle scalability, HGN is composed of layers of GN networks arranged in a pyramid-like composition. The base layer corresponds to the size of the pattern to be used in the pattern recognition application. The size of the pattern is equal to the number of elements in the pattern. Each element may be assigned a fixed number of possible values from the pattern domain. For instance, a

black and white bitmap representation would have two possible values, that is, '1' or '0', for every pixel position (http://ieeexplore.ieee.org/abstract/document/4359217/). The following figure depicts an HGN architecture for pattern size 5 and two possible values:



*Hierarchical Graph Neuron (source Mobile Intelligence by Srinivasan et al)*

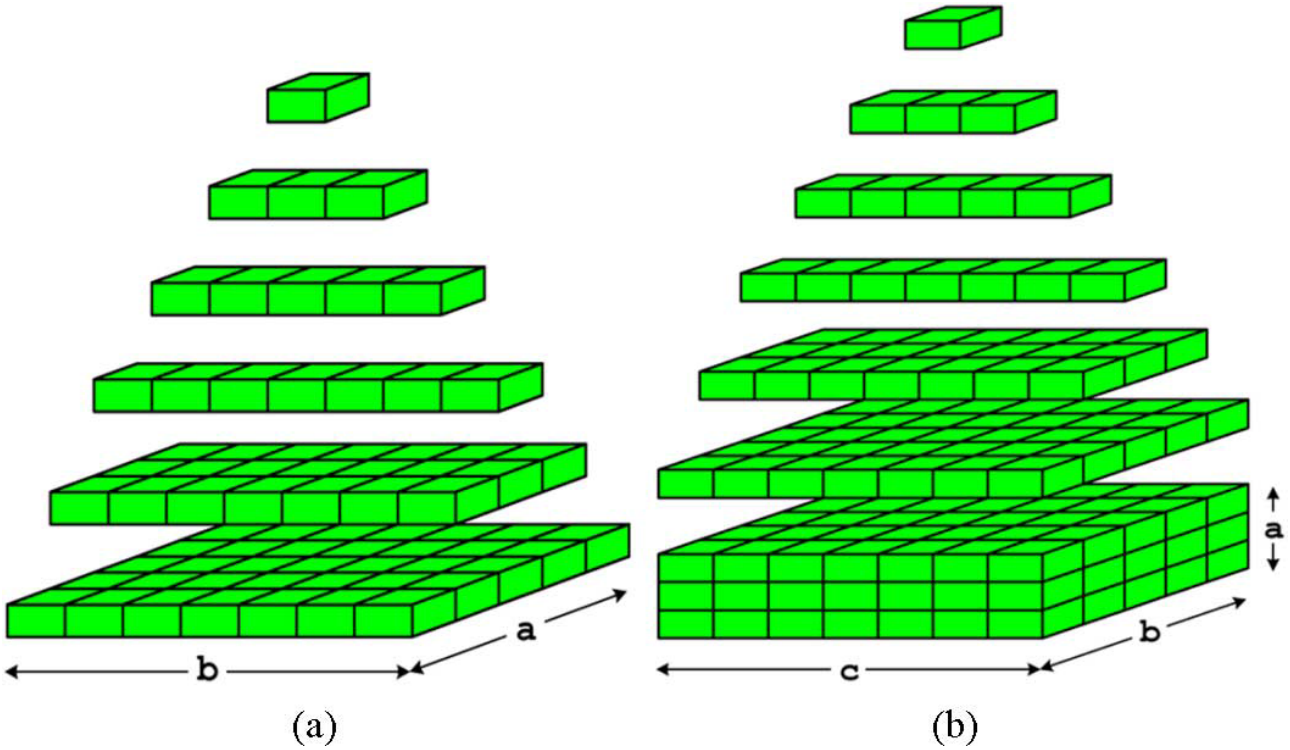
Put simply, HGN adds the support for a hierarchical structure to GN and is a fully distributable online pattern recognizer. Its benefit is that the top GN nodes are able to provide oversight for the base layer, and thus eliminate the cross-talk. Each layer of the HGN acts as a simple GN network with the difference that the higher layers (all layers except the base layer) would store the pair values as $p(\text{index}_{left}, \text{index}_{middle}, \text{index}_{right})$. Where $\text{index}_{left}, \text{index}_{middle}$, and $\text{index}_{right}$ represent the inputs received from the left, bottom, and right GNs respectively. Each of these indices represents the row number in the bias arrays corresponding to the input pattern. The following figure represents an HGN with pattern size 7 and 3 values:



*An HGN Example Hierarchical Graph Neuron (source Mobile Intelligence by Srinivasan et al)*

It is possible to have 2-D and 3-D representations of HGN as depicted in the following figure (http://ieeexplore.ieee.org/abstract/document/4359217/). Part (a) of the following figure represent a 2-D HGN composition with $(7 \times 5) = 35$ pattern size, and part (b) represents a 3-D HGN with

$(7 \times 5 \times 3) = 105$ pattern size:

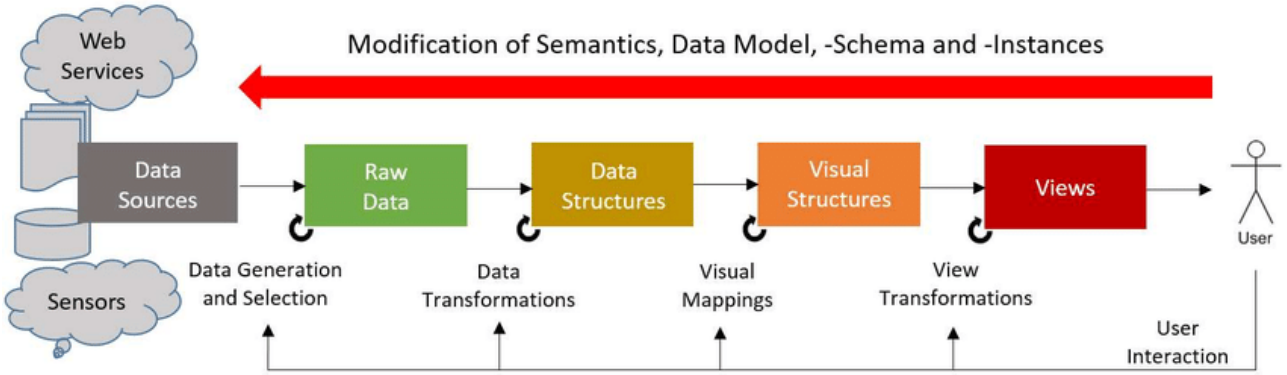



*A 2D Example of HGN (credit: Nasution and Khan)*

# 5.5
# Dynamic Data Visualization

Visualizing data streams which are generated from different sources have become an emerging trend. Visualization of streaming data is strongly related to its temporal context and very often methods that map time to the horizontal axis are used to visualize the data stream. How do we define which part of past data is relevant for the current data and the current point in time? Although, the data being generated and delivered in the streams has a strong temporal component, in many cases it is not only the temporal component that the analysts are interested in. There are other important data dimensions that are equally important and time might be just an additional aspect that they care about. In those cases, we might want to rely on other visualization methods that can show other attributes better than temporal visualizations. How should the visualization change when we add new data? Does the whole layout should be recomputed when we add just one element like in force-directed graphs, or we can easily add it like in scatterplots?

The following figure illustrates the information visualization reference model proposed in this work (https://books.google.com.au/books?hl=en&lr=&id=wdh2gqWfQmgC&oi=fnd&pg=PR13&dq=Readings+in+Information+Visualization:+Using+Vision+to+Think&ots=omHF2vtJMy&sig=JWEZMTAz1nZrFapDRIbxRMd4YIQ#v=onepage&q=Readings%20in%20Information%20Visualization%3A%20Using%20Vision%20to%20Think&f=false):




*The Information Visualization Reference Model by Danowski-Buhren et al.*

In the following, we will be presenting a taxonomy of dynamic data visualizations based on this (http://ieeexplore.ieee.org/document/6400552/) research paper. The dimensions of this taxonomy are spatial and retinal variable treatments as well as identity.

## Spatial Variable Treatment

The spatial variable treatment refers to the position of the data items in the streaming flow. It can be dynamic in changing existing values and changing the number of visual elements (addition/deletion). These two dynamic changes are categorized into four classes:

- **Fixed** - there is no change in data item's position. Also, the number of visual data elements are fixed.
- **Mutable** - the position of the data element is changing but the number of data elements is fixed during the visualization process.
- **Create** - new data elements are created in arriving streams.
- **Create & Delete** - data elements may be created or removed on data records arrival.

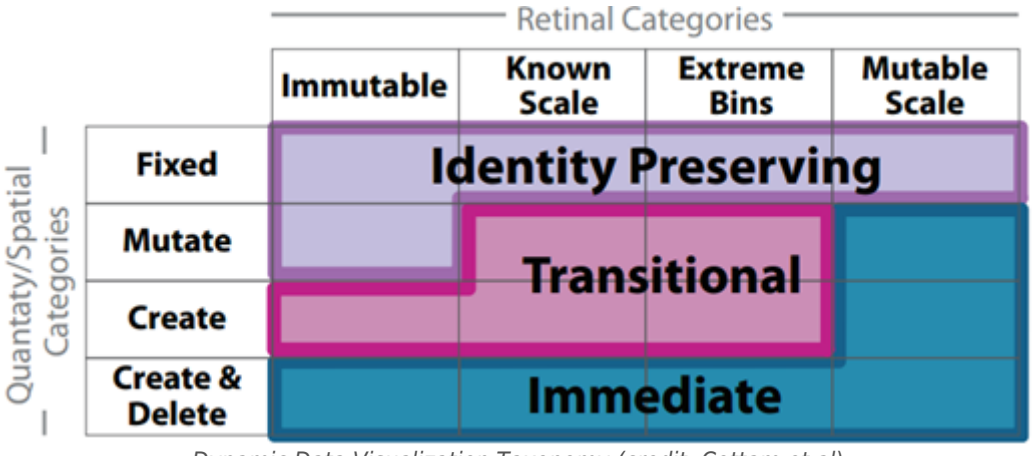
## Retinal Variable Treatment

The retinal variable treatments do not have any impact on the number of data elements' positions, but they can affect the way arriving data elements are grouped. The retinal parameters are size, value, orientation, texture, color, and shape. There are four classes for this parameter:

- **Immutable** - no change in the retinal variable.
- **Known Scale** - The scale is fixed, but future data may change the retinal presentation of an existing element. The known scale implies that all values that may be presented are covered by the scale's regular divisions, which are established when the visualization is initialized. For categorical encodings, known scale means that the number and order of categories are known in advance. For continuous encodings, the range of values is provided in advance and divided into regular intervals.
- **Known Scale** - An extreme bins scale is a known scale with sentinel categories (usually at the endpoints). These sentinel categories may be due to an unknown actual extent of values, special/sentinel categories, or to provide details in a specific sub-range. Values outside of the regular range are assigned to a top or bottom catch-all "bin". Scales like this typically include labels such as "100+", "less than 0", "No Data" or "Error". These open-ended categories are distinguished and outside of the normal divisions of the scale.
- **Mutable Scale** - Updates may change the representation of an element and the mapping function itself. In this category, scales may grow or shrink dynamically to accommodate the data being visualized. For example, this category may apply if the number of levels in a categorical variable is not known in advance. When modifying scales, a single new data point may cause existing visual elements to change representations, even though the underlying data did not change.

## Identity

The comparison is one of the most crucial tasks in visualization interpretation. A visualization that includes dynamics, however, complicates this task of comparison by inviting comparisons to be made across time. The following taxonomy borrowed from the work of Cottam et al (http://ieeexplore.ieee.org/document/6400552/) examines how dynamics influence the task of comparison over time.



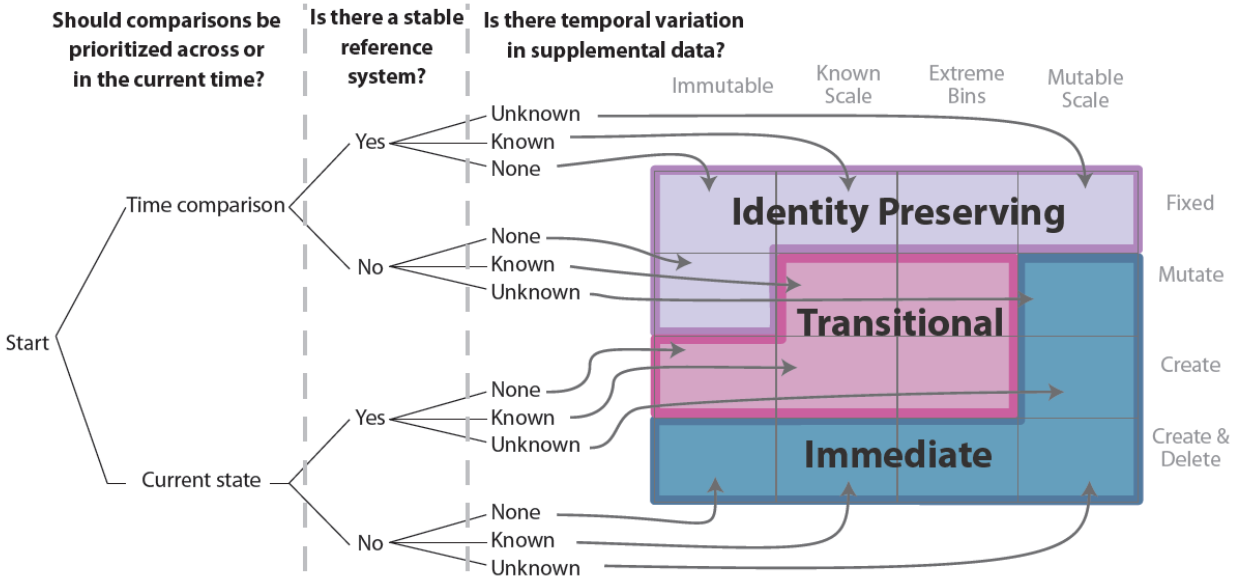

*Dynamic Data Visualization Taxonomy (credit: Cottam et al)*

As the above figure presents, the dynamic data visualization techniques are categorized into three distinct groups: Preserving, Transitional and Immediate. The identity preserving techniques retain the association between visual elements and underlying data across the time scales. These techniques represent a great deal of dynamic information especially when the range of dynamic inputs are known and the comparisons are made over long periods of time.

The second group, transitional techniques, often maintain identity associations, but the association is not as strong as the first group, the preserving techniques. indeed, they hold the identity associations across time by limiting changes to only known values. In contrast with the preserving techniques, the transitional techniques provide fine-grained comparison over short time spans. They also offer greater degrees of freedom for presenting and contextualizing
current information than the preserving techniques.

Finally, the immediate techniques usually do not maintain identity associations. Immediate techniques are valuable in providing detailed information in the most flexible format right now. However, they are poor for making comparisons across time steps.

The following decision tree can be very helpful in choosing a right type of visualization to illustrate a given stream/dynamic data:




*Decision tree for mapping from a specific task to a visualization technique (credit: Cottam et al)*

## References

For more on this topics, please refer to the following references:


1. Watch this: A taxonomy for dynamic data visualization (http://ieeexplore.ieee.org/document/6400552/)
2. Introducing streaming k-means in Apache Spark 1.2
   (https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html)
3. Understanding your Apache Spark Application Through Visualization
   (https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html)
4. Apache Zeppelin for Big Data Visualization (https://zeppelin.apache.org/)
5. The Open Graph Viz Platform (https://gephi.org/)

# 6
# Advanced Topics in Big Data

*Big Data Analytics (http://www.wipo.int)*

In this module, we focus on the following topics regarding Advanced topics in Big Data with Machine Learning and Cluster Performance Measurement Factors:

- A short review of the main Machine Learning Algorithms,
- Apache Spark Machine Learning library (Spark MLlib),
- Introduction to the Data and Compute Clusters,
- Introduction to Apache Hadoop YARN, Apache MESOS, Beowulf Cluster as Cluster Computing Platforms and Software, and
- Investigating key cluster performance metrics.

## Learning Objectives

At the end of this module, you should be able to:

- Demonstrate knowledge of tree searching and be able to implement simple decision trees designed for training and execution for Big Data.
- Demonstrate an understanding of high-level computer clustering provisions within Hadoop and the Beowulf approach for implementing low-level clustering.
- Performance test compute clusters using message passing interface (MPI).

## References

As usual, the main references for this modules are mainly from the Apache projects documentations such as:

- Machine Learning Library (MLlib) Guide (http://spark.apache.org/docs/latest/ml-guide.html)
- Scientific Computing, Machine Learning, and Natural Language Processing (http://www.scalanlp.org/)
- ML Pipelines (http://spark.apache.org/docs/latest/ml-pipeline.html#how-it-works)
- Scala kNN API (https://spark.apache.org/docs/2.1.0/api/scala/index.html#org.apache.spark.ml.clustering.KMeans)
- Scala Random Forest API (https://spark.apache.org/docs/2.1.0/api/scala/index.html#org.apache.spark.ml.classification.RandomForestClassifier)
- Spark Classification and Regression Guide (https://spark.apache.org/docs/2.1.0/ml-classification-regression.html)
- Machine Learning Library (MLlib) Programming Guide (https://spark.apache.org/docs/2.1.0/mllib-guide.html)
- ML Tuning: model selection and hyperparameter tuning (http://spark.apache.org/docs/latest/ml-tuning.html)
- Apache Hadoop YARN (Yet another resource negotiator) (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html)
- Apache Mesos (http://mesos.apache.org/documentation/latest/architecture/)
- Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center (http://mesos.berkeley.edu/mesos_tech_report.pdf)
- Beowulf cluster (https://en.wikipedia.org/wiki/Beowulf_cluster)
- Open MPI: Open Source High-Performance Computing (https://www.open-mpi.org/)

# 6.1
# Machine Learning - A Brief Review

**Machine Learning (ML)**, as a subset of computer science field, is a data analysis approach which makes the analytical model building automated or "to learn without being explicitly programmed" using algorithms that iteratively learn from data (https://en.wikipedia.org/wiki/Machine_learning). In general, ML algorithms are able to learn from the fed data and apply that knowledge in predicting similar/other kinds of data. This capability is referred to as **Predictive Analytics** - learning from current or historical events to extrapolate likely future events (https://en.wikipedia.org/wiki/Predictive_analytics). ML tasks are typically categorized into the following groups given the nature of learning: **Supervised Learning** - where the input data elements are tagged (supervised) with the help of an external expert. Input data is called **training data** and has a known label or result at a time. The system then learns from those input elements and applies its knowledge to map the input to the output. A model is prepared through a training process in which it is required to make predictions and is corrected when those predictions are wrong. The training process continues until the model achieves the desired level of accuracy on the training data; **Unsupervised Learning** - through which no tagged/labeled data elements are given to the algorithm. In this case, a model is prepared by deducing structures present in the input data. This may be to extract general rules. It may be through a mathematical process to systematically reduce redundancy, or it may be to organize data by similarity. **Semi-Supervised Learning** - where the input is a combination of tagged and untagged elements. A model must learn the structures to organize the data as well as make predictions.

To formulate diverse ML techniques, one needs to formalize them using algorithms. Some common ML algorithms are as the following:

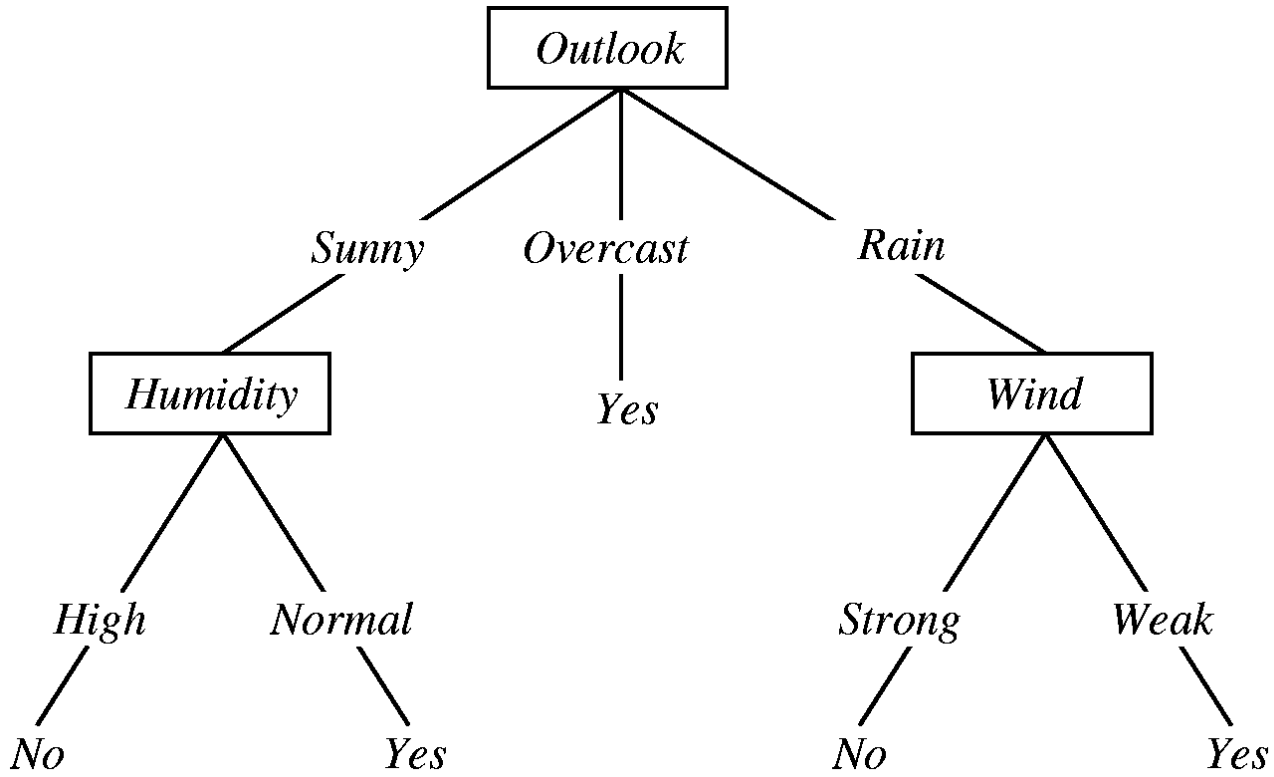
- **Classification -** given the set of input elements classes (current or historical data), the model should identify the classes of the new data elements. The model learns from the training data set (the input) and identifies to which category (class) the new data elements belong. Classification is considered an instance of **Supervised Learning**. The outputs in Classification are discrete.
- **Regression** - is similar to Classification except that the outputs are continuous.
- **Clustering** - unlike Classification and Regression, Clustering algorithms are instances of **Unsupervised Learning** where a set of observed data elements are assigned into subsets (clusters) in a way that observations in the same cluster are similar to each other (given some criteria).

Another important aspect of ML is selecting proper attributes/features of the input data to build an accurate predictive model. The process of selecting input data attributes which are most relevant to the predictive model problem automatically is typically called **Feature Selection (http://machinelearningmastery.com/an-introduction-to-feature-selection/)**. **Feature Transformation** is another term that is concerned with putting training data elements in an optimal format for learning and generalization (https://docs.aws.amazon.com/machine-learning/latest/dg/data-transformations-for-machine-learning.html). To make the input data ready for transformation, one first needs to reduce the huge volume of data into useful dimensions. This process is called **Dimensionality Reduction** and it is focused on reducing the number of random variables given certain criteria (https://en.wikipedia.org/wiki/Dimensionality_reduction). This process can be divided into feature selection and feature extraction sub-categories.

In the next chapter, we focus on one particular supervised learning algorithm: **Decision Trees**.

# 6.1.1
# Decision Trees (DT)

A Decision Tree (https://en.wikipedia.org/wiki/Decision_tree) constructs a model of decisions made based on actual values of attributes in the data. Decisions fork in tree structures until a prediction decision is made for a given record. Decision trees are trained on data for classification and regression problems. Decision trees are often fast and accurate and a big favorite in machine learning (https://en.wikipedia.org/wiki/Decision_tree_learning). Generally, DTs are tree-like graph decisions (similar to flowchart structure) that comprise **Decision Node(s)** which represent a test on an attribute (and each branch denotes an outcome of the test), and **End Node(s)** that represent the class label and are the final decision after performing all tests through the intermediate decision nodes. The path from the root node to the leaf node (end node) is typically called the Classification Rule (https://en.wikipedia.org/wiki/Decision_tree_learning). The following figure illustrates a simple DT to decide whether the Tennis match should be played or not given the weather condition.




*A Decision Tree Example (source: jmvidal.cse.sc.edu)*

# 6.2
# Spark Machine Learning Library

Apache Spark provides its Spark MLlib as a scalable ML library to developers (https://spark.apache.org/mllib/). Given the benchmarks, Spark MLlib performs up to 100 times faster than traditional MapReduce by utilizing high-quality algorithms. The main reason is that Spark MLlib is capable of performing ML tasks iteratively compared to MapReduce's one-pass approximations. Spark MLlib's primary ML API is DataFrame-based from Spark 2.0 (https://spark.apache.org/docs/latest/ml-guide.html). To perform ML tasks, Spark MLlib utilizes a rich set of APIs and components such as:

- ML Pipelines, Extracting, Transforming, and Selection Features,
- Diverse set of ML algorithms such as Classification, Regression, Clustering, and Collaborative Filtering, and
- Various Optimization algorithms to improve the computational performance.

In the subsequent sections, each of above-mentioned items is elaborated.

# 6.2.1 Spark Machine Learning Pipelines

ML Pipelines are high-level APIs built on top of DataFrames to assist us in building ML pipelines. ML Pipeline entails key concepts such as (https://spark.apache.org/docs/latest/ml-pipeline.html): DataFrame, Transformer, Estimator, and the Pipeline. In the following, we briefly review these concepts.

## DataFrame

As we mentioned in the previous modules, Spark DataFrame possesses a diverse set of data types like text, vector, label, predictions. Therefore, they are the primary data structures that we use when developing an ML application in Spark.

## Transformer

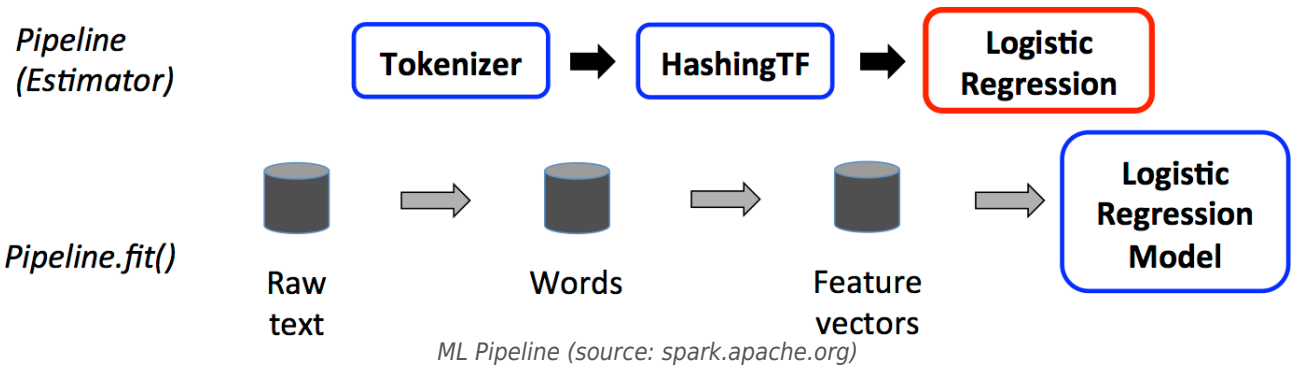
Generally speaking, a transformer is an algorithm which transforms one DataFrame into another one. For example, an ML model is a transformer as it transforms the input DataFrame (including its features) into an output DataFrame (with prediction). Data wrangling operations such as hashing, normalization, and reformation all are transformers.

## Estimator

An estimator is another algorithm that is fit on a given input DataFrame to generate a Transformer. For example, a Decision Tree Classifier (as a learning algorithm) is an Estimator as it trains on a given input DataFrame and generates a learning model.

## Pipeline

A pipeline, as its name suggests, acts as a glue in chaining different Transformers and Estimators together to form a complete ML workflow. The following figure represents a simple text document processing workflow comprising splitting each document into its constituted words (tokenization), converting those words into numerical feature vectors (hashing), and learn a prediction model utilizing the feature vectors and labels (logistic regression).

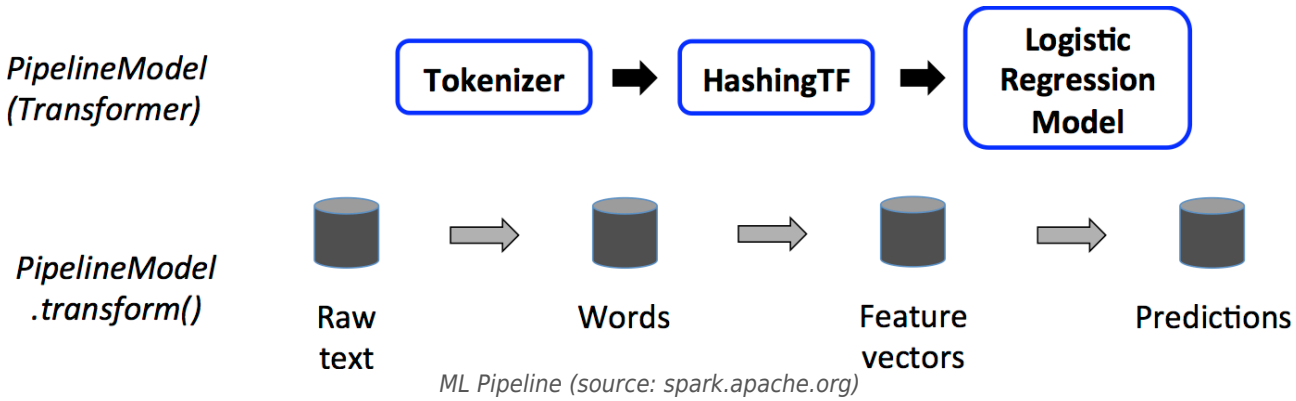



*ML Pipeline (source: spark.apache.org)*

The above figure was an example of an Estimator Pipeline. As depicted in the figure, a pipeline is

illustrated as a sequence of steps, each of which could be a Transformer or an Estimator. The crucial point here is the order that those steps should perform (like a workflow). For the Transformer steps, the `transform()` method is called on the DataFrame and the `fit()` method is called for the Estimator steps. As per the figure, in the top row pipeline, the first 2 out of 3 steps are Transformers and the last one is an Estimator. The bottom row represents the data flow through the pipeline (Cylinders indicate the DataFrames). The `Pipeline.fit()` method is called on the original DataFrame, which has raw text documents and labels. The `Tokenizer.transform()` method splits the raw text documents into words, adding a new column with words to the DataFrame. The `HashingTF.transform()` method converts the words column into feature vectors, adding a new column with those vectors to the DataFrame. Now, since Logistic Regression is an Estimator, the Pipeline first calls `LogisticRegression.fit()` to produce a ***Logistic Regression Model***. If the Pipeline had more Estimators, it would call the LogisticRegressionModel's `transform()` method on the DataFrame before passing the DataFrame to the next stage.

In the following figure, we illustrate a Transformer Pipeline. Please note that the pipeline is named *PipelineModel* now.




*ML Pipeline (source: spark.apache.org)*

In the figure above, the *PipelineModel* has the same number of stages as the original Pipeline, but all Estimators in the original Pipeline have become Transformers. When the PipelineModel's `transform()` method is called on a test dataset, the data are passed through the fitted pipeline in order. Each stage's `transform()` method updates the dataset and passes it to the next stage.

# 6.2.1.1
# Activity: Spark MLlib

## A Simple Classification Example

In this example, we demonstrate a simple classification pipeline using Logistic Regression algorithm. For the educational purposes, we use a very small artificial dataset, here.

### 1. Load Libraries

In the first step, we need to load a few libraries such as logistic regression from ML library:

```
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.sql.Row
```

### 2. Create Training Data

As mentioned above, we create a small synthetic dataset and store it as a DataFrame.

```
// Prepare training data from a list of (label, features) tuples.
val training = spark.createDataFrame(Seq(
  (1.0, Vectors.dense(0.0, 1.1, 0.1)),
  (0.0, Vectors.dense(2.0, 1.0, -1.0)),
  (0.0, Vectors.dense(2.0, 1.3, 1.0)),
  (1.0, Vectors.dense(0.0, 1.2, -0.5))
)).toDF("label", "features")
```

In the above example, `training` DataFrame consists of two parts: a label column and three feature columns. The labels are either 0 or 1, but the domain of the data can take any continuous value.

### 3. Create an Estimator

Now, we create our logistic regression model as an estimator. We can also set the model parameters such as the maximum number of iterations at this stage.

```
// Create a LogisticRegression instance. This instance is an Estimator.
val lr = new LogisticRegression()
// Print out the parameters, documentation, and any default values.
println("LogisticRegression parameters:\n" + lr.explainParams() + "\n")
// We may set parameters using setter methods.
lr.setMaxIter(10).setRegParam(0.01)
```

## 4. Train the Model

We already created the model. Now, we should train it on the training data using `fit` function.

```
// Learn a LogisticRegression model. This uses the parameters stored in lr.
val model1 = lr.fit(training)
```

We can also easily print out the parameters of the fitted model:

```
// Since model1 is a Model (i.e., a Transformer produced by an Estimator),
// we can view the parameters it used during fit().
// This prints the parameter (name: value) pairs, where names are unique IDs for
this
// LogisticRegression instance.
println("Model 1 was fit using parameters: " + model1.parent.extractParamMap)
```

Note that we can change the parameter values using `ParamMap`:

```
// We may alternatively specify parameters using a ParamMap,
// which supports several methods for specifying parameters.
val paramMap = ParamMap(lr.maxIter -> 20)
  .put(lr.maxIter, 30)  // Specify 1 Param. This overwrites the original
maxIter.
  .put(lr.regParam -> 0.1, lr.threshold -> 0.55)  // Specify multiple Params.
```

or even change the name of the output variable and retrain the model:

```
// One can also combine ParamMaps.
val paramMap2 = ParamMap(lr.probabilityCol -> "myProbability")  // Change output
column name.
val paramMapCombined = paramMap ++ paramMap2
// Now learn a new model using the paramMapCombined parameters.
// paramMapCombined overrides all parameters set earlier via lr.set* methods.
val model2 = lr.fit(training, paramMapCombined)
println("Model 2 was fit using parameters: " + model2.parent.extractParamMap)
```

## 5. Create the Test Data

The models are trained. To use the model, we should have a test data. Note that the structure of the test data should be similar to the training data.

```
// Prepare test data.
val test = spark.createDataFrame(Seq(
  (1.0, Vectors.dense(-1.0, 1.5, 1.3)),
  (0.0, Vectors.dense(3.0, 2.0, -0.1)),
  (1.0, Vectors.dense(0.0, 2.2, -1.5))
)).toDF("label", "features")
```

## 6. Perform Prediction

The final step is to predict the labels of the test data using the model we trained on the training data.

```
// Make predictions on test data using the Transformer.transform() method.
```

```
// LogisticRegression.transform will only use the 'features' column.
// Note that model2.transform() outputs a 'myProbability' column instead of the
usual
// 'probability' column since we renamed the lr.probabilityCol parameter
previously.
model2.transform(test)
  .select("features", "label", "myProbability", "prediction")
  .collect()
  .foreach { case Row(features: Vector, label: Double, prob: Vector, prediction:
Double) =>
    println(s"($features, $label) -> prob=$prob, prediction=$prediction")
  }
```

That's it!

# A Text Processing Example

In this example (https://spark.apache.org/docs/latest/ml-pipeline.html#example-pipeline), we practice the logistic regression once again. However, the dataset we use here is a simple text dataset. Since logistic regression only takes numeric values, we have to perform some data transformation to convert the text datasets to a numeric form. There are many different approaches one can take to perform such transformation. In this example, we use the widely used Term Frequency technique.

In the first step, let's import the required classes for this example:

```
//start by importing needed classes
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.sql.Row
```

Next, we need to have our training documents prepared from a list of given tuples of `id, text`, and `label`:

```
// Prepare training documents from a list of (id, text, label) tuples.
val training = spark.createDataFrame(Seq(
  (0L, "a b c d e spark", 1.0),
  (1L, "b d", 0.0),
  (2L, "spark f g h", 1.0),
  (3L, "hadoop mapreduce", 0.0)
)).toDF("id", "text", "label")
```

Now, having the trained documents, we need to configure our ML pipeline by following the three main steps of tokenizer, hashingTF, and Logistic Regression (lr). The `tokenizer` part will be setting the input and output columns as `text` and `words`, respectively. The hashingTF will be setting the number of desired features, setting the input column which is the tokenizer's defined output column, and defining its output column which is named `features` in this example. Next, the Logistic Regression part will be defining how many iterations should be made along with the regression parameter. Finally, we have all the ingredients required for constructing our pipeline. So, we create a pipeline instance and feed it with our recently generated elements.

```
// Configure an ML pipeline, which consists of three stages: tokenizer,
hashingTF, and lr.
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setNumFeatures(1000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.001)
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))
```

Now, it's time to fit our defined pipeline instance to the training documents:

```
// Fit the pipeline to training documents.
val model = pipeline.fit(training)
```

In the meantime, if you like (optionally), you can store your fitted and unfitted pipelines to the disk and retrieve them back.

```
// Now we can optionally save the fitted pipeline to disk
model.write.overwrite().save("/tmp/spark-logistic-regression-model")

// We can also save this unfit pipeline to disk
pipeline.write.overwrite().save("/tmp/unfit-lr-model")

// And load it back in during production
val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")
```

Next, let's have our test documents prepared. They are in the unlabeled `(id, text)` records format. We need to create the corresponding DataFrame for that:

```
// Prepare test documents, which are unlabeled (id, text) tuples.
val test = spark.createDataFrame(Seq(
  (4L, "spark i j k"),
  (5L, "l m n"),
  (6L, "spark hadoop spark"),
  (7L, "apache hadoop")
)).toDF("id", "text")
```

Finally, having all the required ingredients, we can now make predictions on test documents using the created model's `transform()` method.

```
// Make predictions on test documents.
model.transform(test)
  .select("id", "text", "probability", "prediction")
  .collect()
  .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double)
=>
    println(s"($id, $text) --> prob=$prob, prediction=$prediction")
  }
```

All done! Congratulations, you have just used the logistic regression approach in terms of Term Frequency technique.

We are done with the example. The following code snippet shows the whole process in one seamless piece:

```
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.sql.Row

// Prepare training documents from a list of (id, text, label) tuples.
val training = spark.createDataFrame(Seq(
  (0L, "a b c d e spark", 1.0),
  (1L, "b d", 0.0),
  (2L, "spark f g h", 1.0),
  (3L, "hadoop mapreduce", 0.0)
)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer,
hashingTF, and lr.
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setNumFeatures(1000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.001)
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))

// Fit the pipeline to training documents.
val model = pipeline.fit(training)

// Now we can optionally save the fitted pipeline to disk
model.write.overwrite().save("/tmp/spark-logistic-regression-model")

// We can also save this unfit pipeline to disk
pipeline.write.overwrite().save("/tmp/unfit-lr-model")

// And load it back in during production
val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")

// Prepare test documents, which are unlabeled (id, text) tuples.
val test = spark.createDataFrame(Seq(
  (4L, "spark i j k"),
  (5L, "l m n"),
  (6L, "spark hadoop spark"),
  (7L, "apache hadoop")
)).toDF("id", "text")
```

```
// Make predictions on test documents.
model.transform(test)
  .select("id", "text", "probability", "prediction")
  .collect()
  .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double)
=>
    println(s"($id, $text) --> prob=$prob, prediction=$prediction")
  }
```

For further readings on the Extracting, Transforming and Selecting features, please refer to this link (https://spark.apache.org/docs/latest/ml-features.html). It includes useful examples in **Feature Extraction** techniques (such as TF-IDF, Word2Vec, CountVectorizer), **Feature Transformation** methods (like Tokenizer, StopWordsRemover, n-gram, PCS, ...), **Feature Selection** approaches (such as VectorSlicer, RFormula, ChiSqSelector), and **Locality Sensitive Hashing** techniques (like different LSH Operations and Algorithms).

# 6.2.2
# ML Algorithms Coverage

Spark ML covers a wide variety of ML algorithms. For a complete list of Classification and Regression, and Clustering Algorithms please refer to the following links respectively: classification and regression algorithms (https://spark.apache.org/docs/latest/ml-classification-regression.html#classification-and-regression), clustering algorithms (https://spark.apache.org/docs/latest/ml-clustering.html). The following lists some well-known algorithms in each category and we will proceed with elaborating on Decision Tree algorithm in detail:

- **Classification** **(https://spark.apache.org/docs/latest/ml-classification-regression.html#classification)** **Algorithms** - Logistic regression (Binomial logistic regression, Multinomial logistic regression), Decision tree classifier, Random forest classifier, Gradient-boosted tree classifier, Multilayer perceptron classifier, One-vs-Rest classifier (a.k.a. One-vs-All), Naive Bayes
- **Regression** **(https://spark.apache.org/docs/latest/ml-classification-regression.html#regression)** **Algorithms** - Linear regression, Generalized linear regression (Available families), Decision tree regression, Random forest regression, Gradient-boosted tree regression, Survival regression, Isotonic regression
- **Clustering** **(https://spark.apache.org/docs/latest/ml-clustering.html)** **Algorithms** - K-means, Latent Dirichlet Allocation (LDA), Bisecting k-means, Gaussian Mixture Model (GMM).

# 6.2.2.1 Decision Tree Algorithm in Spark MLlib

In this section, we provide two code examples for the Decision Trees usage in Classification and Regression algorithms (https://spark.apache.org/docs/latest/mllib-decision-tree.html). But prior to mentioning the code, please note the following input and output descriptions of any Decision (or Ensemble) Tree (https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-trees):

- **Decision Tree Inputs**

Input Columns

| Param name | Type(s) | Default | Description |
|---|---|---|---|
| labelCol | Double | "label" | Label to predict |
| featuresCol | Vector | "features" | Feature vector |

- **Decision Tree Outputs (Prediction)** - Please note that all the output columns are optional

Output Columns

| Param name | Type(s) | Default | Description | Notes |
|---|---|---|---|---|
| predictionCol | Double | "prediction" | Predicted label | |
| rawPredictionCol | Vector | "rawPrediction" | Vector of length # classes, with the counts of training instance labels at the tree node which makes the prediction | Classification only |
| probabilityCol | Vector | "probability" | Vector of length # classes equal to rawPrediction normalized to a multinomial distribution | Classification only |
| varianceCol | Double | | The biased sample variance of prediction | Regression only |

Now, we can proceed with two code samples of Decision Trees (https://spark.apache.org/docs/latest/mllib-decision-tree.html):

- **Decision Trees in Classification (https://spark.apache.org/docs/latest/mllib-decision-tree.html#classification)**
- The process is done in four simple steps:
- 1) importing the relevant classes required for the classification task at hand and loading/parsing the data file:

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split the data into training and test sets (30% held out for testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
```

- 2) training the decision tree model using the `train()` method:

```
// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32

val model = DecisionTree.trainClassifier(trainingData, numClasses,
categoricalFeaturesInfo,
  impurity, maxDepth, maxBins)
```

- 3) validating and evaluating the created model by applying that on the test data and calculating the error:

```
// Evaluate model on test instances and compute test error
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count().toDouble /
testData.count()
println("Test Error = " + testErr)
println("Learned classification tree model:\n" + model.toDebugString)
```

- 4) finalizing the process and storing/retrieving the created and tested decision tree model:

```
// Save and load model
model.save(sc, "target/tmp/myDecisionTreeClassificationModel")
val sameModel = DecisionTreeModel.load(sc,
"target/tmp/myDecisionTreeClassificationModel")
```

- Now, the whole process in one single code snippet:

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split the data into training and test sets (30% held out for testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32

val model = DecisionTree.trainClassifier(trainingData, numClasses,
categoricalFeaturesInfo,
  impurity, maxDepth, maxBins)
```

```
// Evaluate model on test instances and compute test error
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count().toDouble /
testData.count()
println("Test Error = " + testErr)
println("Learned classification tree model:\n" + model.toDebugString)

// Save and load model
model.save(sc, "target/tmp/myDecisionTreeClassificationModel")
val sameModel = DecisionTreeModel.load(sc,
"target/tmp/myDecisionTreeClassificationModel")
```

- **Decision Trees in Regression (https://spark.apache.org/docs/latest/mllib-decision-tree.html#regression)**
- Similar to the above-mentioned decision tree example, we have four simple steps to use the decision tree technique with regression. Please note the use of the `trainRegressor()` method in step 2 and compare that to the similar one of the decision trees in classification example mentioned above.
- 1) importing, loading and parsing the data file:

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split the data into training and test sets (30% held out for testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
```

- 2) training the decision tree model using the `trainRegressor()` method:

```
// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "variance"
val maxDepth = 5
val maxBins = 32

val model = DecisionTree.trainRegressor(trainingData,
categoricalFeaturesInfo, impurity,
  maxDepth, maxBins)
```

- 3) validating and evaluating the created model:

```
// Evaluate model on test instances and compute test error
val labelsAndPredictions = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testMSE = labelsAndPredictions.map{ case (v, p) => math.pow(v - p, 2)
}.mean()
```

```
println("Test Mean Squared Error = " + testMSE)
println("Learned regression tree model:\n" + model.toDebugString)
```

- 4) storing/retrieving the created and tested decision tree model:
- ```
  // Save and load model
  model.save(sc, "target/tmp/myDecisionTreeRegressionModel")
  val sameModel = DecisionTreeModel.load(sc,
  "target/tmp/myDecisionTreeRegressionModel")
  ```
- Now, once again, the whole process in one seamless code segment:
- ```
  import org.apache.spark.mllib.tree.DecisionTree
  import org.apache.spark.mllib.tree.model.DecisionTreeModel
  import org.apache.spark.mllib.util.MLUtils

  // Load and parse the data file.
  val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
  // Split the data into training and test sets (30% held out for testing)
  val splits = data.randomSplit(Array(0.7, 0.3))
  val (trainingData, testData) = (splits(0), splits(1))

  // Train a DecisionTree model.
  //  Empty categoricalFeaturesInfo indicates all features are continuous.
  val categoricalFeaturesInfo = Map[Int, Int]()
  val impurity = "variance"
  val maxDepth = 5
  val maxBins = 32

  val model = DecisionTree.trainRegressor(trainingData,
  categoricalFeaturesInfo, impurity,
    maxDepth, maxBins)

  // Evaluate model on test instances and compute test error
  val labelsAndPredictions = testData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
  }
  val testMSE = labelsAndPredictions.map{ case (v, p) => math.pow(v - p, 2)
  }.mean()
  println("Test Mean Squared Error = " + testMSE)
  println("Learned regression tree model:\n" + model.toDebugString)

  // Save and load model
  model.save(sc, "target/tmp/myDecisionTreeRegressionModel")
  val sameModel = DecisionTreeModel.load(sc,
  "target/tmp/myDecisionTreeRegressionModel")
  ```

# 6.2.3 Optimization Algorithms

To optimize linear methods' performance, Spack MLlib provides three different optimization methods (https://spark.apache.org/docs/latest/ml-advanced.html):

- Limited-memory BFGS (L-BFGS)
- Normal equation solver for weighted least squares
- Iteratively reweighted least squares (IRLS)

For further information on how these methods perform, please refer to this link (https://spark.apache.org/docs/latest/ml-advanced.html).

# 6.3 Introduction to the Data and Compute Clusters

Given the recent adoption of digital technologies and increasing needs for computation and robustness, most applications need to support **performance** and **fault-tolerance**. Cluster computing has emerged to address this need and provides concepts/tools for application developers to cope with the mentioned issues. Clusters provide the **computational power** using parallel programming techniques (coordinating the use of many processors for a single problem). Please note that clusters are not proper tools in cases including accelerating calculations which are neither memory intensive nor processing power intensive or those requiring frequent communication between the processors in the cluster. Another reason for using clusters is to provide **fault tolerance**, that is, to ensure that computational power is always available. Because clusters are assembled from many copies of the same or similar components, the failure of a single part only reduces the cluster's power. Thus, clusters are particularly good choices for environments that require guarantees of available processing power, such as Web servers and systems used for data collection. In general, clustering can be viewed as either of **Data Clusters** or **Compute Clusters**. In the following sub-sections, we elaborate on each case.

# 6.3.1
# Data Cluster

Basically, a data cluster or allocation unit refers to the smallest amount of logical disk space allocated to store a file (https://en.wikipedia.org/wiki/Data_cluster). A cluster can hold files and directories on disk. The filesystem typically allocates a set of contiguous sectors to store files (which are called clusters) instead of allocating individual sectors by default. The following figure illustrates different allocation concepts on a given disk:

*Disk-structure (https://upload.wikimedia.org)*

To generalize the concept of data clusters in distributed and clustered systems, the concept of **Clustered File System** has emerged (https://en.wikipedia.org/wiki/Clustered_file_system). To reduce the computation complexity and increase the system's reliability through utilization of location-independent addressing and redundancy, the Clustered File System shares a filesystem among various servers, simultaneously. A sub-set of Clustered File Systems are **Parallel File Systems** which disseminate the data among various storage nodes and guarantee the performance and redundancy.

Another approach to distributing the data among various nodes is **Data Parallelism** where the data is disseminated to multiple processors in a parallel computing environment (https://en.wikipedia.org/wiki/Data_parallelism). This way, the data is distributed among different nodes. For further readings on Data Parallelism, please refer to the following link (https://en.wikipedia.org/wiki/Data_parallelism).

Large-scale disk reads and writes and Data parallelism is main issues to be addressed by any data parallelism environment.

# 6.3.2 Compute Cluster

This kind of cluster comprises a set of interconnected computers which are coupled loosely/tightly and work together to form one integrated whole system (https://en.wikipedia.org/wiki/Computer_cluster). Each computer in the compute cluster plays as a server and is called a **node** which can run its own operating system and may be different from others in terms of hardware configuration (although in most cases, all nodes follow the same configurations). Each of these nodes is connected to each other using fast local area networks. The main motivation behind compute cluster was to improve **performance** and **availability** compared to the case of having just one computer. Another factor is that the compute cluster is usually more **cost-effective** than enhancing the single computer to the same hardware configuration. One basic approach to building a computer cluster is that of a **Beowulf Cluster** - mentioned in the section 6.4.3. Compute clusters support computation-intensive tasks using a high availability paradigm (https://en.wikipedia.org/wiki/Computer_cluster). One way to provide this robustness is through **load-balancing** clusters where the cluster nodes share their computational workload to help improve performance.

Also, multiple nodes in a compute cluster communicate (https://en.wikipedia.org/wiki/Computer_cluster) with each other using one of the two methods: **Parallel Virtual Machine (PVM)** or **Message Passing Interface (MPI)**. In PVM, the virtual machine needs to be installed on every node and provides various software packages and libraries which seem to be discomforting! Typically, PVM supports message-passing, resource management, and task management through a run-time environment. On the other hand, by using the TCP/IP and socket connections, MPI provides a specification which can be implemented in different systems. Compare the compulsory need for each node in PVM to have the virtual machine be installed and concretely implemented with MPI which provides a specification for you to implement it yourself! This makes the MPI be a widespread communications model to support parallel programming in clusters. Nodes are typically interconnected with a communication network such as Ethernet (https://en.wikipedia.org/wiki/Ethernet), Myrinet (https://en.wikipedia.org/wiki/Myrinet), Quadrics, or Infiniband (https://en.wikipedia.org/wiki/InfiniBand) for MPI communication.

To sum up, Compute Clusters provide environments for performing mainly CPU-bound tasks by enabling data and task parallelism with guaranteeing performance and high-availability. One simple example of such a system could be a Beowulf cluster. Furthermore, all nodes inside the compute cluster are communicating with each other using the MPI.

For further readings on MPI, please refer to the following link (https://computing.llnl.gov/tutorials/mpi/). Also, for some code examples in low-level cluster programming with MPI, please refer to this link (http://www.meccanismocomplesso.org/en/cluster-e-programmazione-in-parallelo-con-mpi-e-raspberry-pi/).

# 6.4
# Cluster Computing Platforms and Software

In this section, we investigate three prominent cluster computing platforms/frameworks which support cluster computing and provide environments to develop applications using the cloud:

- **Apache Hadoop YARN**,
- **Apache MESOS**, and
- **Beowulf Cluster**

# 6.4.1 Apache Hadoop YARN

YARN is Apache Hadoop's distributed application manager over the cluster (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html). It provides resource management and a central platform for various operations to be performed across Hadoop clusters.

YARN improves a Hadoop compute cluster in the following ways

| Feature | Description |
|---|---|
| Multi-tenancy | YARN allows multiple access engines (either open-source or proprietary) to use Hadoop as the common standard for batch, interactive and real-time engines that can simultaneously access the same data set.<br>Multi-tenant data processing improves an enterprise's return on its Hadoop investments. |
| Cluster utilization | YARN's dynamic allocation of cluster resources improves utilization over more static MapReduce rules used in early versions of Hadoop. |
| Scalability | Data center processing power continues to rapidly expand. YARN's ResourceManager focuses exclusively on scheduling and keeps pace as clusters expand to thousands of nodes managing petabytes of data. |
| Compatibility | Existing MapReduce applications developed for Hadoop 1 can run YARN without any disruption to existing processes that already work. |

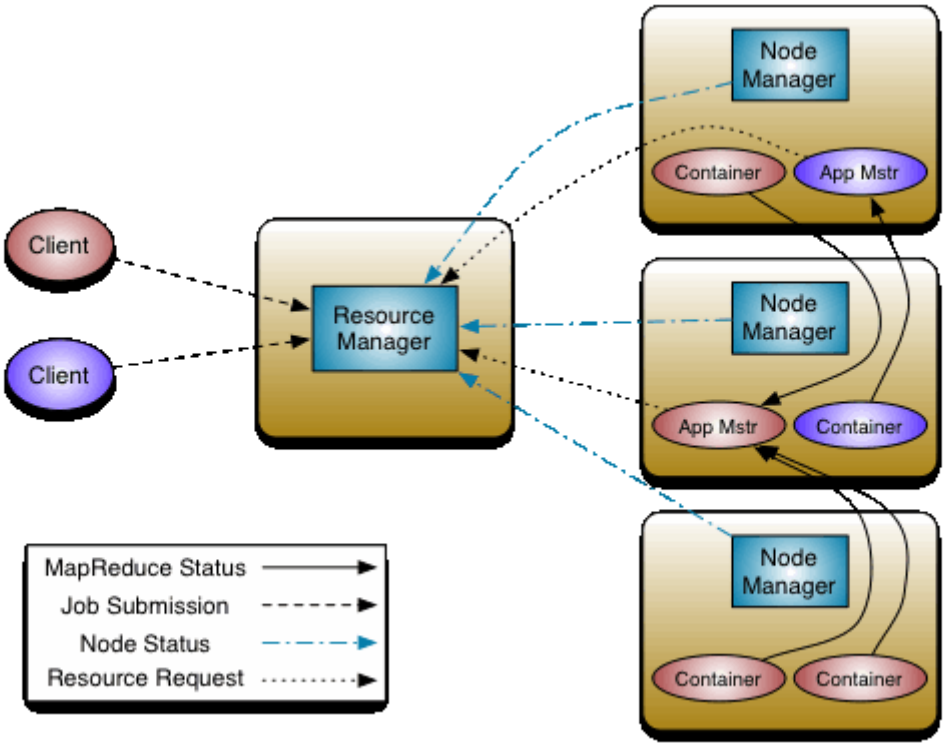
Yarn comprised four major components as its main purpose was to split up two key responsibilities of the Job Tracker and Task Tracker into separate entities:

- A global **ResourceManager (RM)** - which is a scheduler and is concerned with arbitrating available resources in the system among the competing applications.
- A per-application **ApplicationMaster (AM)** - which is an instance of a framework specific library and is responsible for negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the containers and their resource consumption. It has the responsibility of negotiating appropriate resource containers from the ResourceManager, tracking their status and monitoring progress. The ApplicationMaster enables YARN to support the following characteristics:
  - *Scale:* The AM provides almost all functionalities of the traditional RM so that the entire system can scale more. This is one of the key reasons to design the RM as a pure scheduler as it doesn't attempt to provide fault-tolerance for resources. Besides, since there is an instance of an ApplicationMaster per application, the ApplicationMaster itself isn't a common bottleneck in the cluster.
  - *Open:* Moving all application framework-specific code into the ApplicationMaster makes the system more general that multiple frameworks such as MapReduce, MPI and Graph Processing can be supported.
- A per-node slave **NodeManager (NM)** - The NodeManager is the per-machine framework agent responsible for Containers, monitoring their resource usage (CPU, RAM, Disk, Network, ...) which reports to the ResourceManager/Scheduler. The per-application ApplicationMaster is a framework specific library, per se, and its task is to negotiate resources from the ResourceManager and work with the NodeManager(s) to execute and monitor the tasks.
- A per-application **Container** running on a NodeManager - YARN is designed to allow individual applications (via the ApplicationMaster) to utilize cluster resources in a shared, secure and multi-tenant manner. Also, it remains aware of cluster topology to efficiently schedule and optimize data access which means reducing data motion for applications to the extent possible. To address those objectives, the Central Scheduler (in ResourceManager) has extensive information about an

application's resource needs, which allows it to make better scheduling decisions across all applications in the cluster. This leads us to the ResourceRequest and the resulting **Container**. The **Container** is the resource allocation, which is the successful result of the ResourceManager granting a specific ResourceRequest. A Container grants rights to an application to use a specific amount of resources (RAM, CPU, Network, Disk, ...) on a certain host.

A given application may be a single job or a DAG of jobs. The ResourceManager and NodeManager together build the data computation framework which forms a new generic system for managing applications in a distributed manner.

The following figure depicts the architecture of Apache YARN



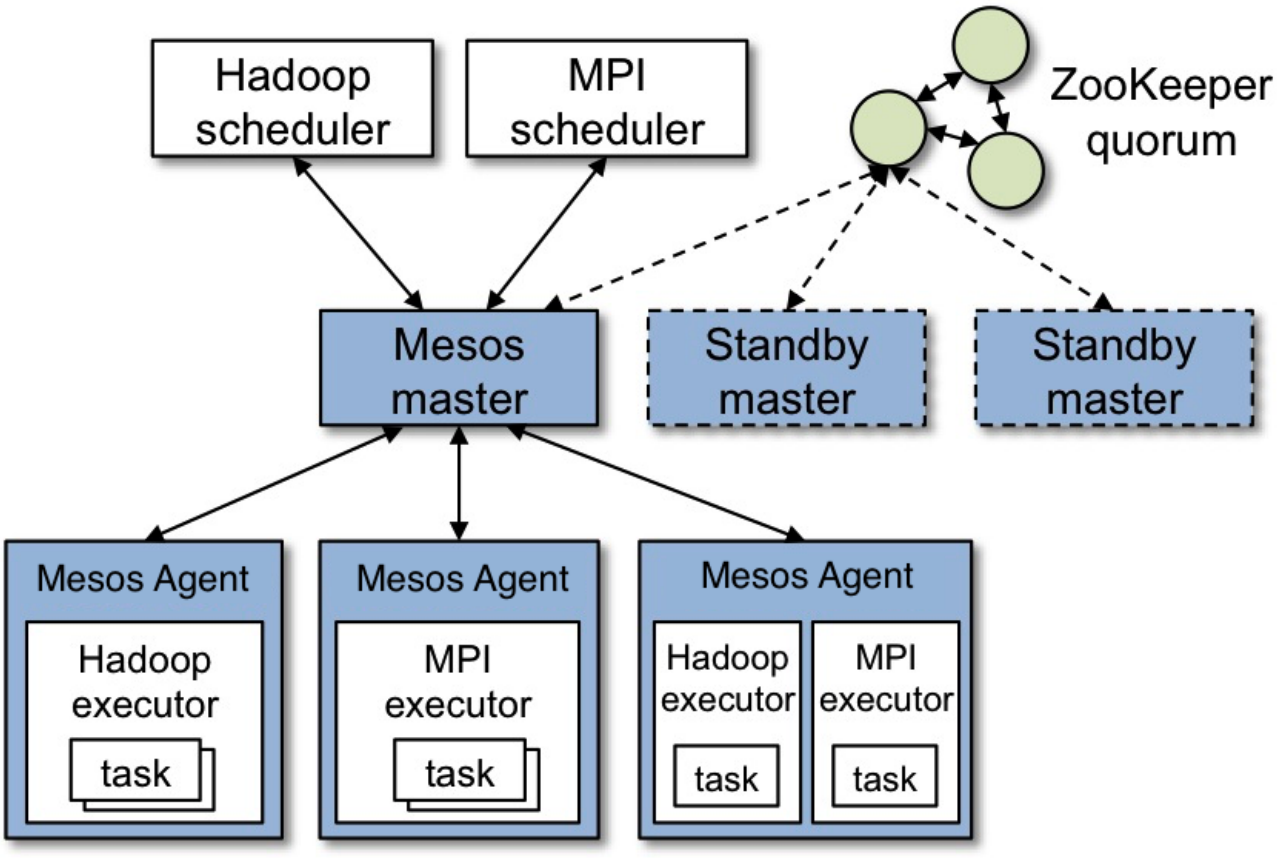

*Apache Hadoop Yarn Architecture (https://hadoop.apache.org)*

For further readings on Apache YARN, please refer to the following link (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html).

# 6.4.2 Apache MESOS

Apache Mesos is a distributed systems kernel (https://mesos.apache.org/). It is a cluster manager that provides efficient resource isolation and sharing across distributed applications or frameworks. According to Apache, Mesos kernel runs on every machine and provides applications (e.g., Hadoop, Spark, Kafka, Elasticsearch) with API for resource management and scheduling across entire datacenter and cloud environments. Apache Mesos enables the following characteristics (https://mesos.apache.org/): linear scalability (in terms of nodes), high availability (utilizing fault-tolerant replicated master and agents with the help of Zookeeper), pluggable isolation (support for CPU, memory, disk, ports, GPU, and modules for custom resource isolation), two level scheduling (running cloud native and legacy applications), and being cross platform which runs on various platforms.

Mesos is an open source software originally developed at the University of California at Berkeley. It sits between the application layer and the operating system and makes it easier to deploy and manage applications in large-scale clustered environments more efficiently. It can run many applications on a dynamically shared pool of nodes. The following figure illustrates Mesos architecture:



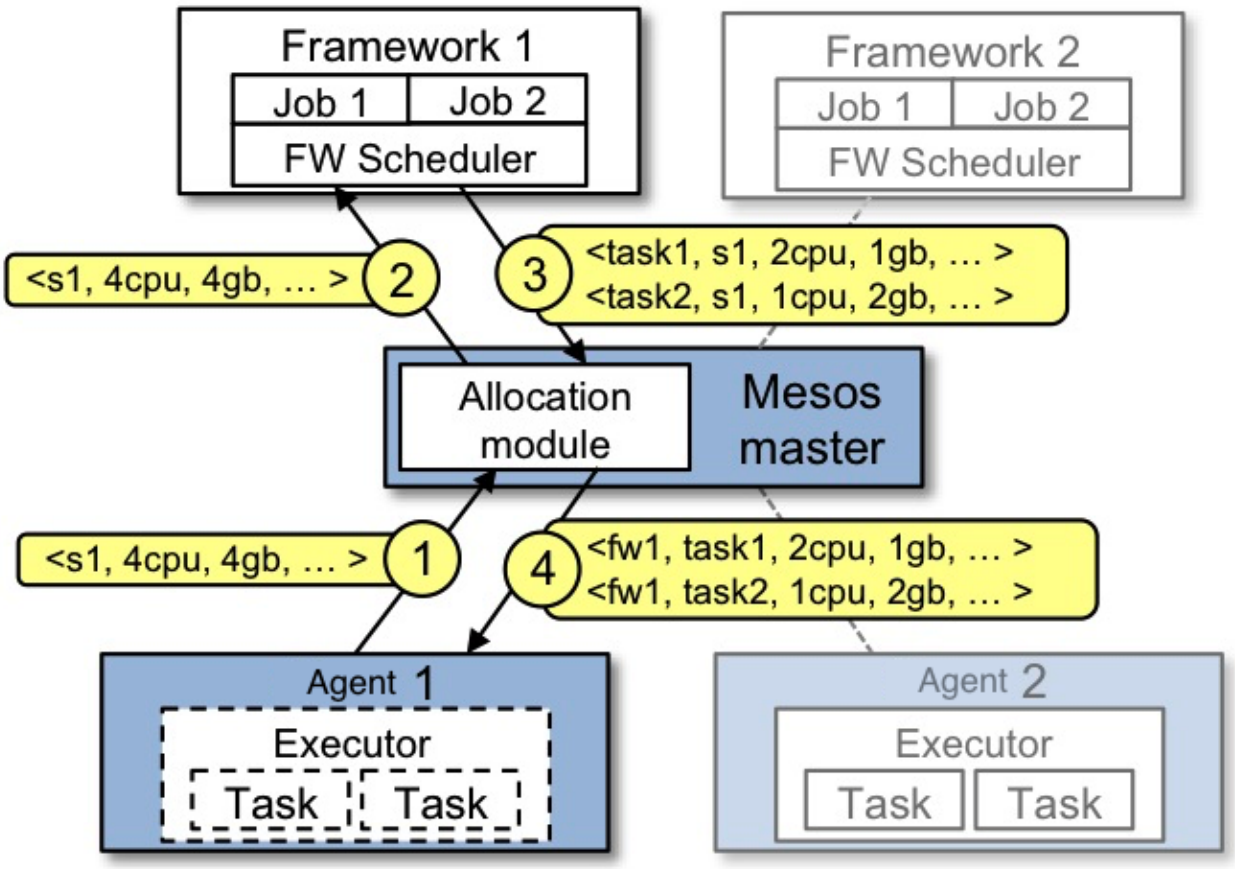

*Apache Mesos Architecture (https://mesos.apache.org)*

According to the above figure, main Mesos components are:

- a **Master Daemon** that manages **agent daemons** running on each cluster nodes, and
- **Mesos Frameworks** which run tasks on these agents. Mesos Framework includes the following components:
    - **Scheduler** that registers with the Master to be offered resources, and

- **Executor Process** that is run on agent nodes to perform the framework's tasks.

The Master provides resource sharing (CPU, RAM, ...) among frameworks by making them **Resource Offers** in a list of *<agent ID, resource1: amount1, resource2: amount2, ...>*. The Master decides how many resources to offer to each framework according to a given organizational policy, such as fair sharing or strict priority. The following figure displays a sample resource offer procedure:




*Apache Mesos - Sample Resouce Offer Procedure (https://mesos.apache.org)*

This is what happens according to the above figure (https://mesos.apache.org/documentation/latest/architecture/):

- Agent 1 reports to the master that it has 4 CPUs and 4 GB of memory free. The master then invokes the allocation policy module, which tells it that framework 1 should be offered all available resources.
- The master sends a resource offer describing what is available on agent 1 to framework 1.
- The framework's scheduler replies to the master with information about two tasks to run on the agent, using <2 CPUs, 1 GB RAM> for the first task, and <1 CPUs, 2 GB RAM> for the second task.
- Finally, the master sends the tasks to the agent, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, the allocation module may now offer them to framework 2.

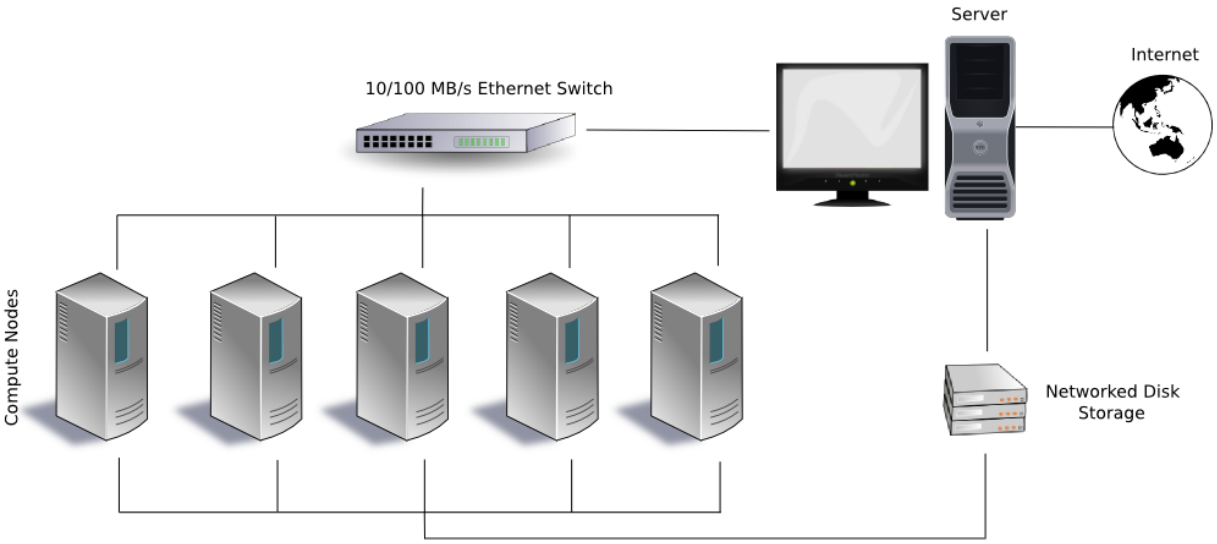
For further readings regarding the Apache Mesos fundamentals, architecture, running Mesos, and advanced features please refer to the following Apache Mesos Documentation link (https://mesos.apache.org/documentation/latest/).

# 6.4.3 Beowulf Cluster

Beowulf Clusters are scalable performance clusters based on commodity hardware, on a private system network, with open source software (Linux) infrastructure (https://en.wikipedia.org/wiki/Beowulf_cluster). Each consists of a cluster of PCs or workstations assigned to perform high-performance computing tasks. The nodes in the cluster don't sit on people's desks; they are dedicated to running cluster jobs. It is usually connected to the outside world through only a single node. As a result, one may have a high-performance parallel computing cluster built upon inexpensive PC hardware. The following figure illustrates one sample Beowulf Cluster:

*A Sample Beowulf Cluster (http://www.cs.mtu.edu)*

The following figure also depicts a typical architecture of a Beowulf cluster configuration:




*A Typical Beowulf Cluster Architecture (https://upload.wikimedia.org)*

# 6.5 Investigating key cluster performance metrics

It is critical for every cluster environment to perform computational tasks safe and sound! It means that they need to follow strict standards in guaranteeing the best quality of service (QoS). In this section, we investigate some well-known cluster performance metrics.

Benchmarking is an extremely important aspect of properly managing and configuring compute and server clusters. Knowledge about how your cluster is performing provides the foundation of any future tweaks and reconfigurations. It's also very useful to know exactly what a cluster is capable of.

Clusters are an excellent way to ensure that you are getting the maximum reliability and scalability out of your hardware, but clusters also bring with them a degree of complexity that is difficult to handle without hard data: bottlenecks can occur that will degrade the overall performance of a cluster, so, being able to analyze each component provides information that's necessary for getting the optimal performance.

Please note that the following metrics are concerned with the performance factors of a given Beowulf cluster. There are myriad of other useful techniques some of which can be found in this interesting article (http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench/) (including TESTDFSIO, TeraSoft Benchmark Suite, NameNode Benchmark, MapReduce Benchmark, ...).

To investigate the performance of a Beowulf cluster, the following metrics have shown to be useful (https://mitpress.mit.edu/books/beowulf-cluster-computing-linux-0):

- **Theoretical Peak Performance** (Maximum aggregate performance or Maximum aggregate floating-point operations) - that is the system's maximum aggregate performance focusing on the maximum aggregate floating-point operations performed per second and is defined as:

```
P = N*C*F*R
```

Where **P** is the indicator for Performance (in Mflops or Gflops), **N** refers to the number of Nodes, **C** is referred to as the number of CPUs per node, and **F** is the number of floating-point operations per CPU clock which is represented as **R**.

- **Application Performance** - is represented as

```
AP = AO/RT
```

Where **AP** indicates the application performance and it is equal to the number of performed operations throughout application execution (**AO**) divided by the total runtime (**RT**).

- **Application Run-Time** - refers to the total amount of time required to perform a given application in terms of CPU clocks.
- **Scalability** - which is represented as

```
S = T(1)/T(N)
```

where **S** is the Scalability metric, **T(1)** is the run-time of a given program on a single processor and **T(N)** is the run-time on N processors.

- **Parallel Efficiency** - which is defined as

  ```
  P(N)/N
  ```

  and the near one value means close to an ideal situation.

- **Percentage of Peak** - is the statistics in terms of the Theoretical Peak Performance (mentioned in number 1 above) and is useful in situations where we want to measure the extent to which an application uses the system's ultimate computational power.
- **Latency and Bandwidth** - **Latency** is referred to the time delay and **Bandwidth** indicates the transfer rate in an inter-processor communications network.
- **System Utilization** - which monitors the system's performance in terms of a long-term throughput basis. This way, we not only consider the CPU-intensive tasks, but we also focus on the I/O-intensive tasks as well.

To test cluster performance considering the above-mentioned factors, several tools have been developed. The most popular ones are The **Ping-Pong Test** **(https://www.mcs.anl.gov/~thakur/papers/thread-tests.pdf)** **(and this** **(https://www.ibm.com/support/knowledgecenter/en/SSF4ZA_9.1.3/pmpi_guide/cluster_test_tools.html))**, **The LINPACK Benchmark** **(https://en.wikipedia.org/wiki/LINPACK_benchmarks)**, and **The NAS Parallel Benchmark Suite** **(https://www.nas.nasa.gov/publications/npb.html)**.

For further readings on these tests as well as other cluster performance metrics, please refer to this book (https://mitpress.mit.edu/books/beowulf-cluster-computing-linux-0).