

# INT 404R01 BIG DATA ANALYTICS

B.Tech      CSE      'A'

Year/Sem: IV/VII

Unit 1 Getting Started with Big Data

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing  
SASTRA University

# DEFINING BIG DATA

- Big data is not a single technology
- The combination of old and new technologies helps companies gain actionable insights.
- Big data has the following 7 characteristics:
  1. volume
  2. velocity
  3. variety
  4. variability
  5. veracity
  6. value
  7. visualization

# **DEFINING BIG DATA**

- **Volume** – How much data
- **Velocity** – How fast the data is generated and processed
- **Variety** – The various types of data
- **Variability**– Big data contains noisy and incomplete data that may obscure valuable insights
- **Veracity**- How accurate is that data in predicting business value
- **Value** – A successful big data analytics must
  - generate value(Meaningful insights)
  - visualization** – Presenting the analysed data in a visually comprehensible manner

# **DEFINING BIG DATA**

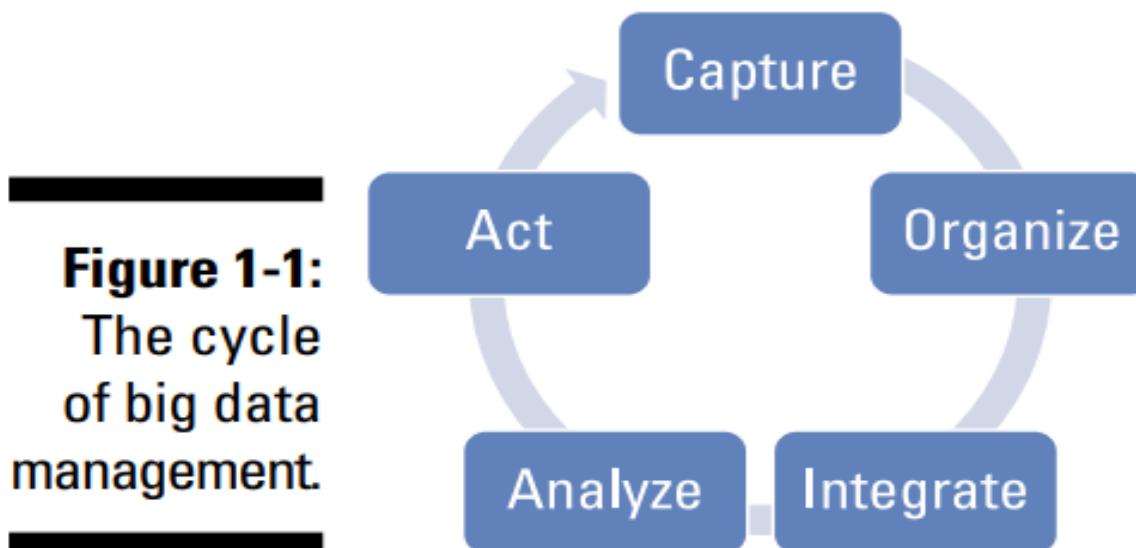
- An innovative business may want to be able to analyze massive amount of data in real time.
- Big data incorporates all data including structured and unstructured data from email, social media, text streams etc. So, companies leverage both types of data.

# BUILDING A SUCCESSFUL BIG DATA MANAGEMENT ARCHITECTURE

- As data has become the fuel of growth and innovation , we need an underlying architecture to support the growing requirements.

Beginning with capture, organize, integrate, analyze and act

Before we delve in to the architecture, lets understand the functional requirements of big data



## BUILDING A SUCCESSFUL BIG DATA MANAGEMENT ARCHITECTURE

- The above figure 1.1 illustrates that data must be captured first and then organized and integrated.
- After this phase is successfully implemented, data can be analyzed based on the problem being addressed.
- Finally, management takes action based on the outcome of that analysis.
- Eg: Amazon.com might recommend a book by analyzing your past purchases.
- **Certain issues in the above functions are:**
  1. validation is required to check whether these sources make sense when integrated.
  2. Certain sources may contain sensitive information. So, you must implement sufficient levels of security and governace.

## **SETTING THE ARCHITECTURAL FOUNDATION**

- In addition to supporting the functional requirements, it is important to **support the required performance**.
- Your **needs depend on the nature of the analysis**
- The right amount of **computational power and speed** is required.
- Even though some analysis will be done in **real time**, still some data need to be **stored as well**.
- Your architecture also requires the **right amount of redundancy** so that we can manage the unanticipated failure of some nodes.

# SETTING THE ARCHITECTURAL FOUNDATION

So, start out by asking yourself the following questions:

1. How much data will my organization need to manage today and in the future?
2. How often my organization need to manage data in real time or near real time?
3. How much risk can be afforded?
4. How important is speed to manage data?
5. How precise does the data need to be?

A big data management architecture must include a variety of services that enable companies to make use of the numerous data sources in a fast and effective manner

# SETTING THE ARCHITECTURAL FOUNDATION

## Big Data Tech Stack



# Interface and feeds

- Before we get into the nitty-gritty of the big data technology stack itself, we'd like you to notice that on either side of the diagram are indications of interfaces and feeds into and out of both internally managed data and data feeds from external sources.
- In fact, what makes big data big is the fact that it relies on picking up lots of data from lots of sources.
- Therefore, open application programming interfaces (**APIs**) will be core to any big data architecture.
- In addition, keep in mind that **interfaces exist at every level and between every layer of the stack.** Without integration services, big data cannot happen.

## **Redundant Physical Infrastructure**

The supporting physical infrastructure is fundamental to the operation and scalability of a big data architecture.

In fact, without the availability of **robust physical infrastructures**, big data would probably not have emerged as such an important trend.

To support an **unanticipated or unpredictable volume of data**, a physical infrastructure for big data has to be different than that for traditional data.

The **physical infrastructure** is **based on a distributed computing model**.

This means that data may be **physically stored in many different locations** and can be **linked together** through networks, the use of a **distributed file system**, and various big data analytic tools and applications.

# Redundant Physical Infrastructure

Redundancy is important because we are dealing with so much data from so many different sources. Redundancy comes in many forms.

If your company has created a private cloud, you will want to have redundancy built within the private environment so that it can scale out to support changing workloads.

It may use external cloud services to augment its internal resources.

In some cases, this redundancy may come in the form of a Software as a Service (SaaS) offering that allows companies to do sophisticated data analysis as a service.

The SaaS approach offers lower costs, quicker startup, and seamless evolution of the underlying technology.

## Security Infrastructure

The more important big data analysis becomes to companies, the more important it will be to secure that data.

For example, if you are a **healthcare company**, you will probably want to use big data applications to determine **changes in demographics** or **shifts in patient needs**.

You will need to take into account **who is allowed to see the data** and under **what circumstances they are allowed** to do so.

You will need to be able to **verify the identity of users** as well as **protect the identity of patients**.

These types of **security requirements** need to be **part of the big data fabric** from the outset and not afterthought

# Operational Data Sources

When you think about big data, it is important to understand that you have to incorporate all the data sources that will give you a complete picture of your business and see how the data impacts the way you operate your business.

Traditionally, an operational data source consisted of highly structured data managed by the line of business in a relational database.

But as the world changes, it is important to understand that operational data now has to encompass a broader set of data sources, including unstructured sources such as customer and social media data in all its forms.

You find new emerging approaches to data management in the big data world, including document, graph, columnar, and geospatial database architectures. Collectively, these are referred to as NoSQL, or not only SQL databases.

# Operational Data Sources

In essence, you need to map the data architectures to the types of transactions.

You also need data architectures that support complex unstructured content.

You need to include both relational databases and non-relational databases in your approach to harnessing big data.

It is also necessary to include unstructured data sources, such as content management systems, so that you can get closer to that 360-degree business view.

# Operational Data Sources

All these operational data sources have several characteristics in common:

- They represent systems of record that **keep track of the critical data required for real-time, day-to-day operation of the business.**
- They are **continually updated** based on **transactions happening** within business units and from the web.
- For these sources to provide an accurate representation of the business, **they must blend structured and unstructured data.**
- These systems also **must be able to scale to support thousands of users on a consistent basis.**

# Performance matters

Your data architecture also needs to perform in concert with your organization's supporting infrastructure.

For example, you might be interested in running models to determine whether it is safe to drill for oil in an offshore area given real-time data of temperature, salinity, sediment resuspension, and a host of other biological, chemical, and physical properties of the water column.

It might take days to run this model using a traditional server configuration.

However, using a distributed computing model, what took days might now take minutes.

## **Performance matters**

Performance might also determine the kind of database you would use.

For example, in some situations, you may want to understand how two very distinct data elements are related.

What is the relationship between buzz on a social network and the growth in sales? This is not the typical query you could ask of a structured, relational database.

A graphing database might be a better choice, as it is specifically designed to separate the “nodes” or entities from its “properties” or the information that defines that entity, and the “edge” or relationship between nodes and properties. Using the right database will also improve performance. Typically the graph database will be used in scientific and technical applications.

# Performance matters: organizing data

## services and tools

- Not all the data that organizations use is **operational**. A growing amount of data comes from a variety of sources that aren't quite as organized or straightforward, including data that comes from **machines or sensors**, and massive public and private data sources.
- In the past, most companies weren't able to either capture or store this vast amount of data. It was simply **too expensive** or too overwhelming.
- Even if companies were able to capture the data, **they did not have the tools** to do anything about it. Very few tools could make sense of these vast amounts of data.
- The tools that did exist were **complex** to use and did not produce results in a reasonable time frame.
- In the end, those who really wanted to go to the enormous effort of analyzing this data were forced to **work with snapshots of data**. This has the undesirable effect of missing important events because they were not in a particular snapshot.

## Performance matters:

### *MapReduce, Hadoop, and BigTable*

with the evolution of computing technology, it is now possible to manage immense volumes of data that previously could have only been handled by supercomputers at great expense.

Prices of systems have dropped, and as a result, new techniques for distributed computing are mainstream.

The real breakthrough in big data happened as companies like Yahoo!, Google, and Facebook came to the realization that they needed to find new technologies that would allow them to store, access, and analyze huge amounts of data in near real time, so that they could monetize the benefits of owning this much data about participants in their networks.

In particular, the innovations MapReduce, Hadoop, and Big Table proved to be the sparks that led to a new generation of data management. These technologies address one of the most fundamental problems – the capability to process massive amounts of data efficiently, cost effectively, and

# Performance matters:

## *MapReduce, Hadoop, and BigTable*

### MapReduce

**MapReduce** was designed by Google as a way of efficiently **executing a set of functions** against a **large amount of data** in **batch mode**.

The “map” component **distributes** the programming problem or **tasks** across a **large number of systems** and handles the placement of the tasks in a way that **balances the load** and **manages recovery from failures**.

After the distributed computation is completed, another function called “**reduce**” **aggregates all the elements back together to provide a result**.

An example of MapReduce usage would be to determine how many pages of a book are

# Performance matters:

## *MapReduce, Hadoop, and BigTable*

### BigTable

Big Table was developed by Google to be a **distributed storage system** intended to manage **highly scalable structured data**.

Data is organized into tables with rows and columns.

Unlike a traditional relational database model, **Big Table** is a **sparse, distributed, persistent multidimensional sorted map**. It is intended to **store huge volumes of data across commodity servers**.

### Hadoop

Hadoop is an **open source framework based on java** that efficiently **stores** and **manages** massive amounts of datasets.

Instead of dumping all the data in to a single machine, **hadoop divides the larger dataset into chunks and stores each chunk in a separate node** and facilitates all the tasks to be carried out in parallel at the same time.

# Traditional and advanced analytics

What does your business now do with all the data in all its forms to try to make sense of it for the business?

It requires many different approaches to analysis, depending on the problem being solved.

Some analyses will use a **traditional data warehouse**, while other analyses will take advantage of **advanced predictive analytics**. Managing big data holistically requires many different approaches to help the business to successfully plan for the future.

## *Analytical data warehouses and datamarts*

After a company sorts through the massive amounts of data available, it is often pragmatic to take the subset of data that reveals patterns and put it into a form that's available to the business.

These warehouses and marts provide compression, multilevel partitioning and a massively parallel

# Traditional and advanced analytics

## *Big data analytics*

The capability to manage and analyze petabytes of data enables companies to deal with clusters of information that could have an impact on the business.

This requires analytical engines that can manage this highly distributed data and provide results that can be optimized to solve a business problem.

Analytics can get quite complex with big data. For example, some organizations are using predictive models that couple structured and unstructured data together to predict fraud.

Social media analytics, text analytics, and new kinds of analytics are being utilized by organizations looking to gain insight into big data.

# Traditional and advanced analytics

## *Reporting and visualization*

Organizations have always relied on the capability to create reports to give them an understanding of what the data tells them about everything from monthly sales figures to projections of growth.

Bigdata changes the way that data is managed and used.

If a company can collect, manage, and analyze enough data, it can use a new generation of tools to help management truly understand the impact not just of a collection of data elements but also how these data elements offer context based on the business problem being addressed.

with big data, reporting and data visualization become tools for looking at the context of how data is related and the impact of those relationships on the future.

# Traditional and advanced analytics

## *Big data applications*

Traditionally, the business expected that data would be used to answer questions about what to do and when to do it.

**with the advent of big data, this is changing.** Now, we are seeing the development of applications that are designed specifically to take advantage of the unique characteristics of big data.

Some of the emerging applications are in areas such as healthcare, manufacturing management, traffic management, and so on. **what do all these big data applications have in common? They rely on huge volumes, velocities, and varieties of data to transform the behavior of a market.**

In **healthcare**, a big data application might be able to monitor premature infants to determine when data indicates when intervention is needed.

In **manufacturing**, a big data application can be used to prevent a machine from shutting down during a production run.

# Traditional and advanced analytics

## *Big data applications*

Traditionally, the business expected that data would be used to answer questions about what to do and when to do it.

Data was often integrated as fields into general-purpose business applications.

With the advent of big data, this is changing. Now, we are seeing the development of applications that are designed specifically to take advantage of the unique characteristics of big data.

Some of the emerging applications are in areas such as healthcare, manufacturing management, traffic management, and so on. What do all these big data applications have in common? They rely on huge volumes, velocities, and varieties of data to transform the behavior of a market.

In healthcare, a big data application might be able to monitor premature infants to determine when data indicates when intervention is needed.

# INT 404R01 BIG DATA ANALYTICS

B.Tech      CSE      'A'

Year/Sem: IV/VII

Unit 1 An Operating System for Big Data

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

# An Operating System for Big Data

## *Chapter 2, An Operating System for Big Data*

Here we provide an overview of the core concepts behind Hadoop and what makes cluster computing both beneficial and difficult. The Hadoop architecture is discussed in detail with a focus on both YARN and HDFS. Finally, this chapter discusses interacting with the distributed storage system in preparation for performing analytics on large datasets.

# An Operating System for Big Data

Data teams are usually structured as small teams of five to seven members who employ a *hypothesis-driven* workflow using agile methodologies.

Data teams therefore are composed of members who fit into three broad categories:

***Data engineers*** are responsible for the practical aspects of the wiring and mechanics of data, usually relating to software and computing resources;

***Data modelers*** focus on the exploration and explanation of data and creating inferential or predictive data products;

***Subjectmatter experts*** provide domain knowledge to problem solving both in terms of process and application.

# An Operating System for Big Data

Data teams that utilize Hadoop tend to place a **primary emphasis** on the **data engineering aspects** of data science due to the technical nature of distributed computing.

Big datasets tend themselves to **aggregation-based approaches** (over instance-based approaches) and a **large toolset** for distributed machine learning and statistical analyses exists already.

For this reason, most literature about Hadoop is targeted at software developers, who usually specialize in Java—the software language the Hadoop API is written in.

# An Operating System for Big Data

In this chapter, we present Hadoop as an operating system for big data.

We discuss the high-level concepts of how the operating system works via its two primary components:

1. The distributed file system, HDFS (“Hadoop Distributed File System”), and
2. workload and resource manager, YARN (“Yet Another Resource Negotiator”).

# An operating System for Big Data – Basic Concepts

## Basic Concepts

- In order to perform computation at scale, Hadoop distributes an analytical computation that involves a massive dataset to many machines that each simultaneously operate on their own individual chunk of data.
- Distributed computing is not new, but it is a technical challenge, requiring distributed algorithms to be developed, machines in the cluster to be managed, and networking and architecture details to be solved.

## Basic Concepts

More specifically, a distributed system must meet the following requirements:

### Fault tolerance

If a component fails, it should not result in the failure of the entire system. The system should gracefully degrade into a lower performing state. If a failed component recovers, it should be able to rejoin the system.

### Recoverability

In the event of failure, no data should be lost.

### Consistency

The failure of one job or task should not affect the final result.

### Scalability

Adding load (more data, more computation) leads to a decline in performance, not

# An operating System for Big Data – Basic Concepts

Hadoop addresses these requirements through several abstract concepts, as defined in the following list :

1. **Data is distributed immediately when added to the cluster** and stored on multiple nodes. Nodes prefer to process data that is **stored locally** in order to **minimize traffic across the network**.

2. **Data is stored in blocks of a fixed size** (usually 128 MB) and each block is **duplicated multiple times** across the system to provide redundancy and data safety.

3. **A computation is usually referred to as a job**; jobs are broken into tasks where each individual node performs the task on a

# An operating system for big data -

## Basic Concepts

4. Jobs are written at a high level without concern for network programming, time, or low-level infrastructure, allowing developers to focus on the data and computation rather than distributed programming details.

5. The amount of network traffic between nodes should be minimized transparently by the system.

Each task should be independent and nodes should not have to communicate with each other during processing to ensure that there are no interprocess dependencies that could lead to deadlock.

6. Jobs are fault tolerant usually through task redundancy, such that if a single node or task fails, the final computation is not

## All operating System for Big Data – Basic Concepts

7. Master programs allocate work to worker nodes such that many worker nodes can operate in parallel, each on their own portion of the larger dataset.

These basic concepts, while implemented slightly differently for various Hadoop systems, drive the core architecture and together ensure that the requirements for fault tolerance, recoverability, consistency, and scalability are met.

These requirements also ensure that Hadoop is a data management system that behaves as expected for analytical data processing, which has traditionally been performed in relational databases or scientific data warehouses.

# An operating system for big data -

## Basic Concepts

Unlike data warehouses, however, Hadoop is able to run on **more economical, commercial off-the-shelf hardware**.

As such, Hadoop has been leveraged primarily to store and compute upon **large, heterogeneous datasets stored in “lakes”** rather than warehouses, and relied upon for rapid analysis and prototyping of data products

# Hadoop Architecture

Hadoop is composed of two primary components that implement the basic concepts of distributed storage and computation as discussed in the previous section: HDFS and YARN.

HDFS (sometimes shortened to DFS) is the Hadoop Distributed File System, responsible for managing data stored on disks across the cluster.

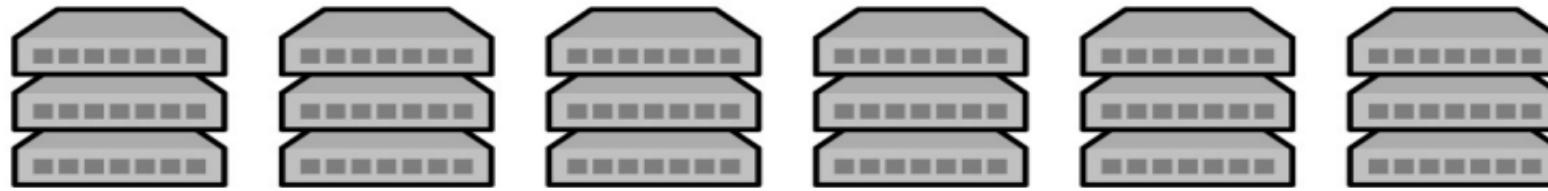
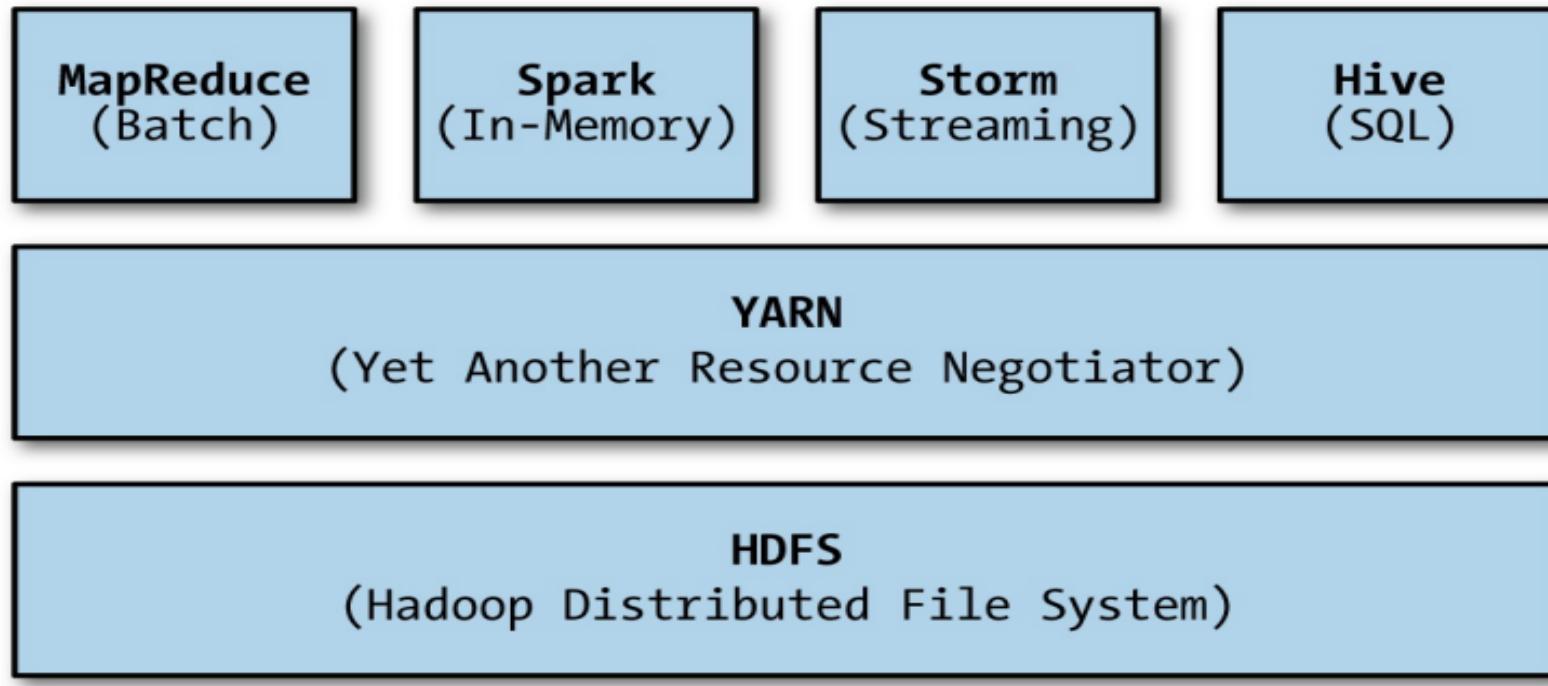
YARN acts as a cluster resource manager, allocating computational assets (processing availability and memory on worker nodes) to applications that wish to perform a distributed computation.

# Hadoop Architecture

The architectural stack is shown in Figure 2-1.

of note, the original MapReduce application is now implemented on top of YARN as well as other new distributed computation applications like the graph processing engine Apache Giraph, and the in-memory computing platform Apache Spark

# Hadoop Architecture



Cluster of economy disks and processors

Figure 2-1. Hadoop is made up of HDFS and

# Hadoop Architecture

HDFS and YARN work in concert to minimize the amount of network traffic in the cluster primarily by ensuring that data is local to the required computation.

Duplication of both data and tasks ensures fault tolerance, recoverability, and consistency.

Moreover, the cluster is centrally managed to provide scalability and to abstract low-level clustering programming details.

Together, HDFS and YARN are a platform upon which big data applications are built; perhaps more than just a platform, they provide an operating system for big data.

# Hadoop Architecture

HDFS and YARN are **flexible**.  
other data storage systems aside from HDFS  
can be integrated into the Hadoop framework  
such as Amazons3 or Cassandra.

Alternatively, data storage systems can be  
built directly on top of HDFS to provide  
more features than a simple file system.

For example, HBase is a columnar data store  
built on top of HDFS and is one the most  
advanced analytical applications that  
leverage distributed storage.

YARN allows richer abstractions of the  
cluster utility, making **new data processing**  
**applications for machine learning, graph**  
**analysis, SQL-like querying of data**, or even  
**streaming data services faster** and more

## **Hadoop Architecture**

Hadoop is observed as a cluster of machines that operate in a coordinated fashion; however, Hadoop is **not hardware** that you have to purchase or maintain.

Hadoop is **actually the name of the software that runs on a cluster**— namely, the distributed file system, HDFS, and the cluster resource manager, YARN, which are collectively composed of six types of background services running on a group of machines.

HDFS and YARN **expose an application programming interface** (API) that abstracts developers from low-level cluster administration details.

A set of machines that is running HDFS and

## Hadoop Architecture

A cluster can have a single node, or many thousands of nodes, but all clusters scale horizontally, meaning as you add more nodes, the cluster increases in both capacity and performance in a linear fashion.

**YARN and HDFS are implemented by several daemon processes**—that is, software that runs in the background and does not require user input.

**Hadoop processes are services**, meaning they run all the time on a cluster node and accept input and deliver output through the network, similar to how an HTTP server works.

Each of these processes runs inside of its own Java Virtual Machine (JVM) so each

# Hadoop Architecture

Each node in the cluster is identified by the type of process or processes that it runs:

## Master nodes

These nodes run coordinating services for Hadoop workers and are usually the entry points for user access to the cluster.

## Worker nodes

These nodes are the majority of the computers in the cluster. Worker nodes run services that accept tasks from master nodes—either to store or retrieve data or to run a particular application.

A distributed computation is run by parallelizing the analysis across worker nodes.

# Hadoop Architecture

Both HDFS and YARN have **multiple master services** responsible for coordinating worker services that run on each worker node.

Worker nodes implement both the HDFS and YARN worker services.

For HDFS, the master and worker services are as follows:

## NameNode(Master)

- Stores the **directory tree of the file system**, **file metadata**, and the **locations of each file in the cluster**. Clients wanting to access HDFS must first locate the appropriate storage nodes by requesting information from the NameNode.

# Hadoop Architecture

## SecondaryNameNode(Master)

Performs **housekeeping tasks** and **checkpointing** on behalf of the NameNode.

Despite its name, it is not a backup NameNode.

## DataNode(worker)

Stores and manages HDFS blocks on the local disk.

Reports **health** and **status** of individual data stores back to the NameNode.

# Hadoop Architecture

At a high level, when data is accessed from HDFS, a client application must first make a request to the NameNode to locate the data on disk.

The NameNode will reply with a list of DataNodes that store the data, and the client must then directly request each block of data from the DataNode.

Note that the NameNode does not store data, nor does it pass data from DataNode to client, instead acting like a traffic cop, pointing clients to the correct DataNodes.

# Hadoop Architecture

Similarly, YARN has multiple master services and a worker service as follows:

## **ResourceManager (Master)**

- Allocates and monitors available cluster resources (e.g., physical assets like memory and processor cores) to applications as well as handling scheduling of jobs on the cluster.

## **ApplicationMaster(Master)**

- Coordinates a particular application being run on the cluster as scheduled by the ResourceManager.

# Hadoop Architecture

## NodeManager(worker)

- Runs and manages processing tasks on an individual node as well as reports the health and status of tasks as they're running.

Similar to how HDFS works, clients that wish to execute a job must first request resources from the ResourceManager, which assigns an application-specific ApplicationMaster for the duration of the job.

The ApplicationMaster tracks the execution of the job, while the ResourceManager tracks the status of the nodes, and each individual NodeManager creates containers and executes tasks within them.

# Hadoop Architecture

Note that there may be other processes running on the Hadoop cluster as well—for example, JobHistory servers or ZooKeeper coordinators, but master and worker services are the primary software running in a Hadoop cluster.

Master processes are so important that they usually are run on their own node so they don't compete for resources and present a bottleneck. However, in smaller clusters, the master daemons may all run on a single node.

## Hadoop Architecture

An example deployment of a small Hadoop cluster with six nodes, two master and four worker, is shown in Figure 2-2.

Note that in larger clusters the NameNode and the Secondary NameNode will reside on separate machines so they do not compete for resources.

The size of the cluster should be relative to the size of the expected computation or data storage because clusters scale horizontally.

Typically a cluster of 20-30 worker nodes and a single master is sufficient to run several jobs simultaneously on datasets in the tens of terabytes.

For more significant deployments of hundreds of nodes, each master requires its own machine and in even larger clusters of

# Hadoop Architecture

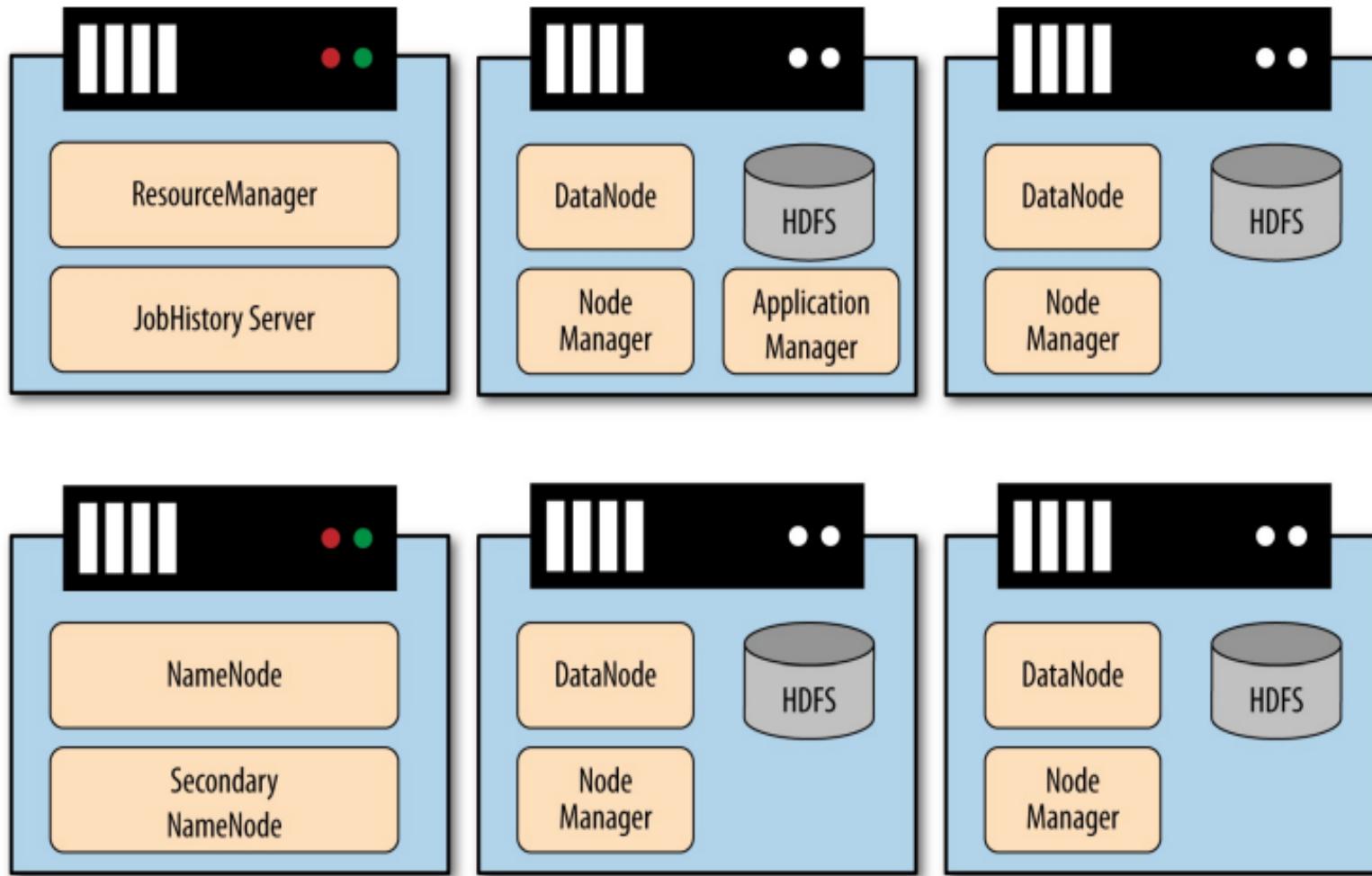


Figure 2-2. A small Hadoop cluster with two master nodes and four workers nodes that implements all six primary Hadoop services

# Hadoop Architecture

Finally, one other type of cluster is important to note: a **single node cluster**.

In “**pseudo-distributed mode**” a single machine runs all Hadoop daemons as though it were part of a cluster, but network traffic occurs through the local loopback network interface.

In this mode, the benefits of a distributed architecture aren’t realized, but it is the perfect setup to develop on without having to worry about administering several machines.

Hadoop developers typically work in a pseudodistributed environment, usually **inside of a virtual machine** to which they connect via SSH. Cloudera, Hortonworks, and

# Hadoop Architecture

## HDFS

HDFS provides redundant storage for big data by storing that data across a cluster of cheap, unreliable computers, thus extending the amount of available storage capacity that a single machine alone might have.

However, because of the networked nature of a distributed file system, HDFS is more complex than traditional file systems.

In principle, HDFS is a software layer on top of a native file system such as ext4 or xfs.

Hadoop generalizes the storage layer

It can interact with local file systems and other storage types like Amazon S3.

# Hadoop Architecture

However, HDFS is the **flagship distributed file system**, and for most programming purposes it will be the **primary file system** you'll be interacting with. HDFS is designed for **storing very large files with streaming data access**, and as such, it comes with a few **caveats(warnings)**:

HDFS performs best with a modest number of very large files—for example, millions of large files (100 MB or more) rather than billions of smaller files that might occupy the same volume.

HDFS implements the WORM pattern—write once, read many. **No random writes or appends to files are allowed.**

# Hadoop Architecture

Therefore, HDFS is **best suited for storing raw input data to computation, intermediary results between computational stages, and final results for the entire job.**

It is not a good fit as a data backend for applications that require updates in real-time, interactive data analysis, or record-based transactional support.

## blocks

HDFS files are **split into blocks**, usually of either 64 MB or 128 MB, although this is configurable at runtime and high-performance systems typically select block sizes of 256 MB.

The block size is the **minimum amount of data that can be read or written** to in HDFS.

# Hadoop Architecture

Larger block sizes help in handling large files efficiently by reducing the overhead of managing numerous smaller blocks.

Each file is split into blocks, and these blocks are stored across multiple nodes in a cluster.

Different blocks from the same file will be stored on different machines to provide for more efficient distributed processing.

By default, each block is replicated three times to ensure fault tolerance. This means if one node fails, the data can still be retrieved from another node.

By default, the replication is three-fold, but this is also configurable at runtime.

Therefore, each block exists on three different machines and three different disks, and if even two nodes fail, the data will not be lost.

# Hadoop Architecture

## Data Management

The master **NameNode** keeps track of what blocks make up a file and where those blocks are located.

The NameNode **communicates with the DataNodes**, the processes that actually hold the blocks in the cluster.

**Metadata associated with each file is stored in the memory of the NameNode master for quick lookups**, and if the NameNode stops or fails, the entire cluster will become **inaccessible!**

The **Secondary NameNode** is not a backup to the **NameNode**, but instead performs housekeeping tasks on behalf of the NameNode, including (and especially) **periodically merging a snapshot of the current data space with the edit log** to ensure that the edit log doesn't get too large.

The edit log is used to **ensure data consistency and prevent data loss**; if the NameNode fails, this information can be used to reconstruct the data.

# Hadoop Architecture

## Data Management contd..

When a client application wants access to read a file, it first requests the metadata from the NameNode to locate the blocks that make up the file, as well as the locations of the DataNodes that store the blocks.

The application then communicates directly with the DataNodes to read the data.

Therefore, the NameNode simply acts like a journal or a lookup table and is **not a bottleneck to simultaneous reads**

# Hadoop Architecture

## YARN

While the original version of Hadoop (Hadoop 1) popularized MapReduce and made large-scale distributed processing accessible to the masses, it only offered MapReduce on HDFS.

This was due to the fact that in Hadoop 1, the MapReduce job/workload management functions were highly coupled to the cluster/resource management functions.

As such, there was no way for other processing models or applications to utilize the cluster infrastructure for other distributed workloads.

# Hadoop Architecture

## YARN

MapReduce can be very efficient for large-scale batch workloads, but it's also quite I/O intensive,

Due to the batch-oriented nature of HDFS and MapReduce, faces significant limitations in support for interactive analysis, graph processing, machine learning, and other memory-intensive algorithms.

while other distributed processing engines have been developed for these particular use cases, the MapReduce-specific nature of Hadoop 1 made it impossible to repurpose the same cluster for these other distributed workloads.

# Hadoop Architecture

## YARN

Hadoop 2 addresses these limitations by introducing YARN, which decouples workload management from resource management so that multiple applications can share a centralized, common resource management service.

By providing generalized job and resource management capabilities in YARN, Hadoop is no longer a singularly focused MapReduce framework but a full-fledged multiapplication, big data operating system.

# INT 404R01 BIG DATA ANALYTICS

B.Tech      CSE      'A'

Year/Sem: IV/VII

Unit 1

TOPIC:WORKING WITH DISTRIBUTED COMPUTATION

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

## **WORKING WITH DISTRIBUTED COMPUTATION**

We need to have a fundamental understanding of distributed computing and its requirements.

While YARN has enabled Hadoop to become a general distributed computing platform, MapReduce (often abbreviated to MR) was the first computational framework for Hadoop.

YARN allows for non-MapReduce frameworks such as Spark, Tez, and Storm (to name a few) to run alongside the original MapReduce application on a Hadoop cluster.

However, for most Hadoop users, MapReduce is still the primary framework for many applications and analytics. Moreover, a general understanding of how MapReduce works allows us to think more deeply about

## WORKING WITH DISTRIBUTED COMPUTATION

In this section, we'll explore the basic principles of the MapReduce programming paradigm and discuss why these functional programming constructs are ideal for distributed systems.

We will demonstrate how MapReduce works via two simple analytics that are routinely used to demonstrate computation in a distributed environment: word counting and shared friendships.

Finally, we will describe

1. How MapReduce applications are implemented on a Hadoop cluster &
2. How to submit and manage a sample MapReduce job, fetching the output via the Hadoop command-line interface

# MapReduce: A Functional Programming Model

MapReduce is a **simple but very powerful computational framework** specifically designed to enable **fault-tolerant distributed computation** across a cluster of centrally managed machines.

It does this by employing a “**functional programming style** that is inherently parallelizable—by allowing multiple independent tasks to execute a function on local chunks of data and aggregating the results after processing.

Functional programming is a **style of programming that ensures computations are evaluated in a stateless manner**. This means functions depend only on their inputs, and they are closed and do not share state.

## MapReduce: A Functional Programming Model

Data is transferred between functions by sending the output of one function as the input to another, wholly independent function.

These traits make functional programming a great fit for distributed, big data computational systems.

Because it allows us to move the computation to any node that has the data input and guarantee that we will still get the same result.

Because functions are stateless and depend solely on their input, many functions on many machines can work independently on smaller chunks of the dataset.

By strategically chaining the outputs of functions to the inputs of other functions.

## MapReduce: A Functional Programming Model

The two functions that **distribute work** and **aggregate results** are called **map** and **reduce**, respectively.

MapReduce utilizes “key/value” pairs to coordinate computation.

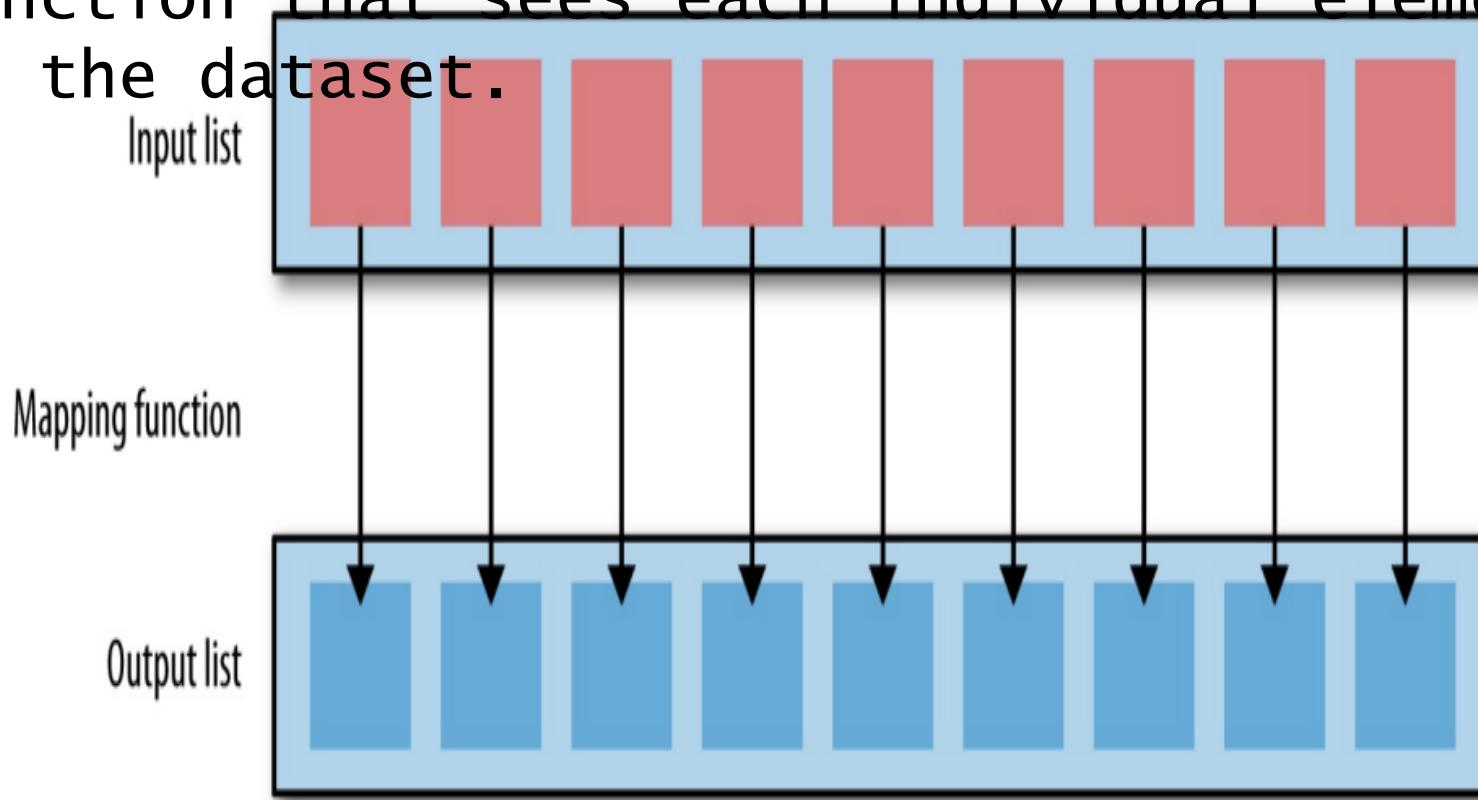
Pseudocode in Python

```
def map(key, value):
    # Perform processing
    return (intermed_key, intermed_value)

def reduce(intermed_key, values):
    # Perform processing
    return (key, output)
```

## MapReduce: A Functional Programming Model

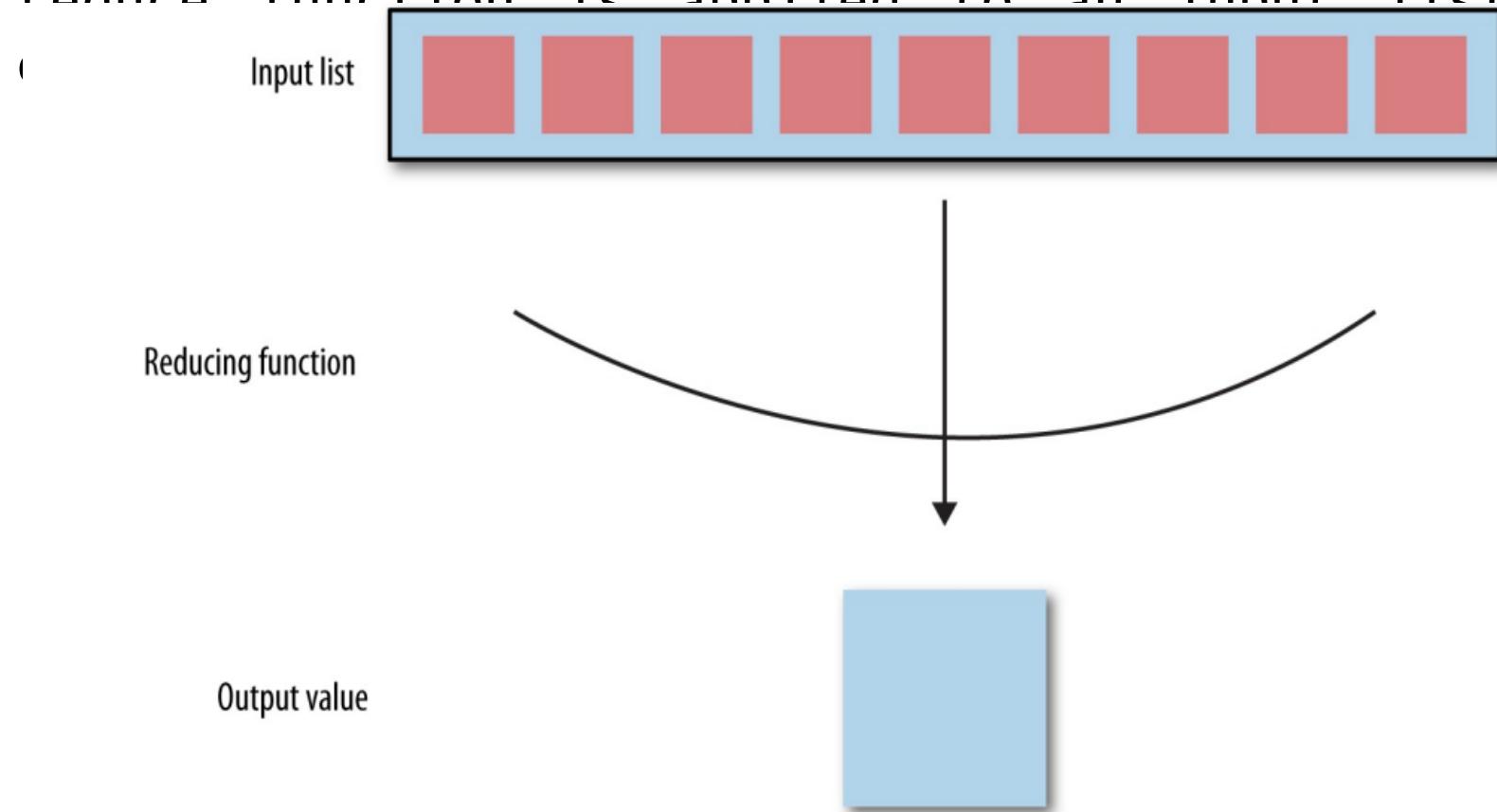
The map operation is where the core analysis or processing takes place, as this is the function that sees each individual element in the dataset.



*Figure 2-3. A map function takes as input a list of key/value pairs and operates singly upon each individual element in the list, outputting zero or more key/value pairs*

# MapReduce: A Functional Programming Model

After the map phase, any emitted key/value pairs will then be **grouped by key** and those key/value groups are applied as input to reduce functions on a per-key basis. As shown in Figure 2-4, a reduce function is applied to an input list to



*Figure 2-4. A reducer takes a key and a list of values as input, operates on the values list as a whole, usually through aggregation operations, and outputs zero or more key/value pairs*

In the MapReduce (MR) framework, `def map(key, value)` is a function that processes input key-value pairs and generates intermediate key-value pairs.

Here's a breakdown of how this works with an illustrative diagram:

- **Explanation**
- **Input Splitting:**
  - The `input` data is split into smaller chunks, each chunk being processed by a separate `map` function.
- **Mapping:**
  - The `map` function takes a `key` and `value` as `input` and processes them to generate intermediate key-value pairs.
  - The `key` is usually an identifier (like a line number in a text file), and the `value` is the actual data (like the text content of the line).

## Shuffling and Sorting:

- The intermediate key-value pairs are shuffled and sorted by the framework. The purpose is to group all values associated with the same key together.

## Reducing:

- The sorted key-value pairs are then passed to the reduce function, which processes them to generate the final output.

## Example: Word Count

Let's consider a **simple word count example** to illustrate this process. The goal is to count the number of occurrences of each word in a set of documents.

### Input

Document 1: "hello world"

Document 2: "hello mapreduce"

- Map Phase
- Input Splitting:
  - The input documents are split into chunks:
    - Chunk 1: ("doc1", "hello world")
    - Chunk 2: ("doc2", "hello mapreduce")

- Mapping:
  - The map function processes each chunk:

```
def map(key, value):  
    for word in value.split():  
        yield (word, 1)
```

Output of the map function: For ("doc1",  
"hello world"):

```
("hello", 1)  
("world", 1)
```

For ("doc2", "hello mapreduce"):

("hello", 1) ("mapreduce", 1)

## Shuffling and Sorting

The intermediate key-value pairs are shuffled and sorted:

("hello", 1), ("hello", 1), ("world", 1),  
("mapreduce", 1)

## Reduce Phase

The reduce function processes the sorted key-value pair

```
def reduce(key, values):
    yield (key, sum(values))
```

Output of the reduce function:

("hello", 2) ("world", 1) ("mapreduce", 1)

```
+-----+  
| Input Data |  
+-----+  
| ("doc1", "hello") |  
| ("doc1", "world") |  
| ("doc2", "hello") |  
| ("doc2", "mapreduce") |  
+-----+  
|  
| v  
+-----+  
| Map Phase |  
+-----+  
| ("hello", 1) |  
| ("world", 1) |  
|  
| ("hello", 1) |  
| ("mapreduce", 1) |  
+-----+  
|  
| v  
+-----+  
| Shuffle & Sort |  
+-----+  
| ("hello", 1), |  
| ("hello", 1), |  
| ("world", 1), |  
| ("mapreduce", 1) |  
+-----+
```

```
|  
| v  
+-----+  
| Reduce Phase |  
+-----+  
| ("hello", [1, 1]) |  
| ("world", [1]) |  
| ("mapreduce", [1]) |  
+-----+  
|  
| v  
+-----+  
| Final Output |  
+-----+  
| ("hello", 2) |  
| ("world", 1) |  
| ("mapreduce", 1) |  
+-----+
```

## **MAPREDUCE:IMPLEMENTED ON A CLUSTER**

Because mappers apply the same function to each element of any arbitrary list of items , they are well suited to distribution across nodes on a cluster.

Each node gets a copy of the mapper operation , and applies the mapper to the key/value pairs that are stored in the blocks of data of the local HDFS data nodes.

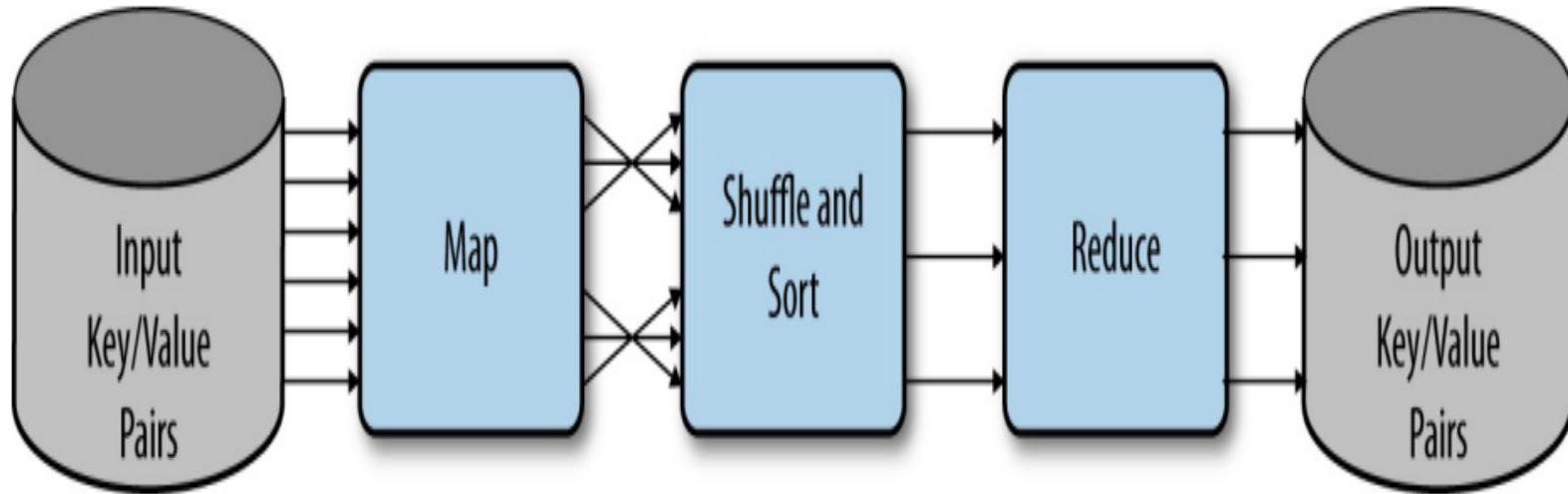
There can be any number of mappers working independently on as much data as possible, really only limited by the number of processors available on the cluster.

Because they are stateless, no network communication between processes is required (or possible). Because mappers are deterministic, their output is not dependent on anything but the incoming values. and

Reducers require as input the output of the mappers on a per-key basis; therefore, reducer computation can also be distributed such that there can be as many reduce operations as there are keys available from the mapper output.

You should correctly expect that each reducer sees all values for a single, unique key. In order to meet this requirement, a shuffle and sort operation is required to coordinate the map and reduce phases, such that reducer input is grouped and sorted by key.

Shuffle and sort partitions the keyspace from the map phase in order to allocate a specific keyspace to specific reducers. Therefore, in broad strokes, the phases of MapReduce are shown in Figure 3-5.



*Figure 2-5. Broadly, MapReduce is implemented as a staged framework where a map phase is coordinated to a reduce phase via an intermediate shuffle and sort*

The phases shown in Figure 2-5 are as follows:

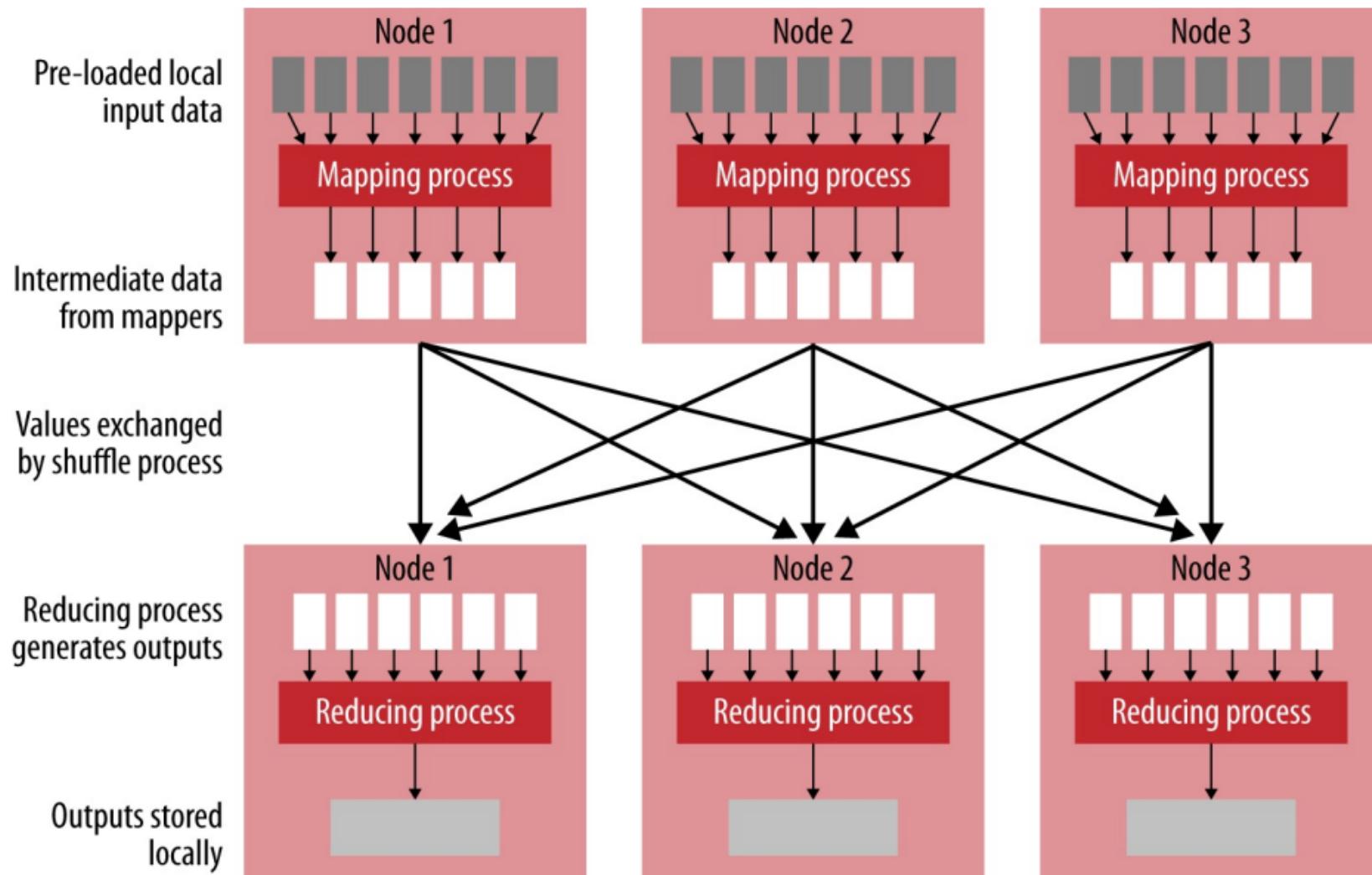
Phase 1 Local data is loaded into a mapping process as key/value pairs from HDFS.

Phase 2 Mappers output zero or more key/value pairs, mapping computed values to a particular key.

Phase 3 These pairs are then sorted and shuffled based on the key and are then passed to a reducer such that all values for a key are available to it.

Phase 4 Reducers then must output zero or more final key/value pairs, which are the output (reducing the results of the map).

# The details of data flow in a MapReduce pipeline executed on a cluster of a few



*Figure 2-6. Data flow of a MapReduce job being executed on a cluster of a few nodes*

A common optimization at this point is to apply a combiner(Shuffle and Sort phase)—a process that aggregates map output for a single mapper.

This prework leads to less work for the reducers and therefore better reducer performance.

The intermediate keys are pulled from the map processes to a partitioner.

The partitioner decides how to allocate the keys to the reducers.

Typically, a uniformly distributed keyspace is assumed, and therefore a hash function is used to evenly divide keys among the reducers. The partitioner also sorts the key/value pairs such that the full “shuffle”

Finally, the reducers start work, pulling an iterator of data for each key and performing a reduce operation such as an aggregation.

Their output key/value pairs are then written back to HDFS using an `OutputFormat` class.

## MAP REDUCE EXAMPLES

In order to demonstrate how data flows through a map and reduce computational pipeline, we will present two concrete examples:

1. word counting and
2. shared friendships.

Both of these applications, while simple, demonstrate how data flows through a distributed system.

Word count in particular is used so commonly to demonstrate distributed computing tasks that it is often referred

The word-counting application takes as **input** one or more **text files** and produces a **list of words and their frequencies as output**.

More specifically, because Hadoop utilizes key/value pairs—the **input key** is a **file ID** and **line number** and the **input value** is a **string**, while the **output key** is a **word** and the **output value** is an **integer**.

Mappers can work on chunks of single documents—the map operation doesn’t care about the context of the words, just that it can count the words it is given as input.

Similarly, we can have multiple reducers working on different keys simultaneously because the output key is a word.

Python pseudocode shows how the algorithm is implemented:

**Note:** `emit` is a function that performs Hadoop I/O similar to `yield` function in python, it sends its arguments to the next phase of the MapReduce pipeline

```
def map(key,value):
    for word in value.split():
        emit(word,1)

def reduce(word,values):
    count=sum(value for value in values)
    emit(word,count)
```

In the diagram in Figure 2-7, we see there are two documents containing two simple sentences.

The map function will receive some unique ID for the text, and a string of the contents of that document.

Its job is to split the value by space and punctuation (getting all the words) and to emit each word as the intermediate key, and the value 1—because the mapper has seen one instance of this word. The data for each mapper is shown here:

```
: # Input to WordCount mappers
```

```
(27183, "The fast cat wears no hat.")
```

```
(31416, "The cat in the hat ran fast.")
```

```
# Mapper 1 output
```

```
("The", 1), ("fast", 1), ("cat", 1),  
 ("wears", 1), ("no", 1),  
 ("hat", 1), (".", 1)
```

```
# Mapper 2 output
```

```
("The", 1), ("cat", 1), ("in", 1), ("the",  
 1), ("hat", 1),  
 ("ran", 1), ("fast", 1), (".", 1)
```

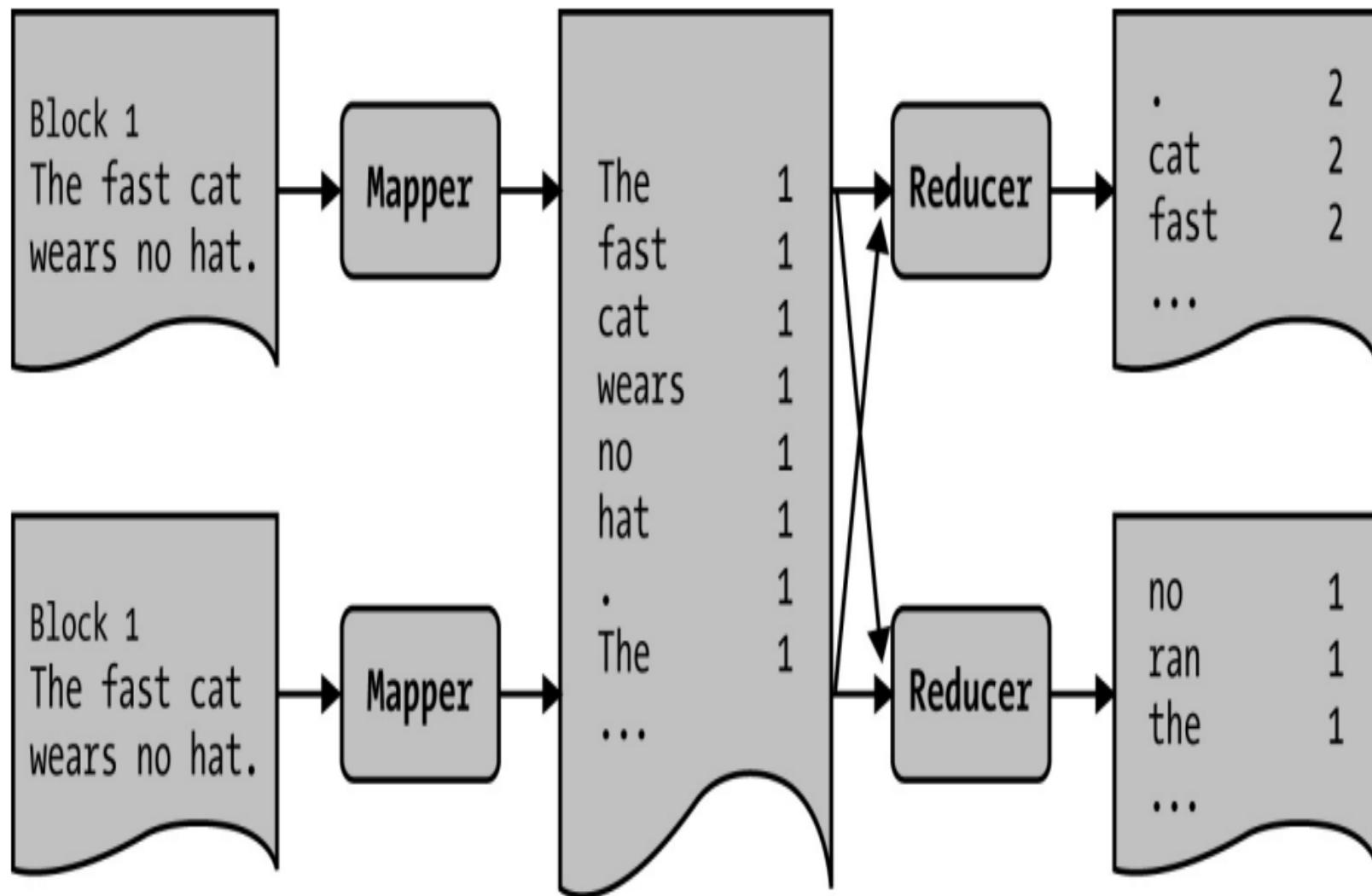


Figure 2-7. Data flow of the word count job being executed on a cluster with two mappers and two reducers

This data is passed to the shuffle and sort phase where the keys (words) are grouped together and sorted and sent to the appropriate reducer.

Each reducer receives as input the word as the key and a list of ones as the values.

In order to get the counts, it simply sums the ones and emits the word as the key and the count as the value.

The data to the input and the output from our exam

```
# Input to WordCount reducers
# This data was computed by shuffle and sort

(".", [1, 1])
("cat", [1, 1])
("fast", [1, 1])
("hat", [1, 1])
("in", [1])
("no", [1])
("ran", [1])
("the", [1])
("wears", [1])
("The", [1, 1])

# Output by all WordCount reducers

(".", 2)
("cat", 2)
("fast", 2)
("hat", 2)
("in", 1)
("no", 1)
("ran", 1)
("the", 1)

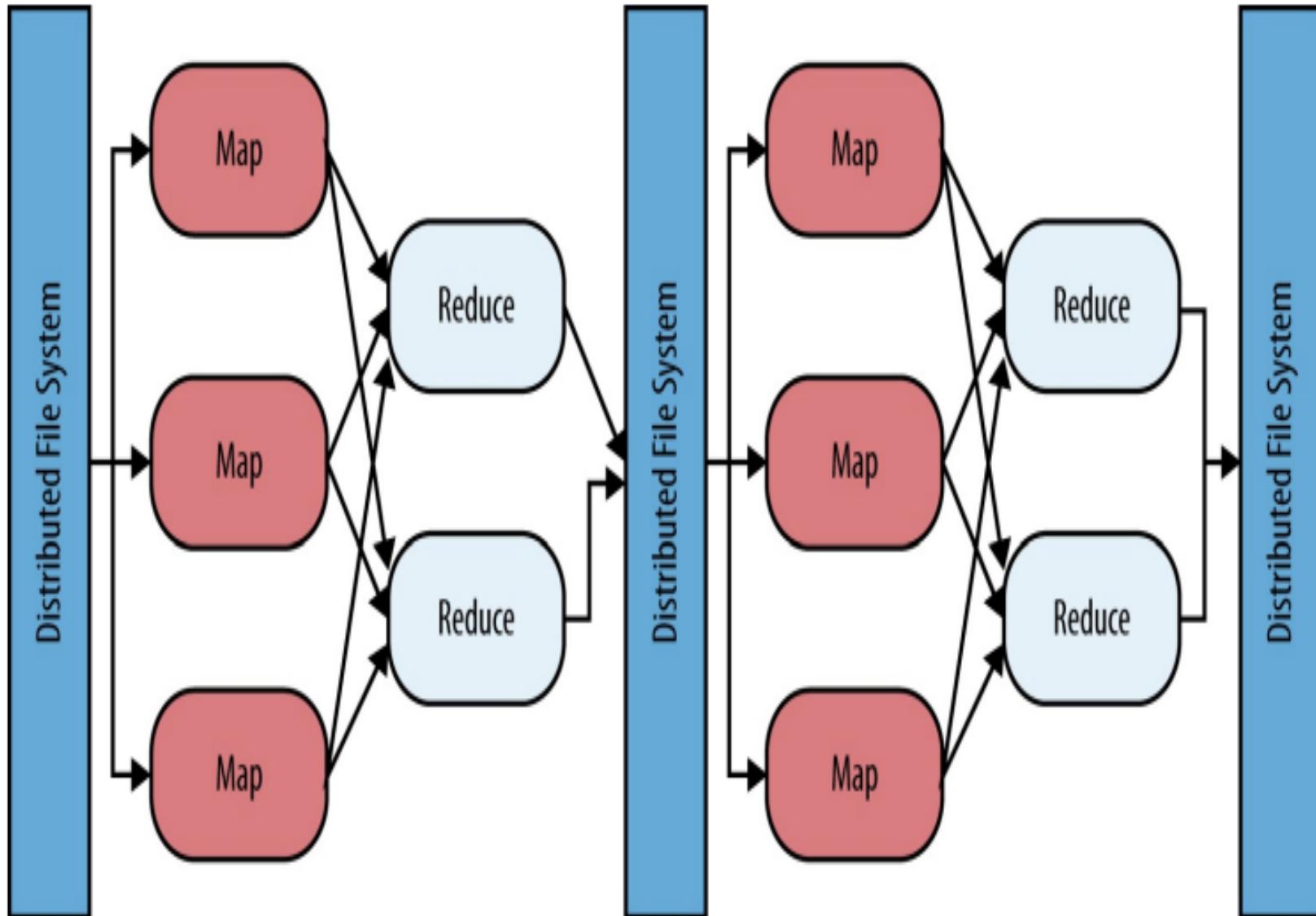
("wears", 1)
("The", 2)
```

## Beyond a Map and Reduce: Job Chaining

In fact, the use of multiple MapReduce jobs to perform a single computation is how more complex applications are constructed, through a process called “job chaining.”

By creating data flows through a system of intermediate MapReduce jobs, as shown in Figure 2-8, we can create a pipeline of analytical steps that lead us to our end result.

As analysts and developers, our job is to devise algorithms that implement map and reduce in order to come to a single analytical conclusion



*Figure 2-8. Complex algorithms or applications are actually made up through the chaining of MapReduce jobs where the input of a downstream MapReduce job is the output of a more recent one*

## SUBMITTING A MAP REDUCE JOB TO YARN

The MapReduce API is written in Java, and therefore MapReduce jobs submitted to the cluster are going to be compiled Java Archive (JAR) files.

Hadoop will transmit the JAR files across the network to each node that will run a task (either a mapper or reducer) and the individual tasks of the MapReduce job are executed.

# SUBMITTING A MAP REDUCE JOB TO YARN

## Prerequisites:

- **Hadoop Installation:** Ensure Hadoop is installed and configured correctly on your cluster.
- **YARN Configuration:** Make sure YARN is properly set up and running on your Hadoop cluster.
- **Job JAR File:** Have your MapReduce job compiled into a JAR file.
- **Steps:**
  - **1. Start YARN**  
Make sure YARN is running. You can start YARN with the following command:

```
$ start-yarn.sh
```

## SUBMITTING A MAP REDUCE JOB TO YARN

To check if YARN is running, use:

```
$ jps
```

You should see ResourceManager and  
NodeManager in the list.

## 2. Prepare Input Data

Place the input data for your job in HDFS  
(Hadoop Distributed File System). For  
example, if your input data is in a local  
directory /local/input, you can copy it to  
HDFS as follows:

```
$ hdfs dfs -mkdir -p /user/hadoop/input
$ hdfs dfs -put /local/input/*
/user/hadoop/input
```

# SUBMITTING A MAP REDUCE JOB TO YARN

## 3. Submit the Job

Use the hadoop command to submit your MapReduce job to YARN. The command format is:

```
$ hadoop jar <path-to-your-jar-file> <main-class> <input-path> <output-path>
```

For example:

```
$ hadoop jar /path/to/your-job.jar  
com.example.YourMainClass /user/hadoop/input  
/user/hadoop/output
```

<path-to-your-jar-file>: Path to your JAR file containing the MapReduce job.

<main-class>: The main class with the main method to be executed.

<input-path>: HDFS directory containing the input data.

# SUBMITTING A MAP REDUCE JOB TO YARN

## 4. Monitor the Job

You can monitor the job's progress through the ResourceManager web UI, typically accessible at:

**http://<resource-manager-host>:8088/**

Here, you can monitor the job's progress,

Application ID:161_01	Application ID:161_02
User: hadoop name:WordCount State: RUNNING(Resources allotted, running)  Progress: 70%	User: Spark name: SparkJob State: ACCEPTED (Resources note yet allotted- ready to be executed)  Progress: 0%

## 5. Retrieve the Output

Once the job completes successfully, you can retrieve the output from HDFS to your local file system if needed. For example:

```
$ hdfs dfs -get /user/hadoop/output /local/output
```

# Example

Start YARN:

```
$ start-yarn.sh
```

Prepare Input Data:

```
$ hdfs dfs -mkdir -p /user/hadoop/input  
$ hdfs dfs -put /local/input/data.txt  
  /user/hadoop/input
```

Submit the Job:

```
$ hadoop jar /path/to/your-job.jar  
  com.example.WordCount /user/hadoop/input  
  /user/hadoop/output
```

## Monitor the Job:

- Open

*<http://<resource-manager-host>:8088/>*

in your browser to monitor the job.

## Retrieve the Output:

*\$ hdfs dfs -get /user/hadoop/output  
/local/output*

# INT 404R01 BIG DATA ANALYTICS

B.Tech      CSE      'A'

Year/Sem: IV/VII

Unit 1

TOPIC: FRAMEWORK FOR PYTHON AND HADOOP STREAMING

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

## FRAMEWORK FOR PYTHON AND HADOOP STREAMING

In this chapter,

We explore the details of :

***How to use Hadoop Streaming***

***Work through the creation of a small framework that will allow us to quickly write MapReduce jobs using Python.***

We will ***extend the simple WordCount program*** to actually use third-party libraries in Python for natural language processing (NLP), and

write a ***MapReduce job that identifies the frequencies of significant bigrams in text.***

Finally we will look at ***some advanced MapReduce topics*** that are essential to understanding Hadoop, and how to apply these topics to Streaming jobs written in Python.

## Hadoop Streaming

Hadoop Streaming is a utility, packaged as a JAR file that comes with the Hadoop MapReduce distribution.

Streaming is used as a normal Hadoop job passed to the cluster via the job client, but allows you to also specify arguments such as the input and output HDFS paths, along with the mapper and reducer executable.

The job is then run as a normal MapReduce job, managed and monitored by the ResourceManager.

## Hadoop Streaming

In order to perform a MapReduce job, **Streaming** utilizes the standard Unix streams for input and output, hence the name Streaming.

**Input to both mappers and reducers is read from stdin, which a Python process can access via the sys module.**

Hadoop expects the Python mappers and reducers to write their output key/value pairs to stdout.

Figure 3-1 demonstrates this process in a MapReduce context.

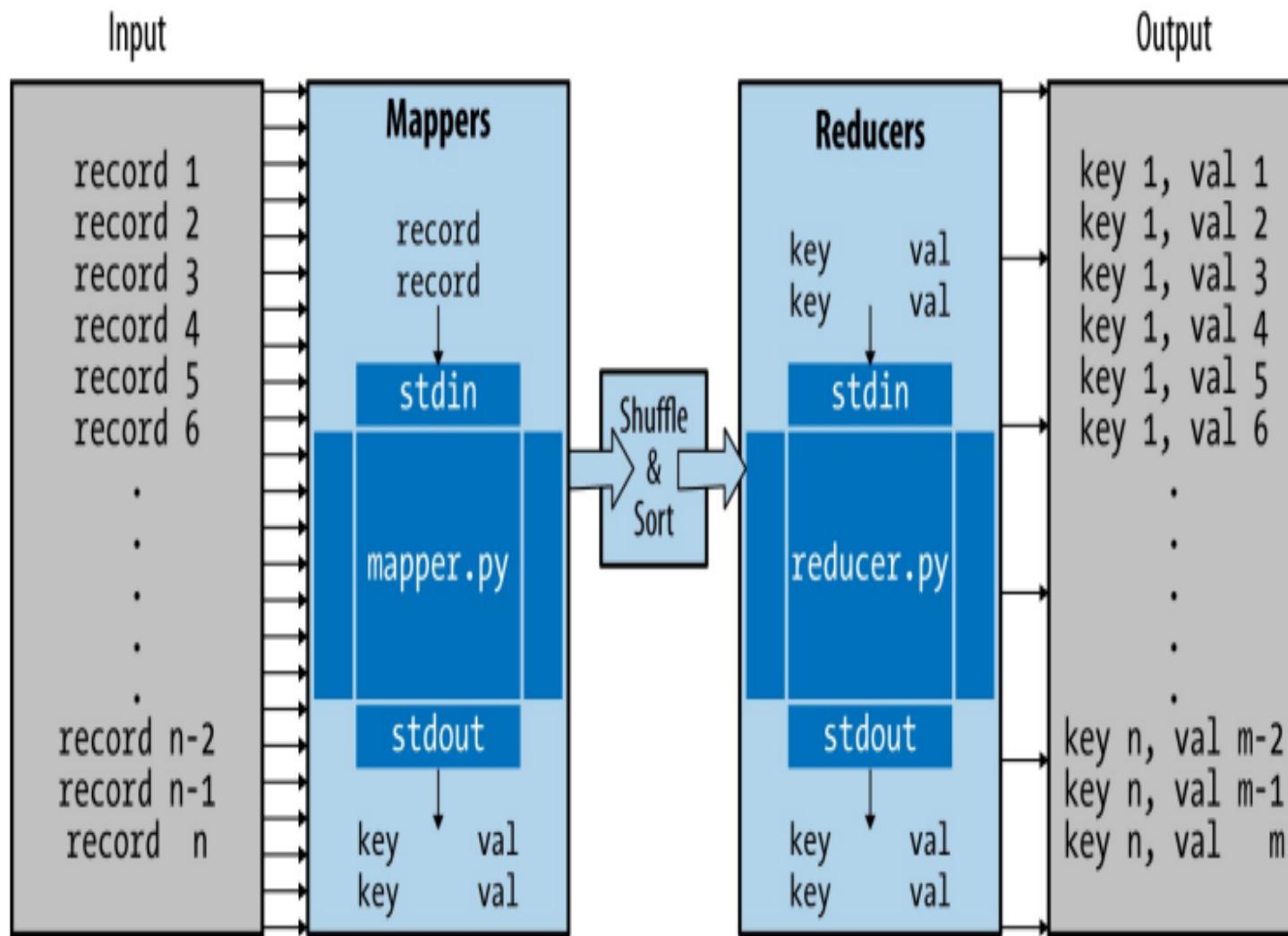


Figure 3-1. Data flow in Hadoop Streaming via Python `mapper.py` and `reducer.py` scripts

when Streaming executes a job, each mapper task will launch the supplied executable inside of its own process.

The mapper then converts the input data into lines of text and pipes it to the stdin of the external process while simultaneously collecting output from stdout.

The input conversion is usually a straightforward serialization of the value because data is being read from HDFS, where each line is a new value.

The mapper expects output to be in a string key/value format, where the key is separated from the value by some separator character, tab (\t) by default.

The reducer is also launched as its own executable after the output from the mappers is shuffled and sorted to ensure that each key is sent to the same reducer.

The key/value output strings from the mapper are streamed to the reducer as input via `stdin`, matching the data output from the mapper, and guaranteed to be grouped by key.

The output the reducer emits to `stdout` is expected to have the same key, separator, and value format as the mapper.

Therefore, in order to write Hadoop jobs using Python, we need to create two Python files, `mapper.py` and a `reducer.py`.

First, we create our executable mapper in a file called *mapper.py*:

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

import sys

if __name__ == "__main__":
    for line in sys.stdin:
        for word in line.split():
            sys.stdout.write("{}\t1\n".format(word))
```

The mapper simply reads each line from `sys.stdin`, splits on space, then writes each word and a 1 separated by a tab, line-by-line to `sys.stdout`.

The reducer is a bit more complex because we have to track which key we're on at every line of input, and only emit a completed sum when we see a new key.

This is because, unlike the native API, individual data values are aggregated to the streaming process during shuffle and sort rather than exposed as a list or iterator.

Keep in mind that each reducer task is guaranteed to see all values for the same key, but may also see multiple keys.

In a file called reducer.py, we implement the reducer executable as follows:

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

```
import sys
```

```
if __name__ == '__main__':
    curkey = None
    total = 0
    for line in sys.stdin:
        key, val = line.split("\t")
        val = int(val)

        if key == curkey:
            total += val
        else:
            if curkey is not None:
                sys.stdout.write("{}\t{}\n".format(curkey, total))

            curkey = key
            total = val
```

curkey - keeps track of the current key being processed

total - accumulates the sum of values associated with curkey

As the reducer iterates over each line in the input from stdin, it splits the line on the separator character and converts the value to an integer.

It then performs a check to ensure that we're still computing the count for the same key; otherwise, it writes the output to stdout and restarts the count for the new key. Both the mapper and reducer are executed in an "ifmain" block.

As each mapper and reducer is treated as executable by Hadoop Streaming, every

Lets consider some high-quality Python code that might be reused for different streaming jobs, and take a look at how to specifically use **Hadoop Streaming** to parse CSV data.

## Computing on CSV Data with streaming

We'll begin to put together a small, reusable framework with which we can quickly deploy Hadoop jobs for our large data-processing needs.

To start on our framework, let's consider a particular example for reading CSV data.

As Streaming serializes on a line-by-line basis, Python streaming jobs are perfect for dealing with CSV files and other plain-text file formats, many of which are

In this example, we'll consider a dataset of the on-time performance of domestic flights in the United States.

This dataset is provided by the US Department of Transportation, Bureau of Transportation Statistics, and can be downloaded from its website .

BTS makes a CSV of every domestic US flight and relevant transportation statistics such as arrival or departure delays available for analysis.

wrangle dataset is available in github

In our case, after wrangling the dataset, we have CSV data that is as follows, where each row contains

the flight date;  
the airline ID;  
a flight number;  
the origin and destination airport;  
the departure time and delay in minutes;  
the arrival time and delay in minutes;  
and finally, the amount of time in the air as well as the distance in miles.

2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00

2014-04-01,19805,2,LAX,JFK,0944,14.00,1736,-29.00,269.00,2475.00

We'll start our example of writing structured MapReduce Python code by computing the average departure delay for each airport. First, let's take a look at the mapper. Write the following code into a file called *mapper.py*:

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

import sys
import csv

SEP = "\t"

class Mapper(object):

    def __init__(self, stream, sep=SEP):
        self.stream = stream
        self.sep    = sep

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def map(self):
        for row in self:
            self.emit(row[3], row[6])

    def __iter__(self):
        reader = csv.reader(self.stream)
        for row in reader:
            yield row

if __name__ == '__main__':
    mapper = Mapper(sys.stdin)
    mapper.map()
```

Now let's take a look at the reducer.  
Write the following code in a new file

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

import sys

from itertools import groupby
from operator import itemgetter

SEP = "\t"

class Reducer(object):
```

```
def __init__(self, stream, sep=SEP):
    self.stream = stream
    self.sep    = sep

def emit(self, key, value):
    sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

def reduce(self):
    for current, group in groupby(self, itemgetter(0)):
        total = 0
        count = 0

        for item in group:
            total += item[1]
            count += 1

        self.emit(current, float(total) / float(count))

def __iter__(self):
    for line in self.stream:
        try:
            parts = line.split(self.sep)
            yield parts[0], float(parts[1])
        except:
            continue

if __name__ == '__main__':
    reducer = Reducer(sys.stdin)
    reducer.reduce()
```

## Executing Streaming Jobs

Before we get into how to execute a **Streaming job on a Hadoop cluster** by submitting the job to the job client, we'll first take a look at a useful way to **test your scripts without Hadoop overhead**.

Because Streaming makes use of the Unix standard pipes, you can **simulate the Hadoop MapReduce pipeline using Linux pipes and the sort command**.

To test your code, make sure your `mapper.py` and `reducer.py` are executable.

Simply use the `chmod` command in your terminal as follows:

**Hostname \$ chmod +x mapper.py**

**Hostname \$ chmod +x reducer.py**

The command `chmod +x mapper.py` makes it executable, so that you can run it as below:

```
#!/usr/bin/env python  
./mapper.py
```

You run the script directly from the command line, provided it has the shebang line at the top specifying the interpreter.

Similarly, you can execute `reducer.py`

```
./reducer.py
```

To test your mapper and reducer using a CSV file as input,

use the cat command to output the contents of the file,

piping the output from stdout to the stdin of the mapper.py, which pipes to sort and then to reducer.py and finally prints the result to the screen.

To test the average delay per airport mapper and reducer from the previous section, execute the following in a terminal where mapper.py, reducer.py, and flights.csv are all in your current working directory

```
hostname $ cat flights.csv | ./mapper.py | sort | ./reducer.py
ABE      -3.57142857143
ABI      55.375
ABQ      3.83333333333
ABR      -4.0
ABY      -1.33333333333
ACT      -8.2
ACV      109.142857143
ACY      -8.0
ADQ      -14.0
AEX      -6.55555555556
AGS      31.4
ALB      -1.5
ALO      -8.5
AMA      0.8
...
TWF      -7.0
TXK      -4.66666666667
TYR      -6.71428571429
TYS      12.9583333333
VEL      -7.5
VLD      -5.0
VPS      5.06666666667
WRG      -3.75
XNA      14.2580645161
YAK      -17.5
YUM      -0.22222222222
```

Unix pipes are a simple and effective way to test Hadoop Streaming mappers and reducers, and effectively illustrate how your mapper and reducer code will be used on the cluster.

This methodology is very good for quick tests as you're writing your scripts, without the overhead of waiting for the Hadoop Streaming job to complete and having to parse a Java traceback.

In order to deploy the code to the cluster, we need to submit the Hadoop Streaming JAR to the job client, passing in our custom operators as arguments.

The location of the Hadoop Streaming job depends on how you've set up and configured Hadoop.

For now, we'll assume that you have an environment variable, `$HADOOP_HOME`, that specifies the location of the install and that `$HADOOP_HOME/bin` is in your `$PATH`. If so, execute the Streaming job against the cluster as follows:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar \
 -input flights.csv \
 -output average_delay \
 -mapper mapper.py \
 -reducer reducer.py \
 -file mapper.py \
 -file reducer.py
```

Executing this command will cause the job to be started on the Hadoop cluster. The mapper.py and reducer.py scripts will be sent to each node in the cluster before processing and will be used in each phase of the pipeline.

Note the use of the `-file` option, which causes the Streaming job to send the scripts across the cluster (otherwise, they would be expected to be on the nodes already).

If there are additional files that should be sent along with the job—for example, a lookup table for the airline IDs—they can also be packaged with the job using the `-file` option.

Any third-party dependencies that you would like to use in your code should also be submitted along with the job, usually packaged in Python ZIP files.

For larger dependencies (e.g., NLTK) or dependencies that require compilation

## Usage of stderr to update the status to the Framework

Slightly more advanced usage of Hadoop Streaming takes advantage of standard error (`stderr`) to update the Hadoop status as well as Hadoop counters to the hadoop framework.

This technique essentially allows Streaming jobs to **access the Reporter object**, a part of the MapReduce Java API that tracks the global status of a job.

By writing specially formatted strings to `stderr`, **both mappers and reducers can update the global job status to report their progress and indicate they're alive**.

For jobs that take a significant amount of time (especially tasks involving the

In Python MapReduce programs, especially when using frameworks like Hadoop Streaming, `stderr` (standard error) is commonly used to send messages to the Hadoop framework, which in turn can be used to communicate with the `Reporter`.

This is useful for logging, debugging, and providing progress updates or custom status messages during the execution of your map and reduce tasks.

**1. Logging and Debugging:** You can print messages to `stderr` to log information or debug your map and reduce functions. These messages will appear in the Hadoop logs.

**2. Progress Updates:** You can inform the `Reporter` about the progress of your task to prevent it from being marked as "stuck".

**3. Custom Counters:** You can define and update custom counters via `stderr` to keep track of specific events or metrics.

To use the Counter and Status features of the Reporter, augment the Mapper and Reducer classes from the last section with the following methods:

```
def status(self, message):  
    sys.stderr.write("reporter:status:{}\n".format(message))
```

```
def counter(self, counter, amount=1, group="ApplicationCounter"):  
    sys.stderr.write(  
        "reporter:counter:{}{},{}{}\n".format(group, counter, amount)  
    )
```

The counter method allows both the map and reduce functions to update the count of any named counter by any amount necessary (defaults to incrementing by one).

The group can be set to any name, and typically the name of the application is the default.

Similarly, the status method allows the MapReduce application to send any arbitrary message to the framework, and make them visible either in logs or in the web user interfaces.

In order to extend our average flight delay application to provide a count of early and delayed flights and to send status updates on start and finish, update the map function as follows:

```
def map(self):
    self.status("mapping started")
    def map(self):
        for row in self:
            if row[6] < 0:
                self.counter("early departure")
            else:
                self.counter("late departure")

            self.emit(row[3], row[6])

    self.status("mapping complete")
```

This simple addition gives us greater insight into what is happening with average delays without a lengthy Hadoop job simply to count early and late flights

In the reducer, we may want to compute the number of airports that we have flight data for. Because the reducer will see every unique airport in our dataset, we can update our reduce function as follows:

```
def reduce(self):
    for current, group in groupby(self, itemgetter(0)):
        self.status("reducing airport {}".format(current))
        ...
        self.counter("airports")
        self.emit(current, float(total) / float(count))
```

As our analytical applications grow, these techniques to implement the full functionality of Hadoop Streaming will become vitally important.

Considering natural language processing again, in order to do part-of-speech tagging or named-entity recognition, the application necessarily will have to load pickled models into memory.

This process can take a few seconds up to a few minutes—using the status mechanism to alert the framework that the task is still running properly will ensure that speculative execution doesn't bog down the cluster. Counters help analyze large datasets even while running other jobs, and give applications a global scope with which

Speaking of global scope, there is one last tool that helps augment Streaming applications written in Python:

Job Configuration variables (JobConf variables for short).

## Counting Bigrams

- § Python code to perform word count on a file is a good example of distributed computing.
- § with hadoop streaming we can simulate this word count using a linux wc command.
- § wc command takes its input from stdin and produces output to stdout.
- § Even though word count seems to be a simple application, it serves as a basis for several large language processing applications.
- § Python supports advanced text processing techniques using which we can do lexical analysis such as bigrams count.

## Counting Bigrams

§ Hadoop streaming is well-suited for text processing because

§ It gives access to libraries like TextBlob and NLTK

§ Deals the input in a line-by-line fashion suitable for text processing

One thing is that it expects tab-delimited text values to be passed through stdin and stdout.

Let us make few improvements to WordCount and implement a map reduce program for bigrams counter.

Bigrams are pairs of consecutive words in a text

Eg: “social media”, “data science”

1. Tokenize the string
2. Normalize the tokens
3. Remove stopwords and punctuations
4. Count the Bigrams

## Counting Bigrams

Stop words:

articles (a, an...)

determiners(this, the, my...)

pronouns(his, they..)

prepositions(on, for...)

Extremely common in text and are in bulk.

So, we remove the stop words th decrease the density of the vocabulary and improve the performance.

Finally, we use our normalized corpus to count the bigrams.

Bigrams are the start of the n-gram language models to predict the next word in the context.

In order to submit this job to the cluster via Hadoop Streaming, use the same command as demonstrated earlier, but make sure to also include the *framework.py* file to be packaged and sent with the job. The job submission command is as follows:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar \
  -input corpus \
  -output bigrams \
  -mapper mapper.py \
  -reducer reducer.py \
  -file mapper.py \
  -file reducer.py \
  -file framework.py
```

# **Advanced MapReduce**

- 1. Combiners**
- 2. Partitioners**
- 3. Job Chaining**
  - 1. Linear job chaining**
  - 2. Data flow job chaining**

## Combiners

Mappers produce a lot of intermediate data that must be sent over the network to be shuffled, sorted, and reduced.

Because networking is a physical resource, large amounts of transmitted data can lead to job delays and memory bottlenecks (e.g., there is too much data for the reducer to hold into memory).

Combiners are the primary mechanism to solve this problem, and are essentially intermediate reducers that are associated with the mapper output.

Combiners reduce network traffic by performing a mapper-local reduction of the data before forwarding it on to the

Consider the following output from two  
manners and a simple sum reduction

Mapper 1 output:

(IAD, 14.4), (SF0, 3.9), (JFK, 3.9), (IAD, 12.2), (JFK, 5.8)

Mapper 2 output:

(SF0, 4.7), (IAD, 2.3), (SF0, 4.4), (IAD, 1.2)

Intended sum reduce output:

(IAD, 29.1), (JFK, 9.7), (SF0, 13.0)

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar \
  -input input_data \
  -output output_data \
  -mapper mapper.py \
  -combiner combiner.py \
  -reducer reducer.py \
  -file mapper.py \
  -file reducer.py \
  -file combiner.py
```

# Partitioners

Partitioners control how keys and their values get sent to individual reducers by dividing up the keyspace.

The default behavior is the **HashPartitioner**, which is often all that is needed.

This partitioner allocates keys evenly to each reducer by computing the hash of the key and assigning the key to a keyspace determined by the number of reducers.

Given a uniformly distributed keyspace, each reducer will get a relatively equal workload.

Takes a key and converts it to a hash value

Hash value/**Number of reducers** gives a number indicating the Reducer to which that key-value pair is sent.

The issue arises when there is a **key imbalance**, such that a **large number of values** are associated with one key, and other keys are less likely.

In this case, a **significant portion of the reducers** are **underworked**, and much of the benefit of reduction parallelism is lost.

A **custom partitioner** can ease this problem by dividing the keyspace according to some other semantic structure besides hashing

## Job Chaining

Most **complex algorithms** cannot be described as a **simple map and reduce**, so in order to implement more complex analytics, a technique called job chaining is required.

If a complex algorithm can be **decomposed into several smaller MapReduce tasks**, then these tasks can be **chained together** to produce a complete output.

Consider a computation to **compute the pairwise Pearson correlation coefficient** for a number of variables in a dataset.

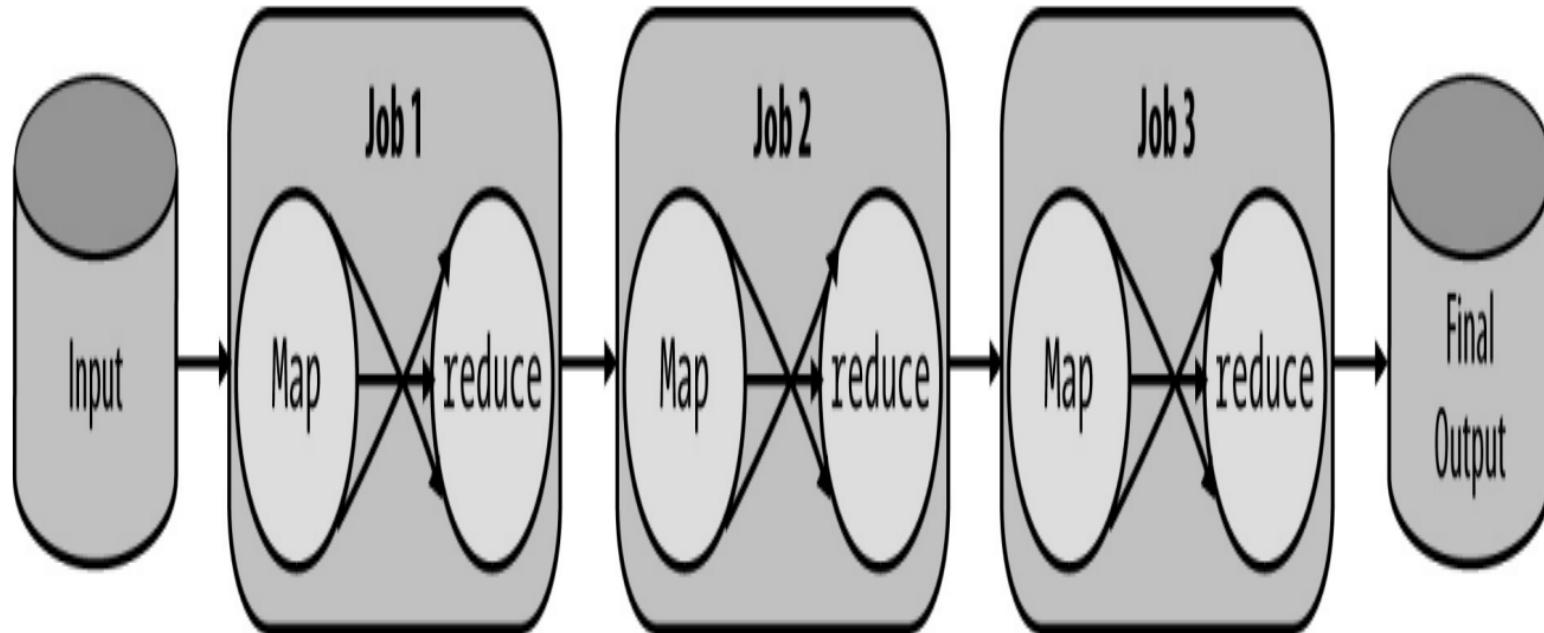


Figure 3-2. Linear job chaining produces complete computations by sending the output of one or more

## Job Chaining

The Pearson correlation requires a computation of the mean and standard deviation of each variable.

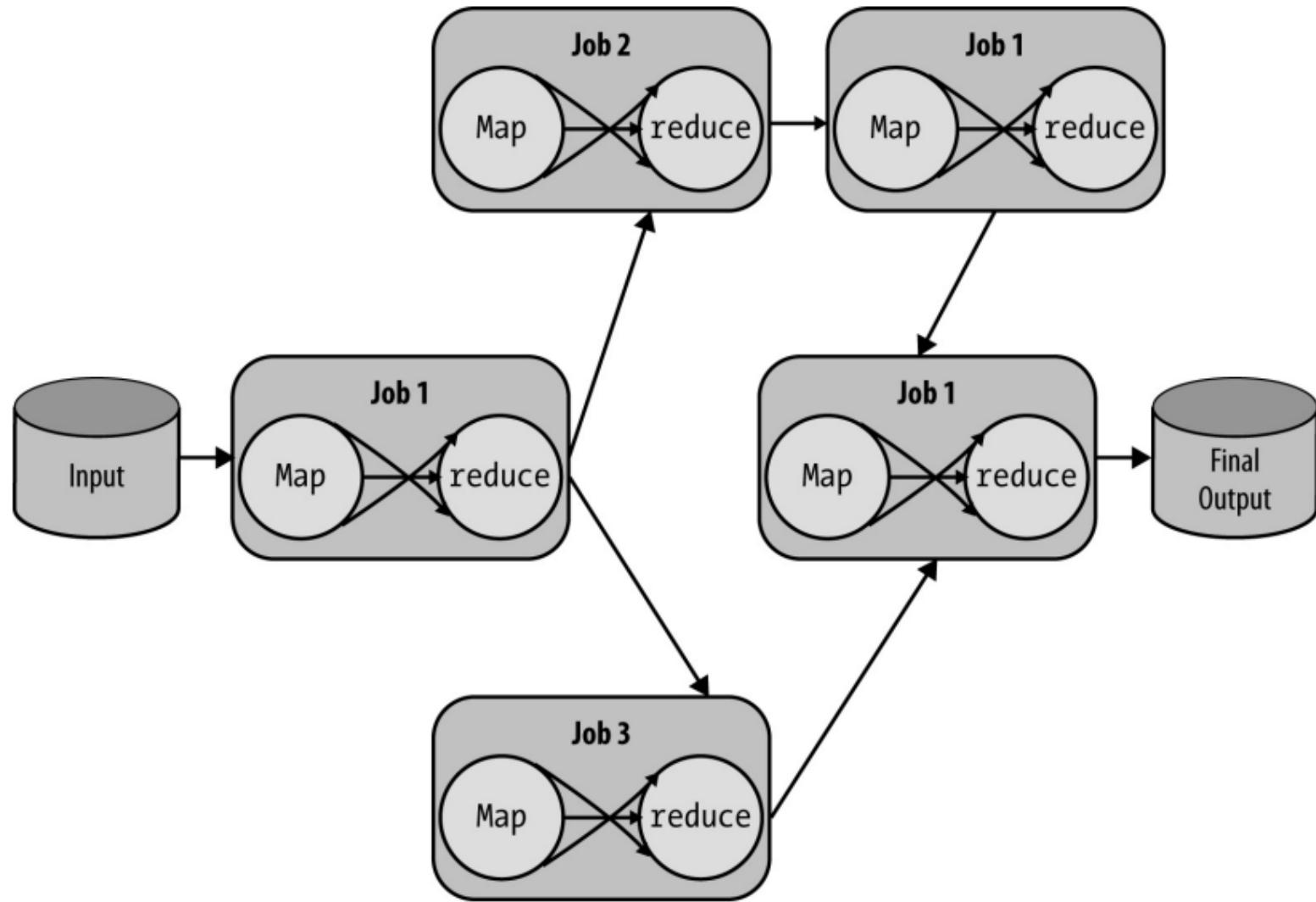
Because this cannot be easily accomplished in a single MapReduce,  
we might employ the following strategy:

1. Compute the mean and standard deviation of each (X, Y) pair.
2. Use the output of the first job to compute the covariance and Pearson correlation coefficient.

Job chaining is therefore the combination of many smaller jobs into a complete computation by sending the output of one or more previous jobs into the input of another.

*Equation 3-1. A formula for computing the sample Pearson correlation coefficient*

$$r = r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$



*Figure 3-3. Data flow job chaining is an extension of linear chaining*

However, this is a simplification of the more general form of job chaining, which is expressed as a data flow where jobs are dependent on one or more previous jobs.

Complex jobs are represented as directed acyclic graphs (DAGs) that describe how data flows from an input source through each job (the directed part) to the next job (never repeating a step, the acyclic part) and finally as final output

# INT 404R01 BIG DATA ANALYTICS

B.Tech      CSE      'A'

Year/Sem: IV/VII

Unit 2

TOPIC:MAP REDUCE AND THE NEW SOFTWARE STACK

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

# ALGORITHMS USING MAP REDUCE

Map Reduce jobs are supported by:

1. Hadoop Distributed File System(HDFS)
2. Google File System(GFS)

The original purpose for which the Google implementation of map reduce was created is:  
:

1. to execute very large matrix-vector multiplications
  - For calculation of page rank
2. to perform relational algebra operations
  - the number of friends every user has (Grouping and aggregation)
  - to find paths of length two in the web(Natural join)

## MATRIX VECTOR MULTIPLICATION BY MAP REDUCE

Suppose we have a matrix A of dimension  $m \times n$   
 $a_{ij}$  denotes the element in row  $i$  and column  $j$ .

We have a vector  $x$  of length  $n$ .

Let its  $j$ th element is denoted by  $v_j$ .

The matrix - vector product is a vector ,  
say,  $y$  of length  $n$

Its  $i$ th element  $y_i$

$$\sum_{j=1}^n a_{ij}x_j$$

$$y_i =$$

## MATRIX VECTOR MULTIPLICATION BY MAP REDUCE

If  $n=100$ , we need not use a DFS with Map reduce for this calculation.

But the actual  $n$  value will be in the order of tens of billions as it is the heart of web page ranking.

The matrix  $m$  and the vector  $v$  each will be stored in a file of the DFS.

We assume that the row-column co-ordinates of each matrix is accessible by its position in the file or it is stored with a triple such as  $(i, j, A_{ij})$ . Similarly, we assume that the position of an element in the vector is also accessible.

# MATRIX VECTOR MULTIPLICATION BY MAP REDUCE

## 1. Matrix and Vector Representation:

- Matrix  $A$  can be represented as a collection of elements  $A_{ij}$  where  $i$  is the row index and  $j$  is the column index.
- Vector  $x$  can be represented as a collection of elements  $x_j$  where  $j$  is the index.

## 2. Map Phase:

- Each element  $A_{ij}$  of the matrix is paired with the corresponding element  $x_j$  of the vector.
- For each pair  $(i, j)$ , the map function emits a key-value pair:  $(i, A_{ij} \times x_j)$ .
- This means that for each element in the matrix, we multiply it by the corresponding element in the vector and associate the result with the row index  $i$ .



# MATRIX VECTOR MULTIPLICATION BY MAP REDUCE

## Example

Given:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

*ALGORITHMS USING MAP REDUCE*

# MATRIX VECTOR MULTIPLICATION BY MAP REDUCE

## Map Phase

For each element  $A_{ij}$  in the matrix and corresponding  $x_j$  in the vector, the map function produces key-value pairs  $(i, A_{ij} \times x_j)$ :

- $(1, a_{11} \times x_1)$
- $(1, a_{12} \times x_2)$
- $(2, a_{21} \times x_1)$
- $(2, a_{22} \times x_2)$

# MATRIX VECTOR MULTIPLICATION BY MAP REDUCE

## Shuffle and Sort Phase

In this phase, we group the key-value pairs by their keys (row indices):

1. Key = 1:

- $(1, a_{11} \times x_1)$
- $(1, a_{12} \times x_2)$

Grouped together:  $(1, [a_{11} \times x_1, a_{12} \times x_2])$

2. Key = 2:

- $(2, a_{21} \times x_1)$
- $(2, a_{22} \times x_2)$

Grouped together:  $(2, [a_{21} \times x_1, a_{22} \times x_2])$

## Result of Shuffle and Sort

After the shuffle and sort phase, we have the following grouped intermediate results:

- $(1, [a_{11} \times x_1, a_{12} \times x_2])$
- $(2, [a_{21} \times x_1, a_{22} \times x_2])$

## Reduce Phase

In the reduce phase, we sum the values for each key:

1. Key = 1:

- Sum the list  $[a_{11} \times x_1, a_{12} \times x_2]$
- Result:  $y_1 = (a_{11} \times x_1) + (a_{12} \times x_2)$

2. Key = 2:

- Sum the list  $[a_{21} \times x_1, a_{22} \times x_2]$
- Result:  $y_2 = (a_{21} \times x_1) + (a_{22} \times x_2)$

# MATRIX VECTOR MULTIPLICATION BY MAP REDUCE

## Final Output

The final output vector  $y$  after the reduce phase is:

$$y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} (a_{11} \times x_1) + (a_{12} \times x_2) \\ (a_{21} \times x_1) + (a_{22} \times x_2) \end{pmatrix}$$

In summary, the shuffle and sort phase outputs the grouped intermediate results for each row of the matrix, which are then summed in the reduce phase to produce the final elements of the resulting vector.

## If the Vector $v$ Cannot Fit in Main Memory

We divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height.

Our goal is to use enough stripes so that the portion of the vector in one stripe can fit conveniently into main memory at a compute node. Figure 2.4 suggests what the partition looks like if the matrix and vector are each divided into five stripes.

# If the vector $v$ cannot fit in Main Memory

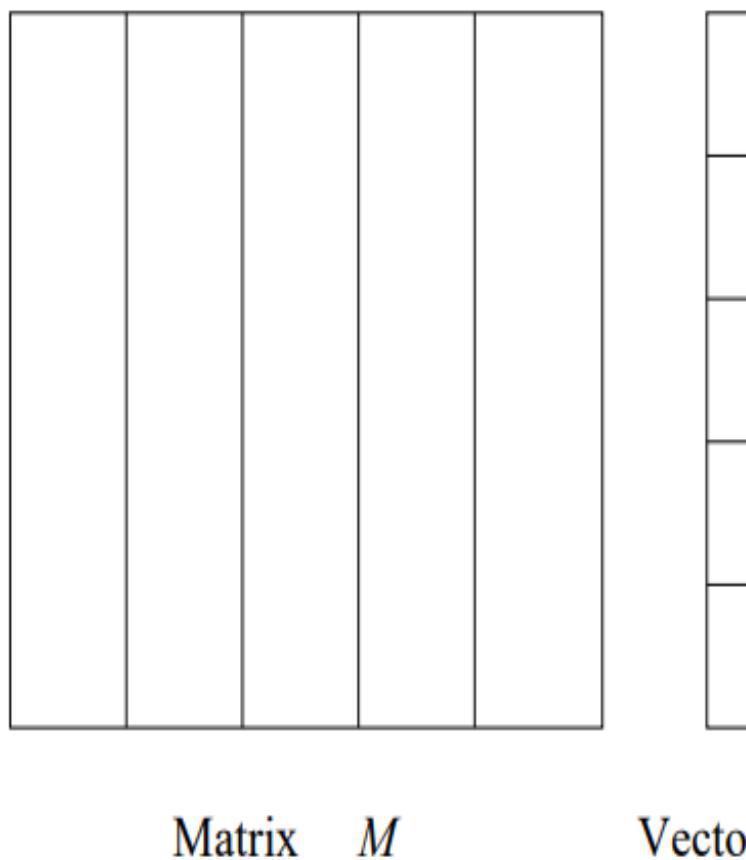


Figure 2.4: Division of a matrix and vector into five stripes

## If the vector $v$ cannot fit in Main Memory

The  $i$ th stripe of the matrix multiplies only components from the  $i$ th stripe of the vector. Thus, we can divide the matrix into one file for each stripe, and do the same for the vector.

Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector.

The Map and Reduce tasks can then act exactly as was described above for the case where Map tasks get the entire vector.

If the vector  $v$  cannot fit in main

Memory

For Page rank calculation, we have an additional constraint that the result vector should be partitioned in the same way as the input vector, so the output may become the input for another iteration of the matrix-vector multiplication.

## Relational-Algebra Operations

Relational algebra operations are required to be done on large scale data. Many traditional database applications involve retrieval of small amounts of data, even though the database itself may be large.

For example, a query may ask for the bank balance of one particular account. Such queries are not useful applications of MapReduce

# Relational-Algebra Operations

A good starting point for exploring applications of MapReduce is by considering the standard operations on relations.

We assume you are familiar with database systems, the query language SQL, and the relational model.

A small review on the terminology is here: a relation is a table with column headers called attributes. Rows of the relation are called tuples.

The set of attributes of a relation is called its schema. We often write an expression like  $R(A_1, A_2, \dots, A_n)$  to say that the relation name is  $R$  and its attributes are  $A_1, A_2, \dots, A_n$ .

# Relational-Algebra Operations

<i>From</i>	<i>To</i>
url1	url2
url1	url3
url2	url3
url2	url4
...	...

Figure 2.5: Relation *Links* consists of the set of pairs of URL's, such that the first has one or more links to the second

## Relational-Algebra Operations

**Example:** Figure. 2.5 we see part of the relation Links that describes the structure of the Web.

There are two attributes, From and To. A row, or tuple, of the relation is a pair of URL's, such that there is at least one link from the first URL to the second.

For instance, the first row of Fig. 2.5 is the pair (url1, url2) that says the web page url1 has a link to page url2.

while we have shown only four tuples, the real relation of the Web, or the portion of it that would be stored by a typical search engine, has billions of tuples.

# **Relational-Algebra Operations**

A relation, however large, can be stored as a file in a distributed file system. The elements of this file are the tuples of the relation.

There are several standard operations on relations, often referred to as relational algebra, that are used to implement queries.

The queries themselves usually are written in SQL. The relational-algebra operations we shall discuss are:

- 1. Selection**
- 2. Projection**
- 3. Union, Intersection and Difference**
- 4. Natural Join**
- 5. Grouping and aggregation**

# Relational-Algebra Operations

## 1. Selection:

Apply a condition  $C$  to each tuple in the relation and produce as output only those tuples that satisfy  $C$ . The result of this selection is denoted  $\sigma_C(R)$ .

## 2. Projection:

For some subset  $S$  of the attributes of the relation, produce from each tuple only the components for the attributes in  $S$ . The result of this projection is denoted  $\pi_S(R)$ .

## 3. Union, Intersection, and Difference:

These well-known set operations apply to the sets of tuples in two relations that have the same schema.

# Relational-Algebra Operations

## 4. Natural Join:

A natural join is a type of join operation in relational databases that combines two tables based on the common columns they share.

When performing a natural join, the database system automatically matches rows from the first table with rows from the second table where the values in the shared columns are equal. It then combines these matching rows into a single row, eliminating duplicate columns.

This operation simplifies queries by automatically handling the ~~ALGORITHMS USING MAP REDUCE~~ column matching, making it useful for combining tables with a clear relational link.

# Relational-Algebra Operations

## Students Table

student_id	name
1	Alice
2	Bob
3	Carol

## Enrollments Table

student_id	course_id
1	CS101
2	CS102
1	CS103
3	CS101

*ALGORITHMS USING MAP REDUCE*

When we perform a natural join on the `Students` and `Enrollments` tables based on the `student\_id` column, we get:

student_id	name	course_id
1	Alice	CS101
1	Alice	CS103
2	Bob	CS102
3	Carol	CS101

# Relational-Algebra Operations

## 5. Grouping and Aggregation

Grouping is the process of organizing rows in a table into sets, or groups, based on the values of one or more columns (attributes). Each group contains all rows that have the same values in the specified columns.

Aggregation involves applying aggregate functions to the grouped data to produce summary information. Common aggregate functions include:

**SUM:** Adds up the values in a group.

**AVG:** Calculates the average of the values in a group.

**COUNT:** Counts the number of rows in a group.  
ALGORITHMS USING MAP REDUCE

**MAX:** Finds the maximum value in a group.

# Relational-Algebra Operations

## Example

Consider the following `Sales` table:

salesperson	region	amount
Alice	North	100
Bob	North	150
Carol	South	200
Alice	North	50
Bob	South	300

## Grouping and Aggregation

Suppose we want to find the total sales amount for each region. We would:

1. Group the rows by the `region` column.
2. Aggregate the `amount` column using the SUM function.

*ALGORITHMS USING MAP REDUCE*

# Relational-Algebra Operations

The relational algebra expression for this operation could be written as:

$$\gamma_{region, SUM(amount)}(\text{Sales})$$

## Result

The result of this grouping and aggregation would be:

region	total_sales
North	300
South	500

# Relational-Algebra Operations

## Example :

Let us try to find the paths of length two in the web, using the relation Links of Fig. 2.5(slide number 17).

That is, we want to find the triples of URL's  $(u, v, w)$  such that there is a link from  $u$  to  $v$  and a link from  $v$  to  $w$ .

We essentially want to take the natural join of Links with itself, but we first need to imagine that it is two relations, with different schemas, so we can describe the desired connection as a natural join.

Thus, imagine that there are ALGORITHMS USING MAP REDUCE two copies of Links, namely  $L1(u_1, u_2)$  and  $L2(u_2, u_3)$ .

## Relational-Algebra Operations

Now, if we compute  $L1 \bowtie L2$ , we shall have exactly what we want.

That is, for each tuple  $t1$  of  $L1$  (i.e., each tuple of Links) and each tuple  $t2$  of  $L2$  (another tuple of Links, possibly even the same tuple), see if their  $U2$  components are the same.

Note that these components are the second component of  $t1$  and the first component of  $t2$ . If these two components agree, then produce a tuple for the result, with schema  $(U1, U2, U3)$ .

This tuple consists of the ALGORITHMS USING MAP REDUCE first component of  $t1$ , the second component of  $t1$  (which must equal the first component of  $t2$ ), and

# Relational-Algebra Operations

We may not want the entire path of length two, but only want the pairs  $(u, w)$  of URL's such that there is at least one path from  $u$  to  $w$  of length two.

If so, we can project out the middle components by computing  $\pi_{U1, U3}(L1 \bowtie L2)$ .

To solve this example , we used the two relational algebra operations:

1. Natural Join and
2. Projection

# Relational-Algebra Operations

**Example :** Imagine that a social-networking site has a relation **Friends**(User, Friend). This relation has tuples that are pairs (a, b) such that b is a friend of a.

The site might want to develop statistics about the number of friends members have.

Their first step would be to compute a count of the number of friends of each user.

This operation can be done by grouping and aggregation, specifically  
 $\gamma_{\text{User}}, \text{COUNT}(\text{Friend})(\text{Friends})$

ALGORITHMS USING MAP REDUCE

# Relational-Algebra Operations

This operation **groups** all the tuples by the value in their first component, so there is one group for each user.

Then, for each group the count of the number of friends of that user is made.

The result will be one tuple for each group, and a typical tuple would look like (Sally, 300), if user “Sally” has 300 friends.

To solve this example , we used the two relational algebra operations:

1. Grouping
2. Aggregation

# Computing Selections by MapReduce

Selections really do not need the full power of MapReduce. They can be done most conveniently in the map portion alone, although they could also be done in the reduce portion alone.

Here is a MapReduce implementation of selection  $\sigma_C(R)$ .

**The Map Function:** For each tuple  $t$  in  $R$ , test if it satisfies  $C$ . If so, produce the key-value pair  $(t, t)$ . That is, both the key and value are  $t$ .

**The Reduce Function:** The Reduce function is the identity. It simply passes each key-value pair to the output. Note that the output is not exactly a relation, because it has key-value pairs. However, a relation can be obtained by using only the value

# Computing Selections by MapReduce

## Example Scenario

Consider a table `Employees` with the following data:

employee_id	name	age	department
1	Alice	30	HR
2	Bob	25	IT
3	Carol	35	IT
4	Dave	40	Finance
5	Eve	28	HR

Suppose we want to select all employees who work in the "IT" department.

# Computing Selections by MapReduce

## Map Phase

For each row in the `Employees` table, the map function checks if the `department` is "IT". If the condition is met, it emits the row.

- Input to a mapper:
  - Row 1: (1, Alice, 30, HR)
  - Row 2: (2, Bob, 25, IT)
  - Row 3: (3, Carol, 35, IT)
  - Row 4: (4, Dave, 40, Finance)
  - Row 5: (5, Eve, 28, HR)

# Computing Selections by MapReduce

- Map output (only emitting rows where department is "IT"):
  - $(\_, (2, \text{Bob}, 25, \text{IT}))$
  - $(\_, (3, \text{Carol}, 35, \text{IT}))$

## Shuffle and Sort Phase

The shuffle and sort phase collects all emitted values (rows):

- Collected rows:
  - $(2, \text{Bob}, 25, \text{IT})$
  - $(3, \text{Carol}, 35, \text{IT})$

# Computing Selections by MapReduce

## Reduce Phase

The reduce function processes the collected rows and outputs them:

- Output:
  - (2, Bob, 25, IT)
  - (3, Carol, 35, IT)

## Final Result

The result of the MapReduce selection operation is the set of rows where the `department` is "IT":

employee_id	name	age	department
2	Bob	25	IT
3	Carol	35	IT

# Computing Projections by MapReduce

Projection is performed similarly to selection, because projection may cause the same tuple to appear several times, the Reduce function must eliminate duplicates.

We may compute  $\pi_S(R)$  as follows.

The Map Function: For each tuple  $t$  in  $R$ , construct a tuple  $t \sqcap$  by eliminating from  $t$  those components whose attributes are not in  $S$ . Output the key-value pair  $(t \sqcap, t \sqcap)$ .

The Reduce Function: For each key  $t \sqcap$  produced by any of the Map tasks, there will be one or more key-value pairs  $(t \sqcap, t \sqcap)$ . The Reduce function turns  $(t \sqcap, [t \sqcap, t \sqcap, \dots, t \sqcap])$  into  $(t \sqcap, t \sqcap)$ , so it produces exactly one pair  $(t \sqcap, t \sqcap)$  for this key.

# Computing Projections by MapReduce

- Input to reducer:
  - (Alice, HR)
  - (Alice, HR)
  - (Bob, IT)
  - (Carol, IT)
  - (Carol, IT)
  - (Dave, Finance)
  - (Eve, HR)
- Reduce output (eliminating duplicates):
  - (Alice, HR)
  - (Bob, IT)
  - (Carol, IT)
  - (Dave, Finance)
  - (Eve, HR)

## **Union, Intersection, and Difference by MapReduce**

**Union:** First, consider the union of two relations. Suppose relations R and S have the same schema.

**Map tasks will be assigned chunks from either R or S; it doesn't matter which.**

The Map tasks don't really do anything except pass their input tuples as key-value pairs to the Reduce tasks. The latter need only eliminate duplicates as for projection.

**The Map Function:** Turn each input tuple  $t$  into a key-value pair  $(t, t)$ .

**The Reduce Function:** Associated with each key  $t$  there will be either one or two values. Produce output  $(t, t)$  in either

*ALGORITHMS USING MAP REDUCE*

- Map output:

- $(\_, (1, \text{Alice}, \text{HR}))$
- $(\_, (2, \text{Bob}, \text{IT}))$
- $(\_, (3, \text{Carol}, \text{IT}))$
- $(\_, (4, \text{Dave}, \text{Finance}))$
- $(\_, (5, \text{Eve}, \text{HR}))$
- $(\_, (6, \text{Frank}, \text{Marketing}))$

- Reduce output:

- $(1, \text{Alice}, \text{HR})$
- $(2, \text{Bob}, \text{IT})$
- $(3, \text{Carol}, \text{IT})$
- $(4, \text{Dave}, \text{Finance})$
- $(5, \text{Eve}, \text{HR})$
- $(6, \text{Frank}, \text{Marketing})$

# Union, Intersection, and Difference by MapReduce

## Intersection.

To compute the intersection, we can use the same Map function.

However, the Reduce function must produce a tuple only if both relations have the tuple.

If the key  $t$  has a list of two values  $[t, t]$  associated with it, then the Reduce task for  $t$  should produce  $(t, t)$ .

However, if the value-list associated with key  $t$  is just  $[t]$ , then one of  $R$  and  $S$  is missing  $t$ , so we don't want to produce a tuple for the intersection.

**The Map Function:** Turn each tuple  $t$  into a key-value pair  $(t, t)$ .  
**The Reduce Function:** If key  $t$  has value list  $[t, t]$ , then produce  $(t, t)$ . Otherwise, produce nothing.

# Union, Intersection, and Difference by MapReduce

## Map Phase

For each row in `Employees1` and `Employees2`, the map function emits the row data with a flag indicating the source table.

- For `Employees1`:
  - Row 1: (Alice, HR, "table1")
  - Row 2: (Bob, IT, "table1")
  - Row 3: (Carol, IT, "table1")
- For `Employees2`:
  - Row 1: (Bob, IT, "table2")
  - Row 2: (Carol, IT, "table2")
  - Row 3: (Dave, Finance, "table2")

# Union, Intersection, and Difference by MapReduce

- Map output:

- (Alice, HR, "table1")
- (Bob, IT, "table1")
- (Carol, IT, "table1")
- (Bob, IT, "table2")
- (Carol, IT, "table2")
- (Dave, Finance, "table2")

- Input to reducer:

- (Alice, HR) : "table1"
- (Bob, IT) : "table1", "table2"
- (Carol, IT) : "table1", "table2"
- (Dave, Finance) : "table2"

- Reduce output (only rows appearing in both tables):

- (Bob, IT)
- (Carol, IT)

# Union, Intersection, and Difference by MapReduce

## Difference

The Difference  $R - S$  requires a bit more thought. The only way a tuple  $t$  can appear in the output is if it is in  $R$  but not in  $S$ .

The Map function can pass tuples from  $R$  and  $S$  through, but must inform the Reduce function whether the tuple came from  $R$  or  $S$ .

**The Map Function:** For a tuple  $t$  in  $R$ , produce key-value pair  $(t, R)$ , and for a tuple  $t$  in  $S$ , produce key-value pair  $(t, S)$ . Note that the intent is that the value is the name of  $R$  or  $S$  (or better, a single bit indicating whether the relation is  $R$  or  $S$ ), not the entire relation..

## Union, Intersection, and Difference by MapReduce

**The Reduce Function:** For each key  $t$ , if the associated value list is  $[R]$ , then produce  $(t, t)$ .

- Map output:

- $(-, (1, \text{Alice}, \text{HR}, \text{"table1"}))$
- $(-, (2, \text{Bob}, \text{IT}, \text{"table1"}))$
- $(-, (3, \text{Carol}, \text{IT}, \text{"table1"}))$
- $(-, (2, \text{Bob}, \text{IT}, \text{"table2"}))$
- $(-, (3, \text{Carol}, \text{IT}, \text{"table2"}))$
- $(-, (4, \text{Dave}, \text{Finance}, \text{"table2"}))$

# Union, Intersection, and Difference by MapReduce

- Input to reducer:
  - $(1, \text{Alice}, \text{HR}) : \text{"table1"}$
  - $(2, \text{Bob}, \text{IT}) : \text{"table1", "table2"}$
  - $(3, \text{Carol}, \text{IT}) : \text{"table1", "table2"}$
  - $(4, \text{Dave}, \text{Finance}) : \text{"table2"}$
- Reduce output (only rows appearing in `table1` but not in `table2`):
  - $(1, \text{Alice}, \text{HR})$

## **Computing Natural Join by MapReduce**

Performing a natural join using MapReduce involves **combining rows from two datasets based on common values in their join key(s)**. A natural join automatically joins columns with the same name from both tables.

We look at the specific case of **joining R(A, B) with S(B, C)**. We must find tuples that agree on their B components, that is the **second component from tuples of R and the first component of tuples of S**.

**The Map Function:** For each tuple (a, b) of R, produce the key-value pair b,(R, a) . For each tuple (b, c) of S, produce the key-value pair b,(S, c)

**The Reduce Function:** Each key value b will

## Computing Natural Join by MapReduce

Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c).

The output from this key and value list is a sequence of key-value pairs. The key is irrelevant. Each value is one of the triples (a, b, c) such that (R, a) and (S, c) are on the input list of value.

# Computing Natural Join by MapReduce

Employees Table

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Carol	101

Departments Table

department_id	department_name
101	HR
102	IT
103	Finance

- Map output:

- (101, ("Employees", 1, "Alice"))
- (102, ("Employees", 2, "Bob"))
- (101, ("Employees", 3, "Carol"))
- (101, ("Departments", "HR"))
- (102, ("Departments", "IT"))
- (103, ("Departments", "Finance"))

We want to perform a natural join on the `department_id` column, producing a combined table with all matching rows from both tables.

# Computing Natural Join by MapReduce

**Input to Reducer**

For 101: [ ("Employees", 1, "Alice"),  
("Employees", 3, "Carol"),  
("Departments", "HR")]

For 102: [ ("Employees", 2, "Bob"), ("Department

For 103: [ ("Departments", "Finance")]

- Reduce output (combining matching rows):
  - (1, "Alice", 101, "HR")
  - (3, "Carol", 101, "HR")
  - (2, "Bob", 102, "IT")

# Grouping and Aggregation by MapReduce

We shall discuss the minimal example of grouping and aggregation, where there is one grouping attribute and one aggregation.

Let  $R(A, B, C)$  be a relation to which we apply the operator  $\gamma A, \theta(B)(R)$ .

Map will perform the grouping, while Reduce does the aggregation.

**The Map Function:** For each tuple  $(a, b, c)$  produce the key-value pair  $(a, b)$

# Grouping and Aggregation by MapReduce

## The Reduce Function:

Each key  $a$  represents a group. Apply the aggregation operator  $\theta$  to the list  $[b_1, b_2, \dots, b_n]$  of  $B$ -values associated with key  $a$ .

The output is the pair  $(a, x)$ , where  $x$  is the result of applying  $\theta$  to the list. For example, if  $\theta$  is SUM, then  $x = b_1 + b_2 + \dots + b_n$ , and if  $\theta$  is MAX, then  $x$  is the largest of  $b_1, b_2, \dots, b_n$ .

Grouping and aggregation in MapReduce involve organizing data into groups based on a key and then applying some aggregation function to each group.

*ALGORITHMS USING MAP REDUCE*

For example, you might want to group

# Grouping and Aggregation by MapReduce

## `Employees` Table

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Carol	101
4	Dave	103
5	Eve	102

We want to count the number of employees in each department.

# Grouping and Aggregation by MapReduce

- Map output:
  - (101, 1)
  - (102, 1)
  - (101, 1)
  - (103, 1)
  - (102, 1)
- Reduce Phase
  - The reduce function processes the grouped values, summing the counts for each key:
    - Input to reducer:
      - 101: [1, 1]
      - 102: [1, 1]
      - 103: [1]
    - Reduce output (sum of counts):
      - (101, 2)
      - (102, 2)
      - (103, 1)

## Matrix Multiplication

If  $M$  is a matrix with element  $m_{ij}$  in row  $i$  and column  $j$ , and  $N$  is a matrix with element  $n_{jk}$  in row  $j$  and column  $k$ , then the product  $P = M \cdot N$  has element  $p_{ik}$  in row  $i$  and column  $k$ .  
$$p_{ik} = \sum_j m_{ij} n_{jk}$$

It is required that the number of columns of  $M$  equals the number of rows of  $N$ , so the sum over  $j$  makes sense.

- We can think of a matrix as a relation with three attributes: the row number, the column number, and the value in that row and column.

## Matrix Multiplication

Thus, we could view matrix M as a relation  $M(I, J, V)$ , with tuples  $(i, j, mij)$ , and we could view matrix N as a relation  $N(J, K, W)$ , with tuples  $(j, k, njk)$ .

As large matrices are often sparse (mostly 0's), and since we can omit the tuples for matrix elements that are 0, this relational representation is often a very good one for a large matrix.

However, it is possible that i, j, and k are implicit in the position of a matrix element in the file that represents it, rather than written explicitly with the element itself.

*ALGORITHMS USING MAP REDUCE*

In that case, the Map function will have to

## Matrix Multiplication

The product  $MN$  is almost a natural join followed by grouping and aggregation.

That is, the natural join of  $M(I, J, V)$  and  $N(J, K, W)$ , having only attribute  $J$  in common, would produce tuples  $(i, j, k, v, w)$  from each tuple  $(i, j, v)$  in  $M$  and tuple  $(j, k, w)$  in  $N$ .

This five-component tuple represents the pair of matrix elements  $(m_{ij}, n_{jk})$ .

What we want instead is the product of these elements, that is, the four-component tuple  $(i, j, k, v \times w)$ , because that represents the product  $m_{ij} \cdot n_{jk}$ .  
ALGORITHMS USING MAP REDUCE

# Matrix Multiplication

Once we have this relation as the result of one MapReduce operation, we can perform grouping and aggregation, with I and K as the grouping attributes and the sum of  $v \times w$  as the aggregation.

That is, we can implement matrix multiplication as the cascade of two MapReduce operations, as follows.

## First Map-Reduce Operation:

**The Map Function:** For each matrix element  $m_{ij}$ , produce the key value pair  $j, (M, i, m_{ij})$ . Likewise, for each matrix element  $n_{jk}$ , produce the key value pair  $j, (N, k, n_{jk})$ .

# Matrix Multiplication

Note that M and N in the values are not the matrices themselves.

Rather they are names of the matrices or (as we mentioned for the similar Map function used for natural join) better, a bit indicating whether the element comes from M or N.

**The Reduce Function:** For each key j, examine its list of associated values. For each value that comes from M, say (M, i, mij), and each value that comes from N, say (N, k, njk), produce a key-value pair with key equal to (i, k) and value equal to the product of these elements.

AS OF 11/15/13. MAP REDUCE

# Matrix Multiplication

Now, we perform a grouping and aggregation by another MapReduce operation.

## Second Map-Reduce Operation

**The Map Function:** This function is just the identity. That is, for every input element with key  $(i, k)$  and value  $v$ , produce exactly this key-value pair.

**The Reduce Function:** For each key  $(i, k)$ , produce the sum of the list of values associated with this key. The result is a pair  $(i, k), v$ , where  $v$  is the value of the element in row  $i$  and column  $k$  of the matrix  $P = M$

## MATRIX Multiplication with One MapReduce Step

There often is more than one way to use MapReduce to solve a problem. You may wish to use only a single MapReduce pass to perform matrix multiplication  $P = MN$ .

It is possible to do so if we put more work into the two functions. Start by using the Map function to create the sets of matrix elements that are needed to compute each element of the answer  $P$ .

Notice that an element of  $M$  or  $N$  contributes to many elements of the result, so one input element will be turned into many key-value pairs. The keys will be pairs  $(i, k)$ , where  $i$  is a row of  $M$  and  $k$  is a column of  $N$ . Here is a synopsis of the Map and Reduce functions

## MATRIX MULTIPLICATION WITH ONE MapReduce Step

**The Map Function:** For each element  $m_{ij}$  of  $M$ , produce all the key-value pairs  $(i, k)$ ,  $(M, j, m_{ij})$  for  $k = 1, 2, \dots$ , up to the number of columns of  $N$ .

Similarly, for each element  $n_{jk}$  of  $N$ , produce all the key-value pairs  $(i, k)$ ,  $(N, j, n_{jk})$  for  $i = 1, 2, \dots$ , up to the number of rows of  $M$ . As before,  $M$  and  $N$  are really bits to tell which of the two matrices a value comes from.

# MATRIX MULTIPLICATION WITH ONE MapReduce Step

## The Reduce Function:

Each key  $(i, k)$  will have an associated list with all the values  $(M, j, mij)$  and  $(N, j, n_{jk})$ , for all possible values of  $j$ .

The Reduce function needs to connect the two values on the list that have the same value of  $j$ , for each  $j$ .

An easy way to do this step is to sort by  $j$  the values that begin with  $M$  and sort by  $j$  the values that begin with  $N$ , in separate lists.

The  $j$ th values on each list must have their third components,  $mij$  and  $n_{jk}$  extracted and multiplied. Then, these products are summed and the result is paired with  $(i, k)$  in the

# MATRIX MULTIPLICATION WITH ONE MapReduce Step

## The Reduce Function:

Each key  $(i, k)$  will have an associated list with all the values  $(M, j, mij)$  and  $(N, j, n_{jk})$ , for all possible values of  $j$ .

The Reduce function needs to connect the two values on the list that have the same value of  $j$ , for each  $j$ .

An easy way to do this step is to sort by  $j$  the values that begin with  $M$  and sort by  $j$  the values that begin with  $N$ , in separate lists.

The  $j$ th values on each list must have their third components,  $mij$  and  $n_{jk}$  extracted and multiplied. Then, these products are summed and the result is paired with  $(i, k)$  in the

## MINING DATA STREAMS

Most of the algorithms assume that we are mining a database. That is, all our data is available when and if we want it.

We shall make another assumption: data arrives in a stream or streams, and if it is not processed immediately or stored, then it is lost forever.

Moreover, we shall assume that the data arrives so rapidly that it is not feasible to store it all in active storage (i.e., in a conventional database), and then interact with it at the time of our choosing.

The algorithms for processing streams each involve summarization of the stream in some way.

We shall start by considering how to make a useful sample of a stream and how to filter a stream to eliminate most of the “undesirable” elements.

We then show how to estimate the number of different elements in a stream using much less storage than would be required if we listed all the elements we have seen.

### Topics to be covered

1. The stream data model
  1. A Data stream management system
  2. Examples of stream sources
  3. Stream queries
  4. Issues in stream processing
2. Sampling data in a stream
  1. A motivating example
  2. Obtaining a representative sample
  3. The general sampling problem
  4. Varying the sample size

# MINING DATA STREAMS

## The Stream Data Model

Let us begin by discussing the elements of streams and stream processing.

Stream is a continuous flow of data from various external sources like sensors, satellites, social media updates etc. that flow rapidly.

A stream involves huge amount of data at high speed in real time that it must be processed immediately otherwise we may lose it.

It is difficult to store stream data in database as it would require a lot of memory.

# MINING DATA STREAMS

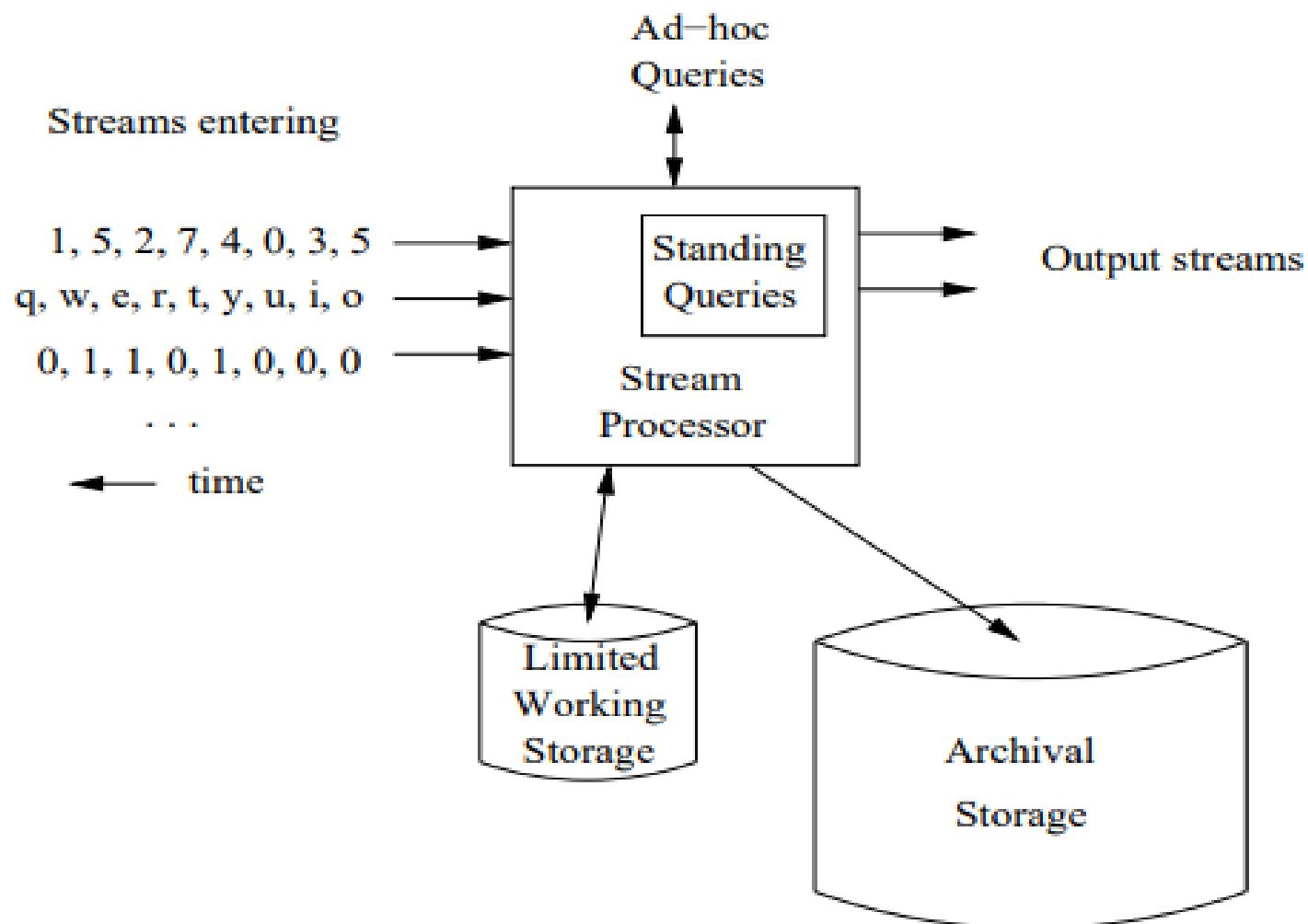


Figure 4.1: A data-stream-management system

# MINING DATA STREAMS

## A Data-Stream-Management System

In analogy to a database-management system, we can view a stream processor as a kind of data-management system, the high-level organization of which is suggested in Fig. 4.1.

Any number of streams can enter the system. Each stream can provide elements at its own schedule; they need not have the same data rates or data types, and the time between elements of one stream need not be uniform.

The fact that the rate of arrival of stream elements is not under the control of the system distinguishes stream processing from the processing of data that goes on within a database-management system.

## **MINING DATA STREAMS**

Stream processor is the crucial component which processes the input stream and generates the output stream with low delay to gain valuable insights. It involves both standing queries and ad-hoc queries.

Standing queries are the pre-defined queries that are automatically executed over the input streams without user intervention. They are used for monitoring and alerting works.

For example, they detect whether input stream contains malicious patterns and if so generate alerts.

*MINING DATA STREAMS*

Ad-hoc queries are the user queries on-demand according to their specific needs. Users may wish to explore the input streams

# MINING DATA STREAMS

There is a working storage of limited capacity. It is the main memory available to store crucial data such as:

Intermediate results

Most recent data

Most relevant data

Subset of data required for answering the queries

There is an archival storage of high capacity which can store the processed data. It contains all the historical data for future references.

# MINING DATA STREAMS

## Examples of Stream Sources

Before proceeding, let us consider some of the ways in which stream data arises naturally.

### Sensor Data

Imagine a **temperature sensor** bobbing about in the ocean, sending back to a base station a reading of the surface temperature each hour.

The data produced by this sensor is a stream of real numbers. It is not a very interesting stream, since the data rate is so low.

It would not stress modern technology, and the entire stream could be kept in main memory, essentially forever.

*MINING DATA STREAMS*

## MINING DATA STREAMS

Now, give the sensor a GPS unit, and let it report **surface height** instead of temperature.

The surface height varies quite rapidly compared with temperature, so we might have the sensor send back a reading every tenth of a second.

If it sends a **4-byte real number each time**, then it produces **3.5 megabytes per day**. It will still take some time to fill up main memory, let alone a single disk.

But one sensor might not be that interesting. To learn something <sup>MINING DATA STREAMS</sup> about ocean behavior, we might want to deploy a million sensors, each sending back a stream, at the

## MINING DATA STREAMS

A million sensors isn't very many; there would be one for every 150 square miles of ocean. Now we have 3.5 terabytes arriving every day, and we definitely need to think about what can be kept in working storage and what can only be archived.

### Image Data

Satellites often send down to earth streams consisting of many terabytes of images per day.

Surveillance cameras produce images with lower resolution than satellites, but there can be many of them, each producing a stream of images at intervals like one second.

# MINING DATA STREAMS

## Internet and Web Traffic

A Switching node in the middle of the Internet receives streams of IP packets from many inputs and routes them to its outputs.

Normally, the job of the switch is to transmit data and not to retain it or query it. But there is a tendency to put more capability into the switch,

e.g., the ability to detect denial-of-service attacks or the ability to reroute packets based on information about congestion in the network.

## MINING DATA STREAMS

Web sites receive streams of various types. For example, Google receives several hundred million search queries per day.

Yahoo! accepts billions of “clicks” per day on its various sites. Many interesting things can be learned from these streams.

For example, an increase in queries like “sore throat” enables us to track the spread of viruses.

A sudden increase in the click rate for a link could indicate some news connected to that page, or it could mean that the link is broken and needs to be repaired

# MINING DATA STREAMS

## Stream Queries

There are two ways that queries get asked about streams.

### 1. Standing Queries

These queries are, in a sense, permanently executing, and produce outputs at appropriate times.

Example: The stream produced by the ocean-surface-temperature sensor might have a standing query to output an alert whenever the temperature exceeds 25 degrees centigrade. This query is easily answered, since it depends only on the most recent stream element.

## MINING DATA STREAMS

Alternatively, we might have a standing query that, each time a new reading arrives, produces the average of the 24 most recent readings.

That query also can be answered easily, if we store the 24 most recent stream elements. When a new stream element arrives, we can drop from the working store the 25th most recent element, since it will never again be needed (unless there is some other standing query that requires it)

Another query we might ask is the maximum temperature ever recorded by that sensor. We can answer this query by ~~mining data streams~~ simple summary: the maximum of all stream elements ever seen. It is not necessary to record the entire stream.

## MINING DATA STREAMS

When a new stream element arrives, we compare it with the stored maximum, and set the maximum to whichever is larger.

We can then answer the query by producing the current value of the maximum.

Similarly, if we want the average temperature over all time, we have only to record two values: the number of readings ever sent in the stream and the sum of those readings.

We can adjust these values easily each time a new reading arrives, and we can produce their quotient as the answer to the query

# MINING DATA STREAMS

## 2. Ad-hoc queries

The other form of query is ad-hoc, a question asked once about the current state of a stream or streams.

If we do not store all streams in their entirety, as normally we can not, then we cannot expect to answer arbitrary queries about streams.

If we have some idea what kind of queries will be asked through the ad-hoc query interface, Example 4.1. then we can prepare for them by storing appropriate parts or summaries of streams.

*MINING DATA STREAMS*

If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a sliding window of

## MINING DATA STREAMS

A **sliding window** can be the most recent n elements of a stream, for some n, or it can be all the elements that arrived within the last t time units, e.g., one day.

If we regard each stream element as a tuple, we can treat the **window** as a relation and query it with any SQL query.

of course the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

**Example:** web sites often like to report the number of unique users over the past month.

If we think of each login as a stream element, we can maintain a window that is all logins in the most recent month.

## MINING DATA STREAMS

We must associate the arrival time with each login, so we know when it no longer belongs to the window.

If we think of the **window** as a **relation** **Logins(name, time)**, then it is simple to

```
SELECT COUNT(DISTINCT(name))  
FROM Logins  
WHERE time >= t;
```

Here,  $t$  is a constant that represents the time one month before the current time.

Note that we must be able to maintain the entire stream of logins for the past month in working storage. However, for even the largest sites, that data is not more than a few terabytes, and so surely can be stored on disk.  $\square$

# MINING DATA STREAMS

**Issues in Stream Processing** Before proceeding to discuss algorithms, let us consider the constraints under which we work when dealing with streams.

First, streams often deliver elements very rapidly. We must process elements in real time, or we lose the opportunity to process them at all, without accessing the archival storage.

Thus, it often is important that the stream-processing algorithm is executed in main memory, without access to secondary storage or with only rare accesses to secondary storage.

*MINING DATA STREAMS*

Moreover, even when streams are “slow,” as

## MINING DATA STREAMS

Even if each stream by itself can be processed using a small amount of main memory, the requirements of all the streams together can easily exceed the amount of available main memory.

Thus, many problems about streaming data would be **easy to solve** if we had enough **memory**, but become rather hard and require the invention of new techniques in order to execute them at a realistic rate on a machine of realistic size.

Here are two generalizations about stream algorithms worth bearing in mind as you read through this chapter:

*MINING DATA STREAMS*

- often, it is much more efficient to get

## MINING DATA STREAMS

- A variety of techniques related to **hashing** turn out to be useful. Generally, these techniques introduce useful randomness into the algorithm's behavior, in order to produce an approximate answer that is very close to the true result.

## SAMPLING DATA IN A STREAM

As our first example of managing streaming data, we shall look at **extracting reliable samples from a stream**. As with many stream algorithms, the “trick” involves using hashing in a somewhat unusual way.

**A Motivating Example** The general problem we shall address is **selecting a subset of a stream so that we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole**.

If we know what queries are to be asked, then there are a number of methods that might work, but we are looking for a technique that will allow ad-hoc queries on the sample.

# MINING DATA STREAMS

## SAMPLING DATA IN A STREAM

Our running example is the following. A search engine receives a stream of queries, and it would like to study the behavior of typical users.

We assume the stream consists of tuples (user, query, time).

Suppose that we want to answer queries such as “what fraction of the typical user’s queries were repeated over the past month?”

Assume also that we wish to store only 1/10th of the stream elements.

## MINING DATA STREAMS

### SAMPLING DATA IN A STREAM

The obvious approach would be to generate a random number, say an integer from 0 to 9, in response to each search query.

Store the tuple if and only if the random number is 0. If we do so, each user has, on average, 1/10th of their queries stored.

Suppose a user has issued  $s$  search queries one time in the past month,  $d$  search queries twice, and no search queries more than twice.

If we have a 1/10th sample, of queries, we shall see in the sample for that user an expected  $s/10$  of the search queries issued once.

# MINING DATA STREAMS

## SAMPLING DATA IN A STREAM

Total searches:

Number of unique search queries : s

Number of repeated search queries: d

Our query is what is the fraction of repeated searches:

Exact answer is :  $d/(s+d)$

This answer will be obtained if we analyse the entire search queries ( $s+d$ ). But as analysing the entire stream of search queries is not possible , we take a sample and check whether we are able to answer the query.

We take  $1/10^{\text{th}}$  of the stream as a sample.

Unique searches

1. Probability(unique search queries in the sample)= $s*1/10=s/10$ :

# SAMPLING DATA IN A STREAM

## Repeated searches

2. Probability(repeated search queries appears exactly once in the sample)=**18d/100**

Reason: Say a query appears twice in stream

Probability(1st instance is included and 2<sup>nd</sup> instance is not included) = $1/10 * 9/10$

Probability(1st instance is not included and 2<sup>nd</sup> instance is included) = $9/10 * 1/10$

Either of these two scenarios may happen,

So, the probability(repeated search appearing exactly once)= $9/100 + 9/100 = 18/100$

## SAMPLING DATA IN A STREAM

3. Probability that repeated query appearing twice in the sample=Probability(1<sup>st</sup> instance appears in the (2<sup>nd</sup> instance appears in the sample))

$$=1/10 * 1/10$$

$$=1/100$$

For d queries, the probability of appearing twice =  $d/100$

Total probability=P(s queries appearing once)+P(d queries appearing twice)

$$=s/10+18d/100+d/100$$

$$=(10s+18d+d)/100$$

$$=(10s+19d)/100$$

# MINING DATA STREAMS

## SAMPLING DATA IN A STREAM

Fraction of queries appearing twice in the sample =

$$(d/100) / (10s+19d)/100$$

$$= d/100 * 100/(10s+19d)$$

$$= d/(10s+19d)$$

which is not the actual correct answer

$$d/(s+d)$$

So, this way of taking 1/10 of the queries as sample will not be useful to ~~answer the~~ <sup>MINING DATA STREAMS</sup> queries. We need a representative sample.

# MINING DATA STREAMS

## Obtaining a Representative Sample

Most of the queries about the statistics of typical users, cannot be answered by taking a sample of each user's search queries.

Thus, we must strive to pick 1/10th of the users, and take all their searches for the sample, while taking none of the searches from other users.

If we can store a list of all users, and whether or not they are in the sample, then we could do the following.

Each time a search query arrives MINING DATA STREAMS in the stream, we look up the user to see whether or not they are in the sample.

## MINING DATA STREAMS

If so, we add this search query to the sample, and if not, then not. However, if we have no record of ever having seen this user before, then we generate a random integer between 0 and 9.

If the number is 0, we add this user to our list with value “in,” and if the number is other than 0, we add the user with the value “out.”

That method works as long as we can afford to keep the list of all users and their in/out decision in main memory, because there isn’t time to go to disk for every search that arrives.

## MINING DATA STREAMS

If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not. Note we do not actually store the user in the bucket;

in fact, there is no data in the buckets at all. Effectively, we use the hash function as a randomnumber generator, with the important property that, when applied to the same user several times, we always get the same “random” number.

That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.

# MINING DATA STREAMS

## The General Sampling Problem

The running example is typical of the following general problem. Our stream consists of tuples with n components.

A subset of the components are the key components, on which the selection of the sample will be based.

In our running example, there are three components – user, query, and time – of which only user is in the key.

However, we could also take a sample of queries by making query be the key, or even take a sample of user-query pairs by making both those components form the key

# MINING DATA STREAMS

## The General Sampling Problem

To take a sample of size  $a/b$ , we hash the key value for each tuple to  $b$  buckets, and accept the tuple for the sample if the hash value is less than  $a$

### Varying the Sample Size

often, the sample will grow as more of the stream enters the system. In our running example, we retain all the search queries of the selected 1/10th of the users, forever.

As time goes on, more searches for the same users will be accumulated, and new users that are selected for the sample will appear in the stream.

## MINING DATA STREAMS

### Steps to vary sample size

1. Choose b: that is the total number of buckets you want to use eg:10
2. Choose a: This is how many of those buckets you want to accept items from.

If you want a sample size of 30%

Set a to 3(Reason: 30% of 10 is 3)

### Define hash function

A simple hash function might convert each item to a number between 0 and b-1.

**If  $\text{hash(item)} < a$  , accept it into sample.**

For 30% sample size, a=3 and b=10

So, accept items whose hash value is 0, 1 or 2.

## MATRIX Multiplication using Map Reduce

1. Natural Join
  2. Grouping and Aggregation

Consider 2 matrices:

$M(I, k, v)$  8

$$N(k, j, w)$$

$I$  - row index }  
 $k$  - column index } of matrix  $M$

$k$  - Row index  
 $\ell$  - Column index } of matrix  $N$

v - element of M

w - element of N

As M & N matrices have common attribute k, we can perform

## Natural join of M $\bowtie_k$ N

$$\therefore (I \wedge v) \vee (v \wedge J) \wedge (J \wedge w)$$

$\Rightarrow (I \ K \ J \ V \ W) \rightarrow 5$  tuple  
 (where the duplicate K is removed)

We can reduce 5-tuple to 4-tuple

After this, we perform grouping & aggregation to obtain the result matrix.

Example

Let  $A = \begin{pmatrix} 2 & 3 \\ 1 & 5 \end{pmatrix}$  &  $B = \begin{pmatrix} 6 & 7 \\ 8 & 9 \end{pmatrix}$

In general,

$$M = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad N = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$M \times N \Rightarrow \begin{pmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} & a_{11} \times b_{12} + a_{12} \times b_{22} \\ a_{21} \times b_{11} + a_{22} \times b_{21} & a_{21} \times b_{12} + a_{22} \times b_{22} \end{pmatrix}$$

Matrix  $\textcircled{A}$

i	(k)	v
1	1	2
1	2	3
2	1	4
2	2	5

Matrix  $\textcircled{B}$

(k)	j	w
1	1	6
1	2	7
2	1	8
2	2	9

Natural join

$A \bowtie B$

over  $K$

Map Phase

- 1,  $(A, 1, 2)$
- 2,  $(A, 1, 3)$
- 1,  $(A, 2, 4)$
- 2,  $(A, 2, 5)$

Map plan ↴

- 1,  $(B, 1, 6)$
- 1,  $(B, 2, 7)$
- 2,  $(B, 1, 8)$
- 2,  $(B, 2, 9)$

↓

Shuffle & Sort

1.  $\left[ (A, 1, 2) \xrightarrow{(A, 2, 4)} (B, \overbrace{1, 6}^{\rightarrow}, \overbrace{B, 2, 7}^{\rightarrow}) \right]$
2.  $\left[ (A, 1, 3) \xrightarrow{(A, 2, 5)} (B, 1, 8), (B, 2, 9) \right]$

Reducer 1  
(key 1)

- $((1, 1), 2 \times 6)$   
 $((1, 2), 2 \times 7)$   
 $((2, 1), 4 \times 6)$   
 $((2, 2), 4 \times 7)$

Reducer 2  
(key 2)

- $((1, 1) 3 \times 8)$   
 $((1, 2) 3 \times 9)$   
 $((2, 1) 5 \times 8)$   
 $((2, 2) 5 \times 9)$

## 2. Grouping & Aggregation

Map phase (1)

key	value	key	value
1, 1	12	1, 1	24
1, 2	14	1, 2	27
2, 1	24	2, 1	40
2, 2	28	2, 2	45

Shuffle & Sort

- $(1, 1) (12, 24)$   
 $(1, 2) (14, 27)$   
 $(2, 1) (24, 40)$   
 $(2, 2) (28, 45)$

Reducer

P.T.O

↓

(1,1) 36

(1,2) 41

(2,1) 64

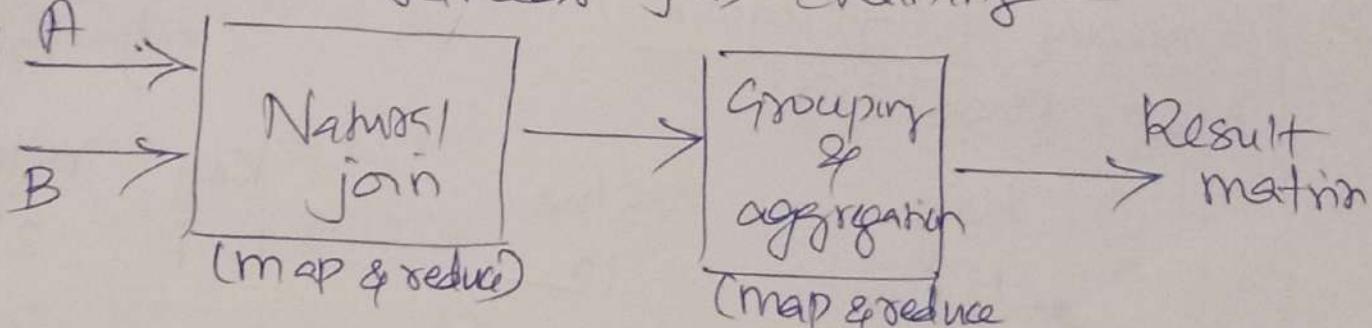
(2,2) 73

final  
O/P of the  
Reducer

$$\therefore \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} \times \begin{pmatrix} 6 & 7 \\ 8 & 9 \end{pmatrix} = \begin{pmatrix} 36 & 41 \\ 64 & 73 \end{pmatrix}$$

using 2 operations for Map Reduce  
(Natural join &  
Grouping Aggregation)

Linear job chaining



Matrix multiplication using single  
Step (1 map reduce job)

$$\text{Let } A = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} \text{ & } B = \begin{pmatrix} 6 & 7 \\ 8 & 9 \end{pmatrix}$$

Relation A

I	K	V
1	1	2
1	2	3
2	1	4
2	2	5

Mapper 1  $\downarrow (9, K)$

(1, K) : (A, 1, 2)

(1, K) : (A, 2, 3)

(2, K) : (A, 1, 4)

(2, K) : (A, 2, 5)

Relation B

K	J	V
1	1	6
1	2	7
2	1	8
2	2	9

Mapper 2  $\downarrow (K, j)$

(K, 1) : (B, 1, 6)

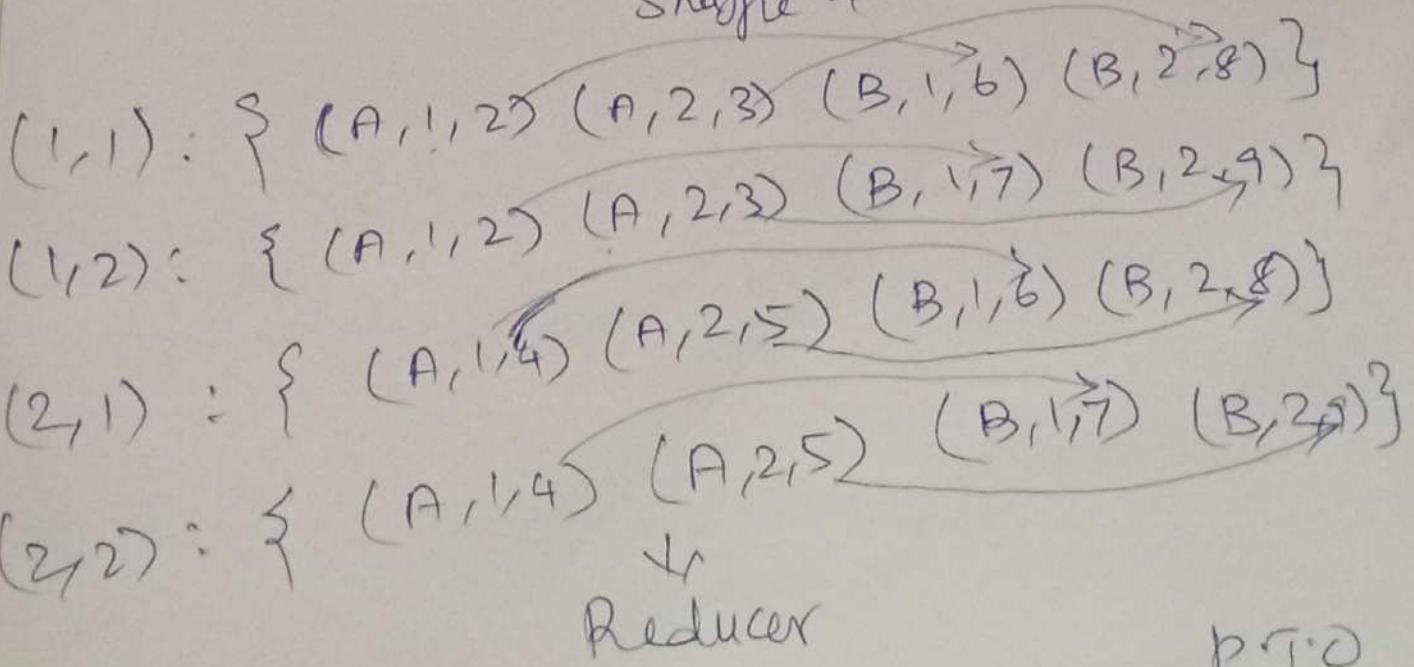
(K, 2) : (B, 1, 7)

(K, 1) : (B, 2, 8)

(K, 2) : (B, 2, 9)

↓

Shuffle & Sort



$$(1,1) : (2 \times 6 + 3 \times 8)$$

$$(1,2) : (2 \times 7 + 3 \times 9)$$

$$(2,1) : (4 \times 6 + 5 \times 8)$$

$$(2,2) : (4 \times 7 + 5 \times 9)$$

↓ trial of form  
reducer

$$(1,1) : (36)$$

$$(1,2) : (41)$$

$$(2,1) : (64)$$

$$(2,2) : (73)$$

Communication cost Model

- An algorithm is implemented as an acyclic network of tasks.
- The tasks may be map tasks feeding reduce tasks (or) cascaded jobs (or) any workflow.
- \* Communication cost of an algorithm is the sum of the communication cost of all tasks involved in it.
- Usually, communication cost of a task is the size of the input to the task. It is measured in bytes (or) tuples.
- It is a way to measure the efficiency of an algorithm.

Importance of communication cost can be justified as below:

- (1) Let the interconnect speed of the cluster be 1 gigabit /sec. It seems to be a lot but slow compared with the speed @ which the processor executes instructions.

- will be more
- Competition for interconnect speed among nodes that tend to communicate @ same time.
    - i.e) time required to communicate is  $>$  than time required to operate on data.
  - (2). Even though a task has its chunk locally, still it takes time to move the chunk from disk to memory.
- \* This implies that communication cost is a dominant cost. The reasons for why we consider only the input size & not the output size is that:
- (a) Output of one task will be the input of another task, so the output size will be accounted while measuring input size. So, except the final output, we need not measure any other output.
  - (b) Massive outputs are summarized aggregated in some way.

Let us evaluate the communication cost of simple join algorithm.

$$R(A, B) \bowtie_B S(B, C)$$

Let  $\alpha$  be the size of relation R.  
Let  $\beta$  be the size of relation S.

$$\therefore \text{Sum of the communication cost for map tasks} = \gamma + s$$

as we need to send  $\gamma$  tuples &  $s$  tuples over the network to map nodes.

- Each map task will hold its chunk on disk & then transfers it to main memory, convert the I/P tuples to (key-value) pairs!

$$\therefore \text{Computation cost} < \text{Communication cost.}$$

- Each key-value pair must be distributed to Reduce tasks @ different nodes. So, Interconnect cluster communication is required.

So, Communication cost of join algorithm is  $O(\gamma+s)$ .

### Wall-clock Time

- We must be aware of wall-clock time.
- Some may think, we can reduce the communication cost by assigning all works to one task.
- But, for such algorithms, where all works are carried out by one task, the wall-clock time will increase.
- So, the property of an algorithm should be like: Fairly divide the work among the tasks, so that wall-clock time will be as small as it could be given a good number of nodes.

Now, let us find the communication cost for a Multiway join operation.

\* To perform a multiway join of 3 relations  $R(A, B)$ ,  $S(B, C)$  &  $T(C, D)$  using a single join with 2 hash functions  $h$  and  $g$ .

\* We use  $k$  reducers, where,  $K = b \star c$

#### 1. Setup & Hash Functions

\* Let  $h(B)$  be a hash function that hashes  $B$ -values into  $b$  buckets.

\* Let  $g(C)$  be a hash function that hashes  $C$ -values into  $c$  buckets.

\*  $K = \underline{b} \star \underline{c}$ , meaning that the total no. of reducers is  $K$ .

#### 2. Distributing data to Reducers

\* Each reducer is identified by a pair of indices  $(i, j)$ , when  $0 \leq i \leq b$  &  $0 \leq j \leq c$

\* Reducer  $(i, j)$  will handle those tuples for which  $h(B) = i$  &  $g(C) = j$

### 3. Mapping phase

#### \* Relation $R(A, B)$

1. For each tuple  $(a, b) \in R$ ,  
compute  $h(b) = i$
2. Send  $(a, b)$  to all reducers with  
indices  $(i, j)$  for all  $0 \leq j \leq c$ .

#### \* Relation $S(B, C)$

1. For each tuple  $(b, c) \in S$ ,  
compute  $f(b) = i$  &  
 $g(c) = j$
2. Send  $(b, c)$  to all reducers with index  $(i, j)$

#### \* Relation $T(C, D)$

1. For each tuple  $(c, d) \in T$ ,  
compute  $g(c) = j$
2. Send  $(c, d)$  to all reducers with  
indices  $(i, j)$  for all  $0 \leq i \leq b$

### 4. Reducer Phase

\* Each Reducer  $(i, j)$  will now have:

1. Tuples  $(a, b)$  from  $R$  where  $h(b) = i$
2. Tuples  $(b, c)$  from  $S$  where  $h(b) = i$  &  
 $g(c) = j$
3. Tuples  $(c, d)$  from  $T$  where  $g(c) = j$ .

→ The reducer performs the join locally by :

1. Joining  $R(A, B)$  &  $S(B, C)$  on  $B$  to produce tuples  $(A, B, C)$
2. Joining the result  $(A, B, C)$  with  $T(C, D)$  on  $C$  to produce the final tuples  $(A, B, C, D)$ .

Final output :

Collect the output from all the reducers which are the resulting tuples obtained as a result of multiway join of 3 relations  $R(A, B) \bowtie_B S(B, C) \bowtie_C T(C, D)$ .

Let us consider an example.

Consider 3 relations

$$R = \{(a_1, b_1), (a_2, b_2)\}$$

$$S = \{(b_1, c_1), (b_2, c_2)\}$$

$$T = \{(c_1, d_1), (c_2, d_2)\}$$

$$\text{Let } b=2, \quad c=2, \quad K = \frac{b \times c}{2 \times 2} = 4$$

## Step 1. Hash functions

$$h(b1) = 0$$

$$h(b2) = 1$$

$$g(c1) = 0$$

$$g(c2) = 1$$

## Step 2. Mapping phase

### • For R:

Tuple:  $(a1, b1)$

$$h(b1) = 0 \text{ (From step 1)}$$

$\therefore (a1, b1)$  goes to Reducers whose  
q index is 0 & g index is 0 & 1.

$$\therefore (a1, b1) \rightarrow \text{Reducers } (0, 0) (0, 1)$$

Tuple:  $(a2, b2)$

$$h(b2) = 1 \text{ (From step 1)}$$

$\therefore (a2, b2)$  goes to Reducers whose  
q index is 1 & g index is 0 & 1

$$\therefore (a2, b2) \rightarrow \text{Reducers } (1, 0) (1, 1)$$

### • For S:

Tuple:  $(b1, c1)$

$$h(b1) = 0 \text{ & } g(c1) = 0 \text{ (From step 1)}$$

$$\therefore (b1, c1) \rightarrow \text{Reducer } (0, 0)$$

Tuple:  $(b2, c2)$

$$h(b2) = 1 \text{ & } g(c2) = 1 \text{ (From step 1)}$$

$$\therefore (b2, c2) \rightarrow \text{Reducer } (1, 1)$$

Tuple:  $(c_1, d_1)$

~~$g(c_1) = 0$~~  (from step 1)  
 $(c_1, d_1)$  is sent to reducer with  $j$  index 0  
 $\therefore (c_1, d_1) \rightarrow \text{Reducer } (0, 0) (1, 0)$

Tuple:  $(c_2, d_2)$

$g(c_2) = 1$  (From step 1)  
 $\therefore (c_2, d_2)$  is sent to reducers with  
 $j$  index 1  
 $\therefore (c_2, d_2) \rightarrow \text{Reducer } (0, 1) (1, 1)$

Consolidated Data:

Tuple	Reducer
$(a_1, b_1)$	$(0, 0) (0, 1)$
$(a_2, b_2)$	$(1, 0) (1, 1)$
$(b_1, c_1)$	$(0, 0)$
$(b_2, c_2)$	$(1, 1)$
$(c_1, d_1)$	$(0, 0) (1, 0)$
$(c_2, d_2)$	$(0, 1) (1, 1)$

Step 3 from table / this

$\therefore$  Reducer  $(0, 0)$  receives the tuples:

$(a_1, b_1) (b_1, c_1) (c_1, d_1)$

Performs the join

$(a_1 b_1) \bowtie (b_1 c_1) \Rightarrow (a_1 b_1 c_1)$

$(a_1 b_1 c_1) \bowtie (c_1 d_1) \Rightarrow (a_1 b_1 c_1 d_1)$

∴ from the table,

Reducer (1,1) receives the tuples:

$(a_2, b_2)$   $(b_2, c_2)$   $(c_2, d_2)$

Performs the join

$(a_2, b_2) \bowtie (b_2, c_2) \Rightarrow (a_2, b_2, c_2)$

$(a_2, b_2, c_2) \bowtie (c_2, d_2) \Rightarrow (a_2, b_2, c_2, d_2)$

Let's see the detailed calculation of communication cost:

1. Relation  $R(A, B)$

\* Each tuple  $(a, b) \in R$  is sent to  $\leq$  reducers, bcos it needs to be paired with all possible  $c$  values that might join with  $S(B, C) \& T(C, D)$ .

\* If  $R$  has  $n_R$  tuples, total data to be sent is  $n_R \times c$  tuples.

2. Relation  $S(B, C)$

\* Each tuple  $(b, c) \in S$  is sent to 1 reducer only.

\* If  $S$  has  $n_S$  tuples, total data sent is  $n_S \times 1$  tuples.

### 3. Relation T (c, d)

- Each tuple  $(c, d) \in T$  is sent to b reducers.
- If T has  $n_T$  tuples, total data to be sent is  $n_T \times b$  tuples.

The total communication cost (no. of tuples sent) is the sum of the data sent for each relation.

$$\therefore \text{Total Commn. cost} = n_R \times c + n_S \times 1 + n_T \times b$$

Suppose,  $n_R = 1000$ ,  $n_S = 2000$  &  $n_T = 1500$   
 $b = 2$ ,  $c = \frac{5}{2}$  & hence  $k = b \times c = 10$   
 $(2 \times 5 = 10)$

$$\begin{aligned}\text{Total Commn. cost} &= n_R \times c + n_S + n_T \times b \\ &= (1000 \times \frac{5}{2}) + (2000) + (1500 \times 2) \\ &= (5000) + (2000) + (3000) \\ &= 10,000 \text{ tuples}\end{aligned}$$

$\therefore$  10,000 tuples needs to be communicated to the nodes over the network to perform the multiway join of R  $\bowtie$  S  $\bowtie$  T.

# INT404R01 – BIG DATA ANALYTICS

*IV B.Tech CSE 'A' / VII SEMESTER*

Unit 2 – Counting Distinct Elements in a Stream

Dr.M.Devi Sri Nandhini

## **Counting Distinct Elements in a Stream**

Suppose stream elements are chosen from some universal set.

We would like to know how many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past.

Example : Consider a Web site gathering statistics on how many unique users it has seen in each given month.

The universal set is the set of logins for that site, and a stream element is generated each time someone logs in. This measure is appropriate for a site like Amazon, where the typical user logs in with their unique login name.

A similar problem is a Web site like Google that does not require login to issue a search query, and may be able to identify users only by the IP address from which they send the query.

There are about 4 billion IP addresses, 2 sequences of four 8-bit bytes will serve as the universal set in this case.

The obvious way to solve the problem is to keep in main memory a list of all the elements seen so far in the stream. Keep them in an efficient search structure such as a hash table or search tree, so one can quickly add new elements and check whether or not the element that just arrived on the stream was already seen.

As long as the number of distinct elements is not too great, this structure can fit in main memory.

However, if the number of distinct elements is too great, or if there are too many streams that need to be processed at once then we cannot store the needed data in main memory.

## The Flajolet-Martin Algorithm

It is possible to estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long.

The length of the bit-string must be sufficient that there are more possible results of the hash function than there are elements of the universal set.

The Flajolet-Martin algorithm is a powerful probabilistic technique for estimating the number of distinct elements in a data stream. Here's a concise explanation of how it works, followed by a simple example to illustrate the process.

## **Key Concepts**

**Hash Functions:** The algorithm uses a hash function to convert each element into a binary string.

**Trailing Zeros:** The focus is on counting the number of trailing zeros in these binary representations, which provides insight into the distinctiveness of the hashed values.

**Estimate Calculation:** The algorithm keeps track of the maximum number of trailing zeros observed, which is then used to estimate the total number of distinct elements.

## **Steps of the Algorithm**

**Hashing Elements:** For each element in the stream, compute its hash value using a hash function.

**Counting Trailing Zeros:** Convert the hash value to binary and count the number of trailing zeros.

## Track Maximum Zeros: Maintain the maximum count of trailing zeros observed from all hashed values.

4. Estimate Distinct Count: The estimate  $D$  of the number of distinct elements is given by  $D = 2^R$ , where  $R$  is the maximum number of trailing zeros.

### Example

Consider a small data stream:  $\{2, 3, 3, 4, 2, 5\}$ .

#### Step 1: Choose a Hash Function

Assume the hash function:

$$h(x) = (3x + 7) \mod 16$$

#### Step 2: Compute Hash Values

Compute the hash values for each element:

- $h(2) = (3 \times 2 + 7) \mod 16 = 13 \rightarrow$  Binary: 1101 (0 trailing zeros)
- $h(3) = (3 \times 3 + 7) \mod 16 = 16 \mod 16 = 0 \rightarrow$  Binary: 0000 (4 trailing zeros)

- $h(5) = (3 \times 5 + 7) \bmod 16 = 22 \bmod 16 = 6 \rightarrow$  Binary: 0110 (1 trailing zero)

### Step 3: Track Maximum Number of Trailing Zeros

From the computed hash values:

- $h(2) = 13 \rightarrow$  0 trailing zeros
- $h(3) = 0 \rightarrow$  4 trailing zeros
- $h(4) = 3 \rightarrow$  0 trailing zeros
- $h(5) = 6 \rightarrow$  1 trailing zero

The maximum number of trailing zeros is 4.

## Step 4: Estimate the Number of Distinct Elements

Using the maximum trailing zeros  $R = 4$ :

$$D = 2^4 = 16$$

In this example, the estimated distinct count (16) is higher than the actual distinct elements (4). This discrepancy illustrates that the Flajolet-Martin algorithm provides an approximation, and its accuracy can improve with better hash functions and multiple iterations. The key takeaway is the algorithm's ability to estimate distinct elements efficiently without needing to store all elements explicitly.

Whenever we apply a hash function  $h$  to a stream element  $a$ , the bit string  $h(a)$  will end in some number of 0's, possibly none. Call this number the *tail length* for  $a$  and  $h$ . Let  $R$  be the maximum tail length of any  $a$  seen so far in the stream. Then we shall use estimate  $2^R$  for the number of distinct elements seen in the stream.

This estimate makes intuitive sense. The probability that a given stream element  $a$  has  $h(a)$  ending in at least  $r$  0's is  $2^{-r}$ . Suppose there are  $m$  distinct elements in the stream. Then the probability that none of them has tail length at least  $r$  is  $(1 - 2^{-r})^m$ . This sort of expression should be familiar by now.

We can rewrite it as  $((1 - 2^{-r})^{2^r})^{m2^{-r}}$ . Assuming  $r$  is reasonably large, the inner expression is of the form  $(1 - \epsilon)^{1/\epsilon}$ , which is approximately  $1/e$ . Thus, the probability of not finding a stream element with as many as  $r$  0's at the end of its hash value is  $e^{-m2^{-r}}$ . We can conclude:

1. If  $m$  is much larger than  $2^r$ , then the probability that we shall find a tail of length at least  $r$  approaches 1.
2. If  $m$  is much less than  $2^r$ , then the probability of finding a tail length at least  $r$  approaches 0.

We conclude from these two points that the proposed estimate of  $m$ , which is  $2^R$  (recall  $R$  is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

1. If  $m$  is much larger than  $2^r$ , then the probability that we shall find a tail of length at least  $r$  approaches 1.
2. If  $m$  is much less than  $2^r$ , then the probability of finding a tail length at least  $r$  approaches 0.

We conclude from these two points that the proposed estimate of  $m$ , which is  $2^R$  (recall  $R$  is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

## Combining Estimates

Unfortunately, there is a trap regarding the strategy for combining the estimates of  $m$ , the number of distinct elements, that we obtain by using many different hash functions.

Our first assumption would be that if we take the average of the values  $2^R$  that we get from each hash function, we shall get a value that approaches the true  $m$ , the more hash functions we use.

However, that is not the case, and the reason has to do with the influence an overestimate has on the average.

One way is to take the average of all estimates. Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of  $2R$ , so the worry described above for the average should not carry over to the median.

## Space Requirements

Observe that as we read the stream it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function;

this integer records the largest tail length seen so far for that hash function and any stream element.

If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate.

Only if we are trying to process many streams at the same time would main memory constrain the number of hash functions we could associate with any one stream.

In practice, the time it takes to compute hash values for each stream element would be the more significant limitation on the number of hash functions we use.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 2

TOPIC:ESTIMATING MOMENTS

Handled by

Dr.M.Devi Sri Nandhini

AP III/School of Computing

## ESTIMATING MOMENTS

We consider a generalization of the problem of counting distinct elements in a stream. The problem, called computing “moments,” involves the distribution of frequencies of different elements in the stream.

Moments are the statistical properties that deal with the frequency distribution pf elements in a stream.

Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the  $i$ th element for any  $i$ . Let  $m_i$  be the number of occurrences of the  $i$ th element for any  $i$ . Then the  $k$ th-order moment (or just  $k$ th moment) of the stream is the sum over all  $i$  of  $(m_i)^k$ .

# ESTIMATING MOMENTS

Types of Moments:

- **Zeroth Moment ( $M_0$ ):** The zeroth moment is simply the **count of elements** in the stream. It tells you the total number of items, regardless of their value.
- **First Moment ( $M_1$ ):** The first moment is the **sum of the values** of elements in the stream. It's also known as the "mean" when you divide it by the number of elements.
- **Second Moment ( $M_2$ ):** The second moment(Surprise number) is related to the **variance** of the data. It's the sum of the squares of the values, and it helps measure how much the data varies from the mean.
- **Higher Moments ( $M_3, M_4, \text{ etc.}$ ):** Higher moments provide more detailed characteristics of the data's distribution, such as skewness ( $M_3$ ) and kurtosis ( $M_4$ ).

## ESTIMATING MOMENTS

### Formula for the Second Moment

Given a data stream consisting of elements  $a_1, a_2, \dots, a_n$ , let  $f_i$  represent the frequency of element  $i$  in the stream. The second moment  $M_2$  is calculated as:

$$M_2 = \sum_i f_i^2$$

Where:

- $f_i$  is the number of times the element  $i$  appears in the stream.

# ESTIMATING MOMENTS

## Example:

Consider a stream of elements:  $\{1, 2, 1, 3, 1, 2\}$ .

- $f_1 = 3$  (element 1 appears 3 times),
- $f_2 = 2$  (element 2 appears 2 times),
- $f_3 = 1$  (element 3 appears 1 time).

Now, applying the formula for the second moment:

$$M_2 = f_1^2 + f_2^2 + f_3^2 = 3^2 + 2^2 + 1^2 = 9 + 4 + 1 = 14$$

So, the second moment  $M_2 = 14$ . 

## ESTIMATING MOMENTS

The second moment is the sum of the squares of the  $m_i$ 's. It is sometimes called the *surprise number*, since it measures how uneven the distribution of elements in the stream is. To see the distinction, suppose we have a stream of length 100, in which eleven different elements appear. The most even distribution of these eleven elements would have one appearing 10 times and the other ten appearing 9 times each. In this case, the surprise number is  $10^2 + 10 \times 9^2 = 910$ . At the other extreme, one of the eleven elements could appear 90 times and the other ten appear 1 time each. Then, the surprise number would be  $90^2 + 10 \times 1^2 = 8110$ .  $\square$

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

In real applications (like network traffic or social media data), the stream can be huge.

It's impractical to store all the data or keep track of every item's count, especially in a **single pass**.

That's where the AMS algorithm comes in: it estimates the second moment without needing to store the entire stream.

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

## 1. Random Sampling:

- As the stream comes in, instead of keeping track of every element, we randomly pick one element. Let's say we're at time  $t$ , and we have seen  $t$  elements in the stream so far. We pick an element with a probability of  $1/t$  — meaning as the stream grows, each element gets a smaller chance of being picked.

## 2. Count the Frequency:

- Once we've picked an element, we count how many times that element appears as the stream continues. So, we track just **one item's frequency** (not all of them).



# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

## 3. Compute an Estimate:

- We take the frequency of this randomly picked element, call it  $f_j$ , and use it to compute an estimate of the second moment. The formula is:

$$\text{estimate} = (n/f_j)^2$$

where  $n$  is the total number of elements in the stream so far, and  $f_j$  is how many times the chosen element has appeared.

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

## 4. Repeat Multiple Times:

- Since picking one element might give us a rough estimate, the AMS algorithm runs this process multiple times with different random samples. After running it, say,  $k$  times, it takes the average of all these estimates.

## 5. Final Estimate:

- The final estimate of the second moment is the average of these multiple estimates. This helps smooth out any errors from random sampling.

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

- The algorithm gives an **unbiased estimate**, meaning the average result from multiple samples will get close to the true second moment.
- It's memory-efficient because it doesn't store every item in the stream — it only tracks the frequency of a randomly selected item each time.

Example

a, b, a, c, b, a, c, c

- At time  $t = 8$ , we have the stream: a, b, a, c, b, a, c, c.
- We randomly pick an element at some point, let's say we picked a.
- We then count how many times a appeared — in this case, it appeared 3 times.
- The total number of elements is  $n = 8$ , and the frequency of a is  $f_a = 3$ .

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

Using the formula:

$$\text{estimate} = \left(\frac{8}{3}\right)^2 = (2.67)^2 \approx 7.1$$

We repeat this with different elements, compute estimates, and average them to get a final approximation of the second moment.

- The AMS algorithm **samples** the stream, so you don't need to store the entire data.
- It gives a **rough estimate** of the second moment (how often items repeat).
- It's **efficient** and works well for large streams using small memory.

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

① Suppose the stream is  $a, b, c, b, d, a, c, d, a, b, d, c, a, g, b$ .  
The len of stream is  $n=15$ . compute an estimate of second moment by randomly picking three elements using AMS algo.

Pick three elements,  $x_1 = 3^{\text{rd}}$  element

$x_1.\text{element} = c$

finally

$x_1.\text{value} = 1$

$x_1.\text{value} = 3$

$x_2 = 8^{\text{th}}$  ele,  $x_2.\text{element} = d$ ,  $x_2.\text{value} = 1$ , finally  $x_2.\text{value} = 2$ .

$x_3 = 13^{\text{th}}$  ele,  $x_3.\text{element} = a$ ,  $x_3.\text{value} = 1$ , finally  $x_3.\text{value} = 2$ .

Estimates

$$F_2(x_1) = n * (2 * x_1.\text{value} - 1) = 15 * (2 * 3 - 1) = 75$$

$$F_2(x_2) = 15 * (2 * 2 - 1) = 45$$

$$F_2(x_3) = 15 * (2 * 2 - 1) = 45$$

$$\text{Avg}(F_1, F_2, F_3) = \frac{75 + 45 + 45}{3} = 55$$

actual estimation,  $F_2 = f_a^2 + f_b^2 + f_c^2 + f_d^2 = 5^2 + 4^2 + 3^2 + 3^2 = 59$ .

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

For now, let us assume that a stream has a particular length  $n$ . We shall show how to deal with growing streams in the next section. Suppose we do not have enough space to count all the  $m_i$ 's for all the elements of the stream. We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more accurate the estimate will be. We compute some number of *variables*. For each variable  $X$ , we store:

1. A particular element of the universal set, which we refer to as  $X.\text{element}$ , and
2. An integer  $X.\text{value}$ , which is the *value* of the variable. To determine the value of a variable  $X$ , we choose a position in the stream between 1 and  $n$ , uniformly and at random. Set  $X.\text{element}$  to be the element found there, and initialize  $X.\text{value}$  to 1. As we read the stream, add 1 to  $X.\text{value}$  each time we encounter another occurrence of  $X.\text{element}$ .

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

**Example 4.7:** Suppose the stream is  $a, b, c, b, d, a, c, d, a, b, d, c, a, a, b$ . The length of the stream is  $n = 15$ . Since  $a$  appears 5 times,  $b$  appears 4 times, and  $c$  and  $d$  appear three times each, the second moment for the stream is  $5^2 + 4^2 + 3^2 + 3^2 = 59$ . Suppose we keep three variables,  $X_1$ ,  $X_2$ , and  $X_3$ . Also,

assume that at “random” we pick the 3rd, 8th, and 13th positions to define these three variables.

When we reach position 3, we find element  $c$ , so we set  $X_1.\text{element} = c$  and  $X_1.\text{value} = 1$ . Position 4 holds  $b$ , so we do not change  $X_1$ . Likewise, nothing happens at positions 5 or 6. At position 7, we see  $c$  again, so we set  $X_1.\text{value} = 2$ .

At position 8 we find  $d$ , and so set  $X_2.\text{element} = d$  and  $X_2.\text{value} = 1$ . Positions 9 and 10 hold  $a$  and  $b$ , so they do not affect  $X_1$  or  $X_2$ . Position 11 holds  $d$  so we set  $X_2.\text{value} = 2$ , and position 12 holds  $c$  so we set  $X_1.\text{value} = 3$ . At position 13, we find element  $a$ , and so set  $X_3.\text{element} = a$  and  $X_3.\text{value} = 1$ . Then, at position 14 we see another  $a$  and so set  $X_3.\text{value} = 2$ . Position 15, with element  $b$  does not affect any of the variables, so we are done, with final values  $X_1.\text{value} = 3$  and  $X_2.\text{value} = X_3.\text{value} = 2$ .  $\square$

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

We can derive an estimate of the second moment from any variable  $X$ . This estimate is  $n(2X.value - 1)$ .

**Example 4.8:** Consider the three variables from Example 4.7. From  $X_1$  we derive the estimate  $n(2X_1.value - 1) = 15 \times (2 \times 3 - 1) = 75$ . The other two variables,  $X_2$  and  $X_3$ , each have value 2 at the end, so their estimates are  $15 \times (2 \times 2 - 1) = 45$ . Recall that the true value of the second moment for this stream is 59. On the other hand, the average of the three estimates is 55, a fairly close approximation.  $\square$

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

We can prove that the expected value of any variable constructed as in Section 4.5.2 is the second moment of the stream from which it is constructed. Some notation will make the argument easier to follow. Let  $e(i)$  be the stream element that appears at position  $i$  in the stream, and let  $c(i)$  be the number of times element  $e(i)$  appears in the stream among positions  $i, i + 1, \dots, n$ .

**Example 4.9:** Consider the stream of Example 4.7.  $e(6) = a$ , since the 6th position holds  $a$ . Also,  $c(6) = 4$ , since  $a$  appears at positions 9, 13, and 14, as well as at position 6. Note that  $a$  also appears at position 1, but that fact does not contribute to  $c(6)$ .  $\square$

The expected value of  $n(2X.value - 1)$  is the average over all positions  $i$  between 1 and  $n$  of  $n(2c(i) - 1)$ , that is

$$E(n(2X.value - 1)) = \frac{1}{n} \sum_{i=1}^n n(2c(i) - 1)$$

We can simplify the above by canceling factors  $1/n$  and  $n$ , to get

$$E(n(2X.value - 1)) = \sum_{i=1}^n (2c(i) - 1)$$

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

The term for the last position in which  $a$  appears must be  $2 \times 1 - 1 = 1$ . The term for the next-to-last position in which  $a$  appears is  $2 \times 2 - 1 = 3$ . The positions with  $a$  before that yield terms 5, 7, and so on, up to  $2m_a - 1$ , which is the term for the first position in which  $a$  appears. That is, the formula for the expected value of  $2X.value - 1$  can be written:

$$E(n(2X.value - 1)) = \sum_a 1 + 3 + 5 + \cdots + (2m_a - 1)$$

Note that  $1 + 3 + 5 + \cdots + (2m_a - 1) = (m_a)^2$ . The proof is an easy induction on the number of terms in the sum. Thus,  $E(n(2X.value - 1)) = \sum_a (m_a)^2$ , which is the definition of the second moment.

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

## Higher order moments

$$\text{for } k^{\text{th}} \text{ moment} = v^k - (v-1)^k = 25 + 16 + 9 + 9$$

$$F_3 = v^3 - (v-1)^3 = v^3 - v^3 + 3v^2 - 3v + 1 \Rightarrow (3v^2 - 3v + 1)$$

$$F_4 = v^4 - (v-1)^4 \quad [\because (a-b)^4 = a^4 - 4a^2b + 6a^2b^2 - 4ab^2 + b^4]$$

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

## Dealing with Infinite Streams

The number of elements in the stream is not fixed. It keeps on growing. So, your stream is going to be an infinite one. How to count the number of elements? No problem , just keep track of the count at time points.

In AMS, If our estimate is just based on 3 random picks and all of them are early picks, it will not be representative of the stream. Similarly, picking elements from the later positions of the stream is also not correct.

So, we need to pick elements in a uniform fashion.

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

## Dealing with Infinite Streams

The proper technique is to maintain as many variables as we can store at all times, and to throw some out as the stream grows.

The discarded variables are replaced by new ones, in such a way that at all times, the probability of picking any one position for a variable is the same as that of picking any other position.

Suppose we have space to store  $s$  variables.

Then the first  $s$  positions of the stream are each picked as the position of one of the  $s$  variables.

# The Alon-Matias-Szegedy Algorithm for Second Moments (AMS algorithm)

## Dealing with Infinite Streams

Inductively, suppose we have seen  $n$  stream elements, and the probability of any particular position being the position of a variable is uniform, that is  $s/n$ . When the  $(n+1)$ st element arrives, pick that position with probability  $s/(n+1)$ .

If not picked, then the  $s$  variables keep their same positions. However, if the  $(n + 1)$ st position is picked, then throw out one of the current  $s$  variables, with equal probability.

Replace the one discarded by a new variable whose element is the one at position  $n + 1$  and whose value is 1.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 2

TOPIC:COUNTING ONES IN A WINDOW(DGIM ALGORITHM)

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

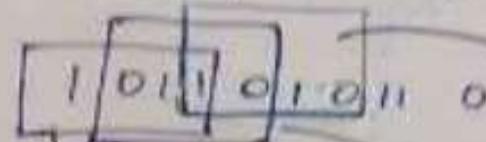
# COUNTING ONES IN A WINDOW

A window is a subsequence of a larger stream.

Counting

Subsequence of a larger stream

$$w = 4$$



no. of ones in  
window = 3

② ②

1<sup>st</sup> store 1<sup>st</sup> window bits in MM

Then 2<sup>nd</sup> window bits in MM

Time consuming

Each time count the no. of ones  
in each window  $\rightarrow$  From scratch

Use sliding window.

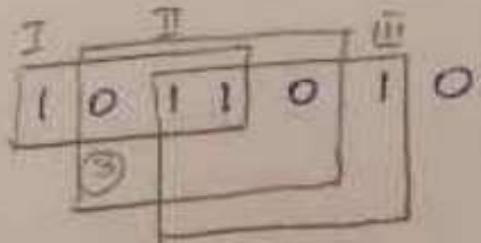
↓

Use a counter

Just subtract the effect of outgoing element  
and Add the effect of incoming element.

# COUNTING ONES IN A WINDOW

1<sup>st</sup> window  
(normal)



$$\boxed{3} \leftarrow 3$$

Counter  
in MM

$$\begin{array}{l} \text{I} \quad 3 \\ \text{II} \quad 3 - 1 + 0 = 2 \\ \text{III} \quad 2 - 0 + 1 = 3 \end{array}$$

Cost of exact ones

↳ Computational Resources  
Time      memory

Time complexity

Counting it's from scratch for every window, time complexity ↑

$$TC = O(w)$$

For all possible windows:  $TC = O(w * (n-w+1))$

Space complexity

Storing every window requires more space: Space complexity=O(W). If using counter, Space Complexity=O(1)

# COUNTING ONES IN A WINDOW

## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

Definition: It is a technique to count 1's in windows using buckets.  
Approximate count

Probability of 50% error

Problem: Sliding window, size N

↓  
Can't accommodate in main memory

Estimate of 1's in it.

Key Idea: Summarize the data using bucket

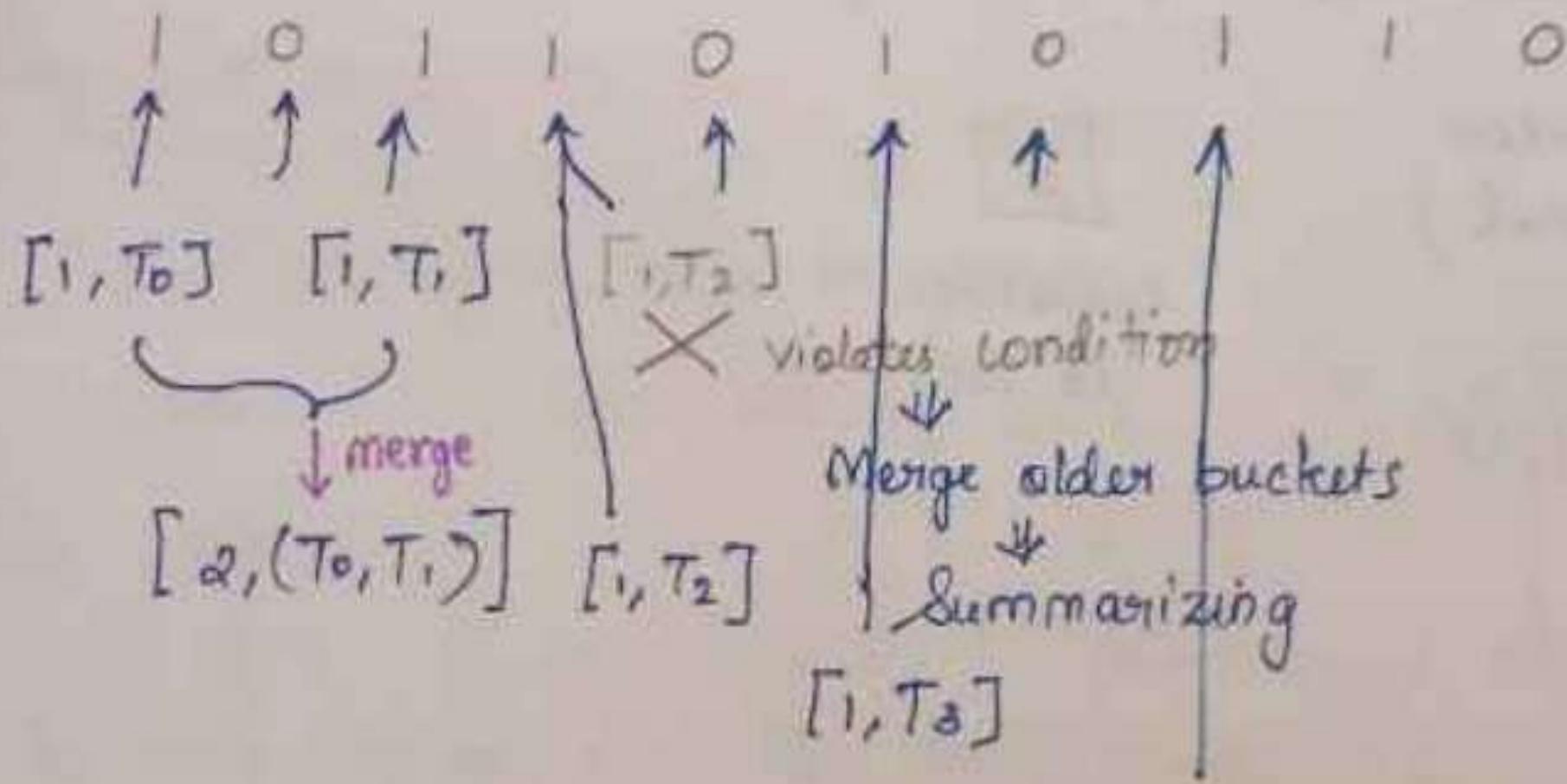
Working:

When you see a 1, add it to the bucket along with a timestamp

There can't be more than two buckets of the same size.

# COUNTING ONES IN A WINDOW

## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)



$[3, (T_0-T_2)]$ ,  $[1, T_3]$   $[1, T_4]$

## COUNTING ONES IN A WINDOW

### DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

#### ① Bucket Representation

When you happen to see a 1, add it to a bucket with a timestamp.  
Each bucket has a size that is the no. of ones it represents.

Size is always  $2^k$ .

#### ② Merging the buckets

As new bits arrive, the algorithm creates a new bucket for each 1. If there are more than two buckets of same size, it merges the oldest two buckets.

Merging ensures that the no. of buckets is manageable.

## COUNTING ONES IN A WINDOW

### DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

③ Maintaining the Window

As the window slides forward, the older bits move out

of the window.

④ Counting ones

To estimate the no. of ones in current window of size  $N$ ,

you sum up the sizes of the buckets that are fully within the

window. For those buckets that are partially in the window take

half the size.

# COUNTING ONES IN A WINDOW

## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

1. Given  
Count the no. of ones in the last 5 bits using

Position	Bit	Bucket
1	1	$[1, T_0]$
2	0	$[1, T_0]$
3	1	$[1, T_0] \cup [1, T_1]$
4	1	$[2, (T_0, T_1)] \cup [1, T_2]$
5	0	$[2, (T_0 - T_1)] \cup [1, T_2]$
6	1	$[2, (T_0 - T_1)] \cup [1, T_2] \cup [1, T_3]$
7	0	$[2, (T_0 - T_1)] \cup [1, T_2] \cup [1, T_3]$
8	1	$[3, (T_0, T_2)] \cup [1, T_3] \cup [1, T_4]$
9	1	$[4, (T_0 - T_3)] \cup [1, T_4] \cup [1, T_5]$
10	0	$[4, (T_0 - T_3)] \cup [1, T_4] \cup [1, T_5]$

# COUNTING ONES IN A WINDOW

DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

In the window  $T_3 - T_5$

Buckets that fit completely:

Last 5 bits  $\Rightarrow 13 - 15$

$[1, T_4] [1, T_5] \rightarrow 1 + 1$

Buckets filled partially

$[1, (T_0 - T_3)] \rightarrow$

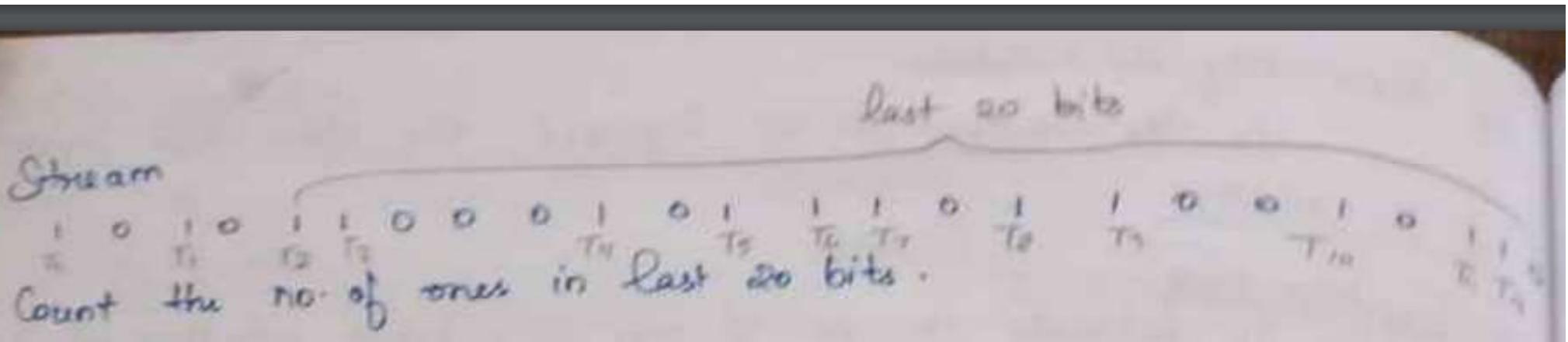


only  $T_3$   
included  
in window

Count of ones in Last 5 bits  $= 1 + 1 + 1 = 3$

# COUNTING ONES IN A WINDOW

## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)



101011000101110110010110 –  
count the number of ones in the last 20 bits  
using DGIM bucket representation

# COUNTING ONES IN A WINDOW

## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

Position	Bit	Bucket
1	1	$[1, T_0]$
2	0	$[1, T_0]$
3	1	$T_1, [1, T_1]$
4	0	$[1, T_0] [1, T_1]$
5	1	$[2, (T_0 - T_1)] [1, T_2]$
6	1	$[2, (T_0 - T_1)] [1, T_2] [1, T_3]$
7	0	
8	0	
9	0	
10	1	$[3, (T_0 - T_2)] [1, T_3] [1, T_4]$
11	0	
12	1	$[4, (T_0 - T_3)] [1, T_4] [1, T_5]$

# COUNTING ONES IN A WINDOW

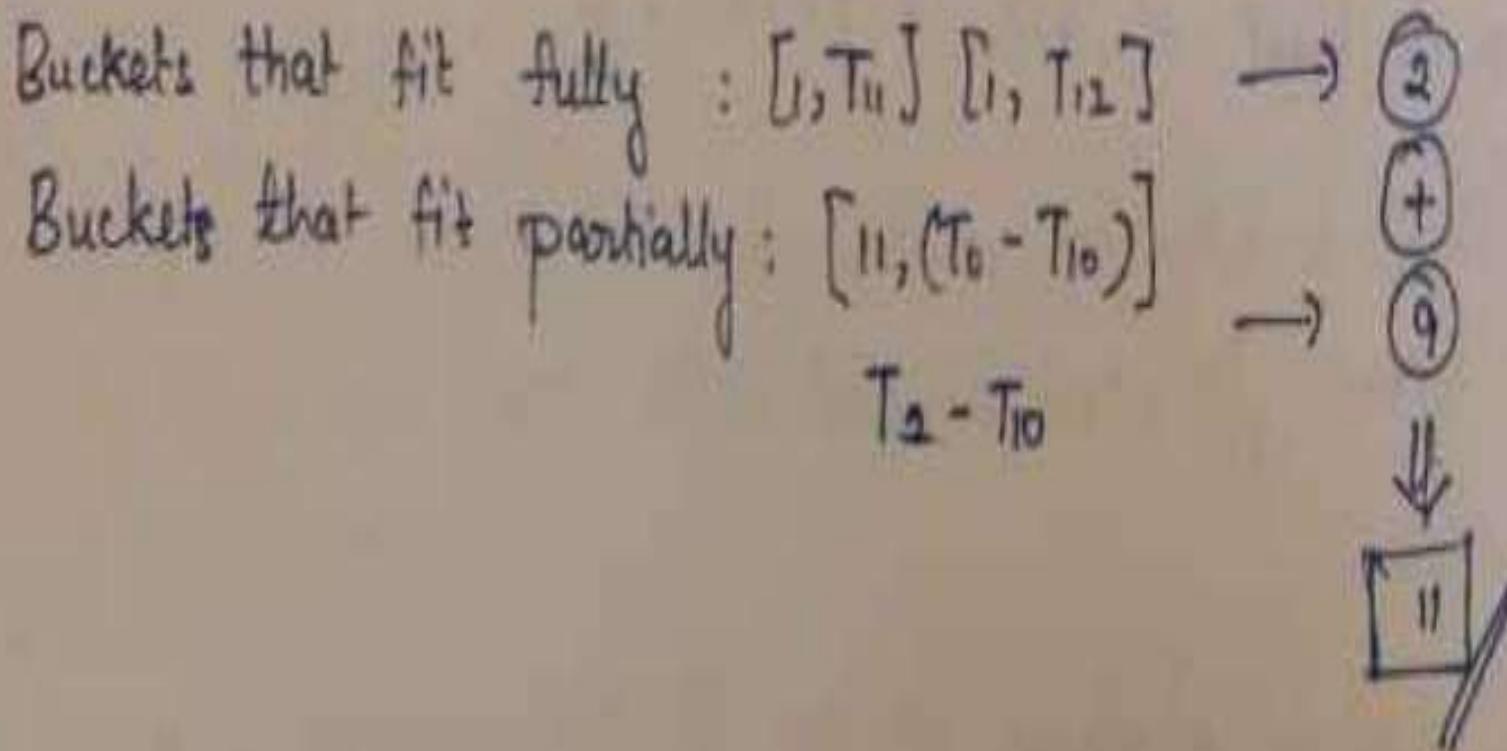
## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

13	1	$[5, (T_0 - T_4)]$	$[1, T_5]$	$[1, T_6]$
14	1	$[6, (T_0 - T_5)]$	$[1, T_6]$	$[1, T_7]$
15	0			
16	1	$[7, (T_0 - T_6)]$	$[1, T_7]$	$[1, T_8]$
17	1	$[8, (T_0 - T_7)]$	$[1, T_8]$	$[1, T_9]$
18	0			
19	0			
20	1	$[9, (T_0 - T_8)]$	$[1, T_9]$	$[1, T_{10}]$
21	0			
22	1	$[10, (T_0 - T_9)]$	$[1, T_{10}]$	$[1, T_{11}]$
23	1	$[11, (T_0 - T_{10})]$	$[1, T_{11}]$	$[1, T_{12}]$
24	0			

# COUNTING ONES IN A WINDOW

## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

Last 20 bits  $\Rightarrow$  Window :  $[T_2, T_{12}]$



# COUNTING ONES IN A WINDOW

## DGIM(Datar – Gionis – Indyk – Motwani Algorithm)

Rules:

1. Every 1, in a stream will be in one bucket
2. Every 1, in a stream will be in one bucket
3. Bucket size is always a power of 2.
4. When you move left, bucket size will increase
5. The rightmost bucket will have a 1.
6. There can be at most 2 buckets of same size

Actual no. of 1's =  $c$

Estimated no. of 1's :  $e$

$$\frac{c}{2} \leq e \leq 2c$$

50% error

Memory :  $O(\log^2 N)$

→ Which movie is popular recently?

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 2

TOPIC:DECAYING WINDOWS

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

## DECAYING WINDOWS

We saw that a sliding window held a certain tail of the stream either the most recent  $n$  elements or all elements that arrived after some time in past.

Sometimes, we want to give more weightage to the recent elements. In this context, we will understand about the exponentially decaying windows.

- (1) The problem of most common elements
- (2) Definition of Decaying window
- (3) Finding the most popular elements

## 1. The problem of most common elements

- You want to track the most popular movies "currently" based on the tickets sold. The idea of "currently" is a bit vague, but the goal is to focus on recent sales and discount older sales.

For example:

- *Movie A* might have been a blockbuster years ago, selling millions of tickets, but its current popularity might be low.
- *Movie B* might be moderately popular but consistent, selling a steady number of tickets each week.
- *Movie C* could have a sudden surge in popularity, selling a lot of tickets recently but not before.

The challenge is to find a way to rank these movies based on **how popular they are *now* rather than over their entire history.**

**The Proposed Solution:** imagine a **bit stream** for each movie:

**Bit Stream for a Movie:** Each movie has a bit stream where the  $i$ th bit is 1 if the  $i$ th ticket was sold for that movie, and 0 otherwise.

For example:

If the stream is 001001100, it means that tickets 3, 6, and 7 were sold for this movie.

**Window Size (N):** This is the number of most recent tickets that will be considered when evaluating a movie's popularity. For example, if  $N = 1000$ , only the last 1000 ticket sales are considered.

## Estimating Popularity

To estimate how many tickets a movie has sold in the last N tickets:

**Windowed Counting:** The solution references Section 4.6, which likely discusses an algorithm (like the DGIM algorithm) for estimating counts over a sliding window.

**Ranking:** Based on these estimates, movies can be ranked by their current popularity.

## Why This Works

This method works for a **moderate** number of movies (thousands) because:

**Feasibility:** Keeping bit streams for each movie is manageable.

**Efficiency:** The DGIM algorithm, for example, allows efficient approximate counting within a sliding window.

## Challenges for Larger Datasets

However, this method becomes problematic for:

**Massive Datasets:** If you need to track the popularity of billions of items (e.g., products on Amazon or tweets by Twitter users), storing bit streams for each item becomes infeasible due to memory and processing constraints.

**Approximation:** The method only gives approximate answers, which might be insufficient for very large datasets where precision is crucial.

### To Summarize:

The key idea is to focus on recent data using a sliding window to evaluate the popularity of movies, which is effective for a moderate number of movies. However, scaling this approach to massive datasets with millions or billions of items would require different techniques due to the limitations of memory and processing power, as well as the inherent approximations of the method.

## (2) Definition of the Decaying Window

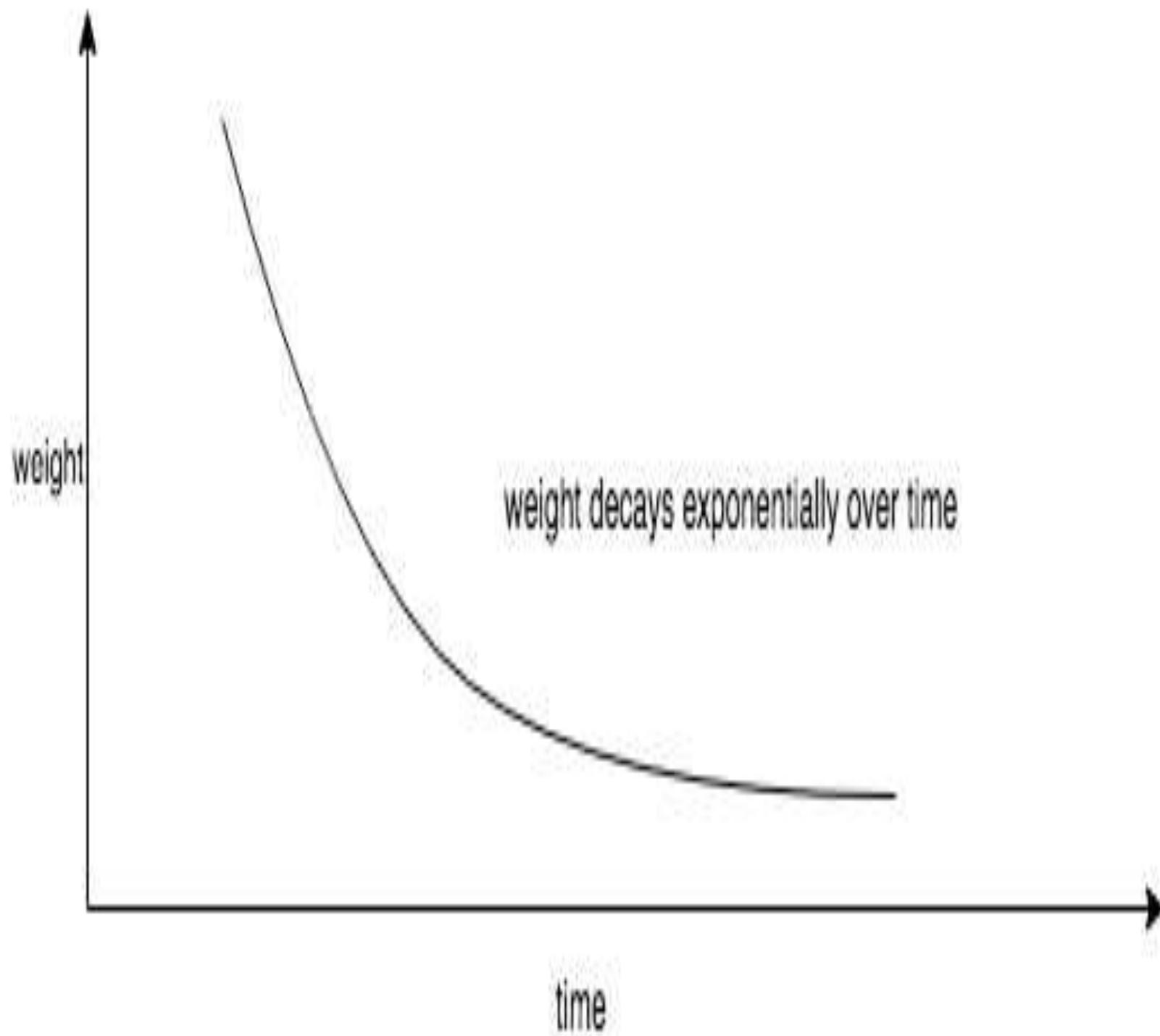
The idea here is to **change the way we measure the popularity of items (like movie tickets) over time, by giving more weight to recent events and less weight to older ones.**

This is done using an **exponentially decaying** weight, which is a method commonly used to smooth out data and focus more on the recent part of the stream.

## **The Problem with the Sliding Window Approach**

The sliding window approach only considers the last N events, discarding all earlier events. This can be limiting, as you might lose important context about the overall trend.

For example, if a movie was very popular two weeks ago but less so now, the sliding window might completely ignore that past popularity if it's outside the window.



## Exponentially Decaying Weights

Instead of a sliding window, you can use an exponentially decaying weight to sum up all the events (in this case, ticket sales) in the stream. Here's how it works:

1. Stream Representation: Suppose the stream currently has elements

$a_1, a_2, \dots, a_t$ , where:

- $a_1$  is the first event (e.g., the first ticket sold),
- $a_t$  is the most recent event.

2. Decay Factor ( $c$ ): Introduce a small constant  $c$  (e.g.,  $10^{-6}$  or  $10^{-9}$ ). This constant controls how quickly the weight of older events decays. The larger the value of  $c$ , the faster the decay.

### 3. Weighted Sum (Exponential Decay):

- For each event in the stream, multiply the event's value by a factor that decreases exponentially based on how far back in time the event occurred.
- The formula for the exponentially decaying sum is:

$$X_t = \sum_{i=0}^{t-1} a_{t-i} \cdot (1 - c)^i$$

Here:

- $a_{t-i}$  is the event that occurred  $i$  steps ago (e.g.,  $a_t$  is the current event,  $a_{t-1}$  is the previous event, and so on).
- $(1 - c)^i$  is the decaying weight, which gets smaller as  $i$  increases (i.e., as the event gets older). 

# How It Works

**Recent Events:** Events that happened more recently (small i) are given a weight close to 1, meaning they contribute almost fully to the sum.

**Older Events:** Events that happened long ago (large i) are given a much smaller weight due to the exponential decay, meaning they contribute less to the sum.

## Example

Let's say you have a stream of ticket sales, and you want to measure the popularity of a movie using this exponentially decaying method.

- Suppose  $c = 0.001$ .
- If a ticket was sold just now ( $i = 0$ ), its weight is  $(1 - 0.001)^0 = 1$ .
- If a ticket was sold 1000 events ago, its weight is  $(1 - 0.001)^{1000} \approx 0.368$ .
- If a ticket was sold 10,000 events ago, its weight is  $(1 - 0.001)^{10000} \approx 0.0045$ .

So, the more recent tickets have a greater influence on the popularity measure, but older tickets still have some impact, though it's reduced.

## Finding the Most Popular Elements

Let us return to the problem of finding the most popular movies in a stream of ticket sales.

We shall use an exponentially decaying window with a constant  $c$ , which you might think of as  $10^{-9}$ . That is, we approximate a sliding window holding the last one billion ticket sales.

This approach is a refined method for tracking the popularity of movies in a stream of ticket sales using an **exponentially decaying window**.

The key idea is to maintain a running score for each movie that reflects its "current" popularity, with older ticket sales having less influence than recent ones.

# The Setup

**Stream of Ticket Sales:** Imagine you're receiving a continuous stream of ticket sales, where each sale corresponds to a specific movie.

**Exponential Decay:** For each movie, you keep a running score that represents its popularity. This score decays over time so that older sales contribute less to the score than more recent ones.

**Threshold:** To manage the large number of possible movies, you only keep track of movies whose scores are above a certain threshold (e.g.,  $1/2$ ).

# How the Method Works

## 1. Decaying the Scores:

- For every movie you are currently tracking (i.e., movies with a score above the threshold), multiply its score by  $(1 - c)$ , where  $c$  is a small constant (e.g.,  $10^{-9}$ ).
- This step simulates the natural decay of the movie's popularity over time, ensuring that older ticket sales have a diminishing impact on the score.

## 2. Updating the Score for the New Ticket:

- When a new ticket sale arrives for a specific movie  $M$ :
  - If  $M$  already has a score: Add 1 to its current score, reflecting the impact of this new ticket sale.
  - If  $M$  does not have a score: Create a new score for  $M$  and set it to 1. This indicates that this is the first time  $M$  is being tracked in this session.

### 3. Pruning Low Scores:

- After updating the scores, check if any movie's score has dropped below the threshold (e.g., 1/2).
- If a movie's score is below this threshold, stop tracking it by dropping its score. This step helps manage the number of movies you're tracking, ensuring that only relatively popular movies are maintained.

## Why It Works

- **Efficient Memory Use:** By dropping scores below the threshold, you avoid wasting memory on tracking unpopular movies.
- **Focus on Recent Popularity:** The use of an exponentially decaying score ensures that the ranking reflects the current popularity rather than cumulative history, making it responsive to recent trends.
- **Threshold Below 1:** The threshold must be less than 1 because you want to ensure that movies that are no longer popular (i.e., those with decayed scores) can be removed from tracking.

## Example

Let's say you're tracking the popularity of movies, and a ticket for *Movie A* arrives:

- **Decaying Step:** If *Movie A* had a score of 0.8 before, after decay (assuming  $c = 10^{-9}$ ), the new score might be slightly less, like 0.79999992.
- **Update Step:** You then add 1 to *Movie A*'s decayed score, making it 1.79999992.
- **Pruning Step:** If another movie *Movie B* had a score that decayed below 1/2, say to 0.4, you would stop tracking *Movie B* by removing its score.

The statement is about understanding why the number of movies whose scores are actively maintained (i.e., those that have a score above the threshold) is limited, and it provides a theoretical upper bound on how many such movies there can be at any given time.

## Key Concepts

1. Exponential Decay & Score Sum: Each movie's score decays over time, as described before, with a decay constant  $c$ . The important point is that the sum of all these scores across all movies in the system remains bounded.
2. Total Score Sum: It is given that the total sum of scores for all movies in the system at any time is approximately  $\frac{1}{c}$ . This is a key constraint that helps us understand the limit on the number of movies being tracked.
3. Threshold (1/2): The system only maintains scores for movies that have a score above a certain threshold, here taken as 1/2.

## The Argument

- **Sum of All Scores:** Suppose you are tracking many movies, and each movie's score is decaying and being updated as new ticket sales come in. Even though individual scores change, the total sum of all these scores remains approximately  $\frac{1}{c}$ .
- **Upper Bound on Active Movies:** Now, consider that you are only interested in tracking movies that have a score of at least  $1/2$ . The argument here is that the total number of such movies is limited by the sum of the scores:
  - If there were more than  $\frac{2}{c}$  movies with a score of at least  $1/2$ , then the sum of these scores alone would exceed  $\frac{1}{c}$ , which is impossible because we know that the total sum of scores cannot exceed  $\frac{1}{c}$ .

Here's why:

- If each of these movies has a score of at least  $1/2$ , then for  $n$  movies, the total score would be at least  $\frac{n}{2}$ .
- To ensure that  $\frac{n}{2}$  does not exceed  $\frac{1}{c}$ , we require:

$$\frac{n}{2} \leq \frac{1}{c}$$

$$n \leq \frac{2}{c}$$

So,  $n \leq \frac{2}{c}$ , meaning the maximum number of movies that can have a score of at least  $1/2$  at any given time is  $\frac{2}{c}$ .

## Practical Implication

- **Concentration on Popular Movies:** While  $\frac{2}{c}$  gives a theoretical upper limit, in practice, the number of movies that are actively popular at any given time is usually much smaller than this limit. Ticket sales typically concentrate on a relatively small number of blockbuster movies, so only a few movies maintain scores above the threshold of 1/2.
- **Memory Efficiency:** This theoretical limit  $\frac{2}{c}$  assures that the system does not need to track an overwhelming number of movies, making the process manageable and memory-efficient, even in a scenario with potentially billions of different movies.

30.08.2024.

## Link Analysis

### PageRank

#### 1 Early Search engines and Term spam

Crawled the web and extracted terms

Stored in Inverted index

User → Search Query (Collection of terms)

Rank the pages according to terms & inverted index.

terms in header → more importance.

e.g.: T-shirt seller adds term "movie" many times (1000)

↳ Search engine considers him as movie page

Ranked 1st

"Term spam" → Fooling the search engine by pretending to be clever, else

Google → Can't be fooled

↳

1. Page Rank

2. Links pointing to the page contain terms.

Pages frequently visited by random surfer → High rank

Links that are pointing to that page should also contain the term

movie →

Creates spam farm → Create millions of links created by spammer pointing to his page

But pages created by spammer will not be considered important acc. to page rank

2. Page Rank

is a function that assigns a real number to every page in the web.

high rank → more important

### Idealized Page rank algorithm

Web → Directed graph

Pages → nodes

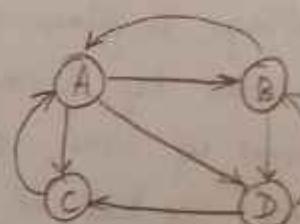
bw pages → edges

Transition matrix M : (n x n)

$$M_{ij} = \frac{1}{k}$$

if there are k out arcs from j to i  
and one of those arcs should go to i

Early search engines were fooled by unethical people trying to pretend as common else



n = no. of nodes

$$M = \begin{matrix} A & B & C & D \\ A & 0 & \frac{1}{2} & 1 & 0 \\ B & \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ C & \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ D & \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{matrix}$$

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$

$$\sum = 1 \Rightarrow \text{"Stochastic matrix"}$$

Principal Eigen vector  
Page Rank vector

Vector (n elements) Iterations → Page Rank Vector

$v_0 = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \vdots \\ \frac{1}{4} \end{bmatrix}$  → Surfer can randomly choose any page  
Probability =  $\frac{1}{4}$

$$v_0 = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix}$$

Initially

Matrix Vector multiplication

$$v_1 = M \times v_0$$

$$v_2 = M \times v_1 = M \times M \times v_0 = M^2 v_0$$

$$v_i = M^i v_0$$

When to stop

When resulting vector stabilizes

No / very little change in  $v_i$

"Markovian process"

When do you reach the limiting distribution

- 1. Graph should be strongly connected (Can reach from one node to any node)
- 2. No dead ends (No outgoing edge from a node)

In general 50 to 75 iterations → can find the page rank of entire web.

e.g.  $v_1 = M v_0$

$$= \begin{bmatrix} 0 & \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix} = \begin{bmatrix} \frac{3}{8} \\ \frac{5}{24} \\ \frac{5}{24} \\ \frac{5}{24} \end{bmatrix} \dots \text{after several iterations, it stabilizes}$$

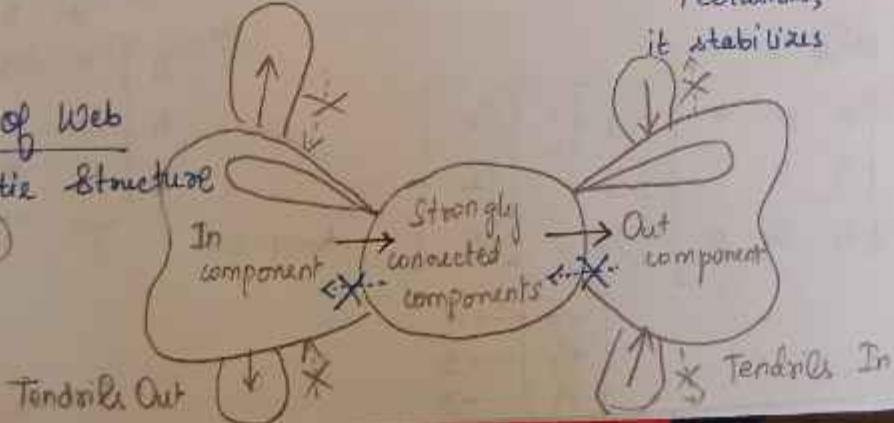
Page Rank

$$\begin{bmatrix} \frac{3}{8} \\ \frac{5}{24} \\ \frac{5}{24} \\ \frac{5}{24} \end{bmatrix} \rightarrow \begin{bmatrix} \frac{3}{8} \\ \frac{2}{9} \\ \frac{2}{9} \\ \frac{2}{9} \end{bmatrix}$$

### 3. Structure of Web

Isolated components

Bowtie Structure



If a random surfer, if it reaches orde component  $\rightarrow$  no further way to move

## PROBLEMS

### ① Dead End

Avoid dead ends to compute page rank

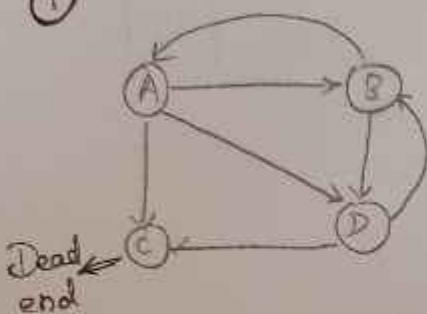
### ② Spider Trap

Set of pages created by spammer

"Trap" You can only loop through those set of pages

#### 4. Avoiding Dead ends

Method  
①



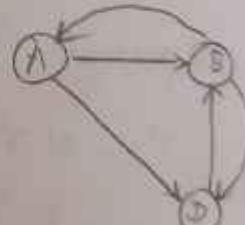
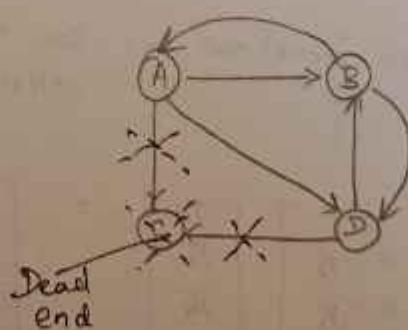
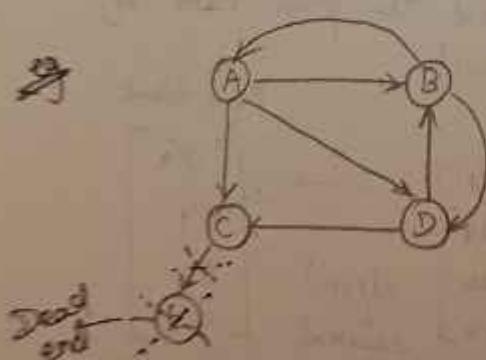
Page with no outgoing links

$$M = \begin{bmatrix} A & B & C & D \\ A & 0 & \frac{1}{2} & 0 & 0 \\ B & \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ C & \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ D & \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix}$$

Sub Stochastic Matrix

Even after doing several iterations, page rank vector will not give any info

- ① Remove the dead end
  - ② Remove all the incoming iterations
- Recursively
- ③ Resulting SCC graph
  - ④ Find the page rank for the nodes of SCC.



Reduced graph  
(SCC)

$$V_1 = M V_0 =$$

$$\begin{bmatrix} 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{2} \\ \frac{1}{3} \end{bmatrix}$$

$$M = \begin{bmatrix} A & B & D \\ A & 0 & \frac{1}{2} & 0 \\ B & \frac{1}{2} & 0 & 1 \\ D & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

$V_0 = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}$

$$\dots \begin{bmatrix} \frac{1}{6} \\ \frac{1}{2} \\ \frac{1}{3} \end{bmatrix} \rightarrow A \\ \frac{1}{6} \rightarrow B \\ \frac{1}{3} \rightarrow D$$

Page Rank of / Probability of reaching C from A

$$PR(C) = \frac{1}{3} PR(A) + \frac{1}{2} PR(D)$$

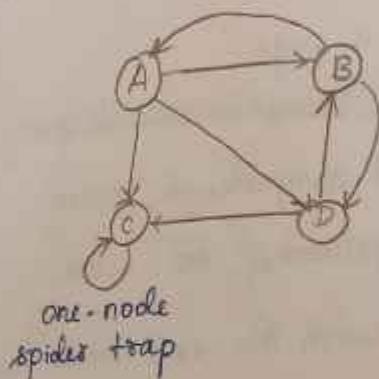
$$= \frac{1}{3} \left(\frac{2}{9}\right) + \frac{1}{2} \left(\frac{3}{9}\right) = \frac{13}{54}$$

$$PR(E) = 1 - PR(c)$$

$$= \frac{13}{54}$$

19.2024  
Spider Traps and Taxation

- ↳ Group of nodes created by the spammer.
- Collection of nodes intentionally created by the spammer to get high page rank for their spam pages.
- Once the surfer gets into the trap, even though he follows outlinks, he can never come out of the trap.



$$M = \begin{bmatrix} A & B & C & D \\ A & 0 & \frac{1}{2} & 0 & 0 \\ B & \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ C & \frac{1}{3} & 0 & 1 & \frac{1}{2} \\ D & \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix} \quad v_0 = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix}$$

$$v_1 = M \times v_0$$

$$v_2 = M \times v_1$$

$$v = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \rightarrow \begin{array}{l} \text{Other pages will get rank 0} \\ \text{Only page C will have high page rank} \end{array}$$

Taxation method

Give a small probability to the surfer to come out of the spider trap.

↳ Modified formula

$\beta$  is a small constant  
e is a vector of ones

n is the no. of nodes in the graph.

$$v_i = \underbrace{\beta M v}_\text{Small constant} + \underbrace{(1-\beta) e/n}_\text{Small prob that the surfer will go to some other node at random}$$

Prob that the surfer will follow the outgoing link of the current node

Prob that the surfer will go to some other node at random

$\Rightarrow$  all  $e_i$  are 1

$$V_i = \beta M V + (1 - \beta) e/n$$

$$\text{Let } \beta = 0.8 = 4/5$$

$$\begin{aligned} \beta M V &= \begin{bmatrix} 0 & \frac{1}{10} & 0 & 0 \\ \frac{4}{15} & 0 & 0 & \frac{4}{10} \\ \frac{4}{15} & 0 & \frac{4}{5} & \frac{4}{10} \\ \frac{4}{15} & \frac{4}{10} & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \\ &+ (1 - \beta) e/n \end{aligned}$$

Route B

$$\begin{bmatrix} 15/148 \\ 19/148 \\ 95/148 \\ 19/148 \end{bmatrix}$$

Still C has higher page rank  
But other pages don't get 0.

### How to use Page Rank

Search Engine  $\rightarrow$  Secret formula

Google

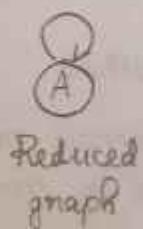
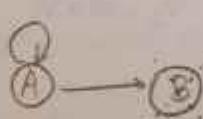
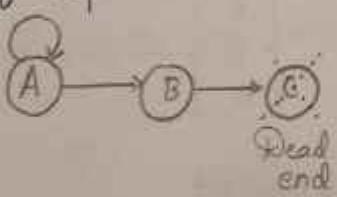
↳ 250 properties to generate score for a page.

Qualified page : Pages that contain the terms in search query

Produces a list of qualified pages and computes a score

Page Rank is just a small component of the score.

~~Ex~~ Find the page rank for the nodes in the graph with the root node having a self loop.



$$M = [1] \quad V_0 = [1]$$

$$V_1 = M V_0 = [1]$$

$$\vdots$$

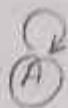
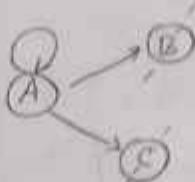
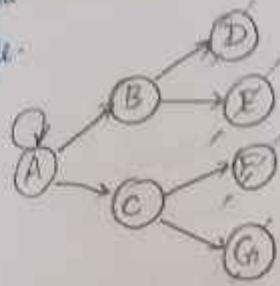
$$PR(A) = 1$$

$$PR(B) = 1 \times PR(A) = 1$$

$$PR(C) = 1 \times PR(B) = 1$$

Continuous linear chain  $\Rightarrow$  all nodes get Page Rank 1 with self loop in root

Find the page rank of the graph with a self loop in the root node.



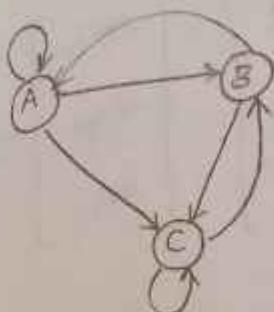
$$PR(A) = 1$$

$$PR(B) = \frac{1}{2} PR(A) = \frac{1}{2}$$

$$PR(C) = \frac{1}{2} PR(A) = \frac{1}{2}$$

$$\begin{aligned} PR(D) &= PR(E) = PR(F) = PR(G) \\ &= \frac{1}{2} PR(B) = \frac{1}{4} \end{aligned}$$

Find page rank of the nodes in the graph with no taxation.



$$M = \begin{bmatrix} 1/3 & 1/2 & 0 \\ 1/3 & 0 & 1/2 \\ 1/3 & 1/2 & 1/2 \end{bmatrix} \quad v_0 = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

Linear Equation Method

$$\begin{aligned} ① \quad R_a &= \frac{1}{3} R_a + \frac{1}{2} R_b + 0 R_c &= \frac{1}{3} R_a + \frac{1}{2} R_b \\ ② \quad R_b &= \frac{1}{3} R_a + 0 R_b + \frac{1}{2} R_c &= \frac{1}{3} R_a + \frac{1}{2} R_c \\ ③ \quad R_c &= \frac{1}{3} R_a + \frac{1}{2} R_b + \frac{1}{2} R_c &= \frac{1}{3} R_a + \frac{1}{2} R_b + \frac{1}{2} R_c \end{aligned}$$

$$\frac{2}{3} R_a = \frac{1}{2} R_b$$

$$\frac{4}{3} R_a = \frac{1}{3} R_a + \frac{1}{2} R_c$$

$$R_b = \frac{4}{3} R_a$$

$$R_a = \frac{1}{2} R_c$$

$$R_c = 2 R_a$$

$$2 R_a = \frac{1}{2} R_a + \frac{1}{2} \left( \frac{4}{3} R_a \right) + \frac{1}{2} (2 R_a)$$

$$R_a + R_b + R_c = 1$$

Sum of all probabilities = 1

$$R_a + \frac{4}{3} R_a + 2 R_a = 1$$

$$\frac{3+4+6}{3} R_a = 1$$

after 4/5 iterations

$$\boxed{R_a = \frac{3}{13} \quad R_b = \frac{4}{13} \quad R_c = \frac{6}{13}}$$

Same graph with taxation:  $\beta = 0.8 = 4/5$

$$M = \begin{bmatrix} 1/3 & 1/2 & 0 \\ 1/3 & 0 & 1/2 \\ 1/3 & 1/2 & 1/2 \end{bmatrix} \quad V_0 = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{aligned} \beta MV + (1-\beta) e/h &= \begin{bmatrix} 4/15 & 4/10 & 0 \\ 4/15 & 0 & 4/10 \\ 4/15 & 4/10 & 4/10 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} + (1 - 4/5) \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} \\ &= \begin{bmatrix} 4/45 + 4/30 \\ 4/45 + 4/30 \\ 4/45 + 8/30 \end{bmatrix} + \begin{bmatrix} 1/15 \\ 1/15 \\ 1/15 \end{bmatrix} = \begin{bmatrix} \frac{8+12+6}{90} \\ \frac{8+12+6}{90} \\ \frac{8+24+6}{90} \end{bmatrix} = \begin{bmatrix} 26/90 \\ 26/90 \\ 38/90 \end{bmatrix} \end{aligned}$$

## Efficient Computation of Page Rank

Suitable for MapReduce  $\leftarrow$  Large scale computation

Two issues:

1. Partial results  $\Rightarrow$  More data to be transmitted

2. Matrix is sparse  $\Rightarrow$  More zeroes

6. Track the location of non zero entries  $\Rightarrow$  Row index / Column index

No need to store zeros

Value  $\Rightarrow$  8 bytes

4 bytes

To track one non zero entry  $\Rightarrow$  16 bytes

4 bytes

Memory space required = Linear.

To store all entries,

Space required = Quadratic

### Representing Transition Matrix

$$M = \begin{bmatrix} 0 & \frac{1}{2} & 1 & 0 \\ V_3 & 0 & 0 & V_2 \\ V_3 & 0 & 0 & V_2 \\ \frac{1}{3} & V_2 & 0 & 0 \end{bmatrix}$$

Store in a table

Src	Out degree	List of successors
A	3	B, C, D
B	2	A, D
C	1	A
D	2	B, C

### Page Rank iteration using MapReduce

n is small  $\rightarrow$  M, V can be stored in Main memory

n  $\rightarrow$  big  $\Rightarrow$  Stripping

Divide the matrix

### Using Combiners

04.09.2024

1. Map Reduce: Use combiners

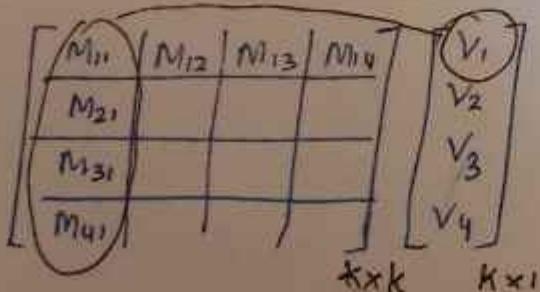
Many partial results  $\rightarrow$  to travel to reducer  
 $\downarrow$

Time consuming

2. No Map Reduce: Single machine

Using many iterations

3.



Use  $\frac{k^2}{k^2}$  map tasks for  $k \times k$  matrix

$M_{11} \times V_1$

$M_{21} \times V_1$

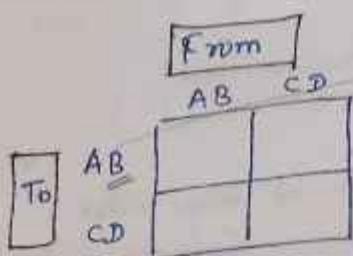
$\dots$

$M_{31} \times V_1$

$M_{41} \times V_1$

Representing blocks of transition matrix

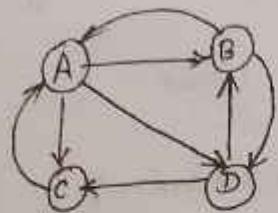
Split the table Quadrant wise



Q1

Src	Outdegree	List of successors
		(A, B, C, D)
A	3	B
B	2	A

Only write  
the successors present  
in To

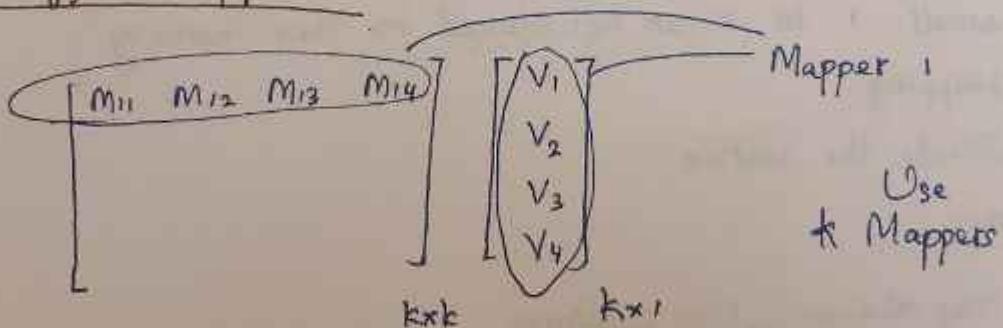


Src	Outdeg	Successors
C	1	A
D	2	B

Src	Outdeg	Successors
A	3	C, D
B	2	D

Src		

Other efficient approaches



Any way, Matrix element just needs to be sent once,  
Vector elements must be sent 4 times.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC:TOPIC- SENSITIVE PAGE RANK

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

## TOPIC – SENSITIVE PAGE RANK

There are several improvements we can make to PageRank. One, to be studied in this section, is that we can weight certain pages more heavily because of their topic.

The mechanism for enforcing this weighting is to alter the way random surfers behave, having them prefer to land on a page that is known to cover the chosen topic.

### Motivation for Topic-Sensitive Page Rank

Different people have different interests, and sometimes distinct interests are expressed using the same term in a query.

The canonical example is the search query *jaguar*, which might refer to the animal, the automobile, a version of the MAC operating system, or even an ancient game console.

If a search engine can deduce that the user is interested in automobiles, for example, then it can do a better job of returning relevant pages to the user.

Ideally, each user would have a private PageRank vector that gives the importance of each page to that user.

It is not feasible to store a vector of length many billions for each of a billion users, so we need to do something simpler.

The topic-sensitive PageRank approach creates one vector for each of some small number of topics, biasing the PageRank to favor pages of that topic.

We then endeavour to classify users according to the degree of their interest in each of the selected topics.

While we surely lose some accuracy, the benefit is that we store only a short vector for each user, rather than an enormous vector for each user.

Example 5.9 : One useful topic set is the 16 top-level categories (sports, medicine, etc.) of the Open Directory (DMOZ).<sup>6</sup> We could create 16 PageRank vectors, one for each topic.

If we could determine that the user is interested in one of these topics, perhaps by the content of the pages they have recently viewed, then we could use the PageRank vector for that topic when deciding on the ranking of pages.

### 5.3.2 Biased Random Walks

Suppose we have identified some pages that represent a topic such as “sports.” To create a topic-sensitive PageRank for sports, we can arrange that the random surfers are introduced only to a random sports page, rather than to a random page of any kind.

The consequence of this choice is that random surfers are likely to be at an identified sports page, or a page reachable along a short path from one of these known sports pages.

Our intuition is that pages linked to by sports pages are themselves likely to be about sports.

The pages they link to are also likely to be about sports, although the probability of being about sports surely decreases as the distance from an identified sports page increases

The mathematical formulation for the iteration that yields topic-sensitive PageRank is similar to the equation we used for general PageRank.

The only difference is how we add the new surfers. Suppose  $S$  is a set of integers consisting of the row/column numbers for the pages we have identified as belonging to a certain topic (called the teleport set).

Let  $\mathbf{e}_S$  be a vector that has 1 in the components in  $S$  and 0 in other components. Then the topic-sensitive PageRank for  $S$  is the limit of the iteration

$$\mathbf{v}' = \beta M \mathbf{v} + (1 - \beta) \mathbf{e}_S / |S|$$

Here, as usual,  $M$  is the transition matrix of the Web, and  $|S|$  is the size of set  $S$ .

**Example 5.10:** Let us reconsider the original Web graph we used in Fig. 5.1, which we reproduce as Fig. 5.15. Suppose we use  $\beta = 0.8$ . Then the transition matrix for this graph, multiplied by  $\beta$ , is

$$\beta M = \begin{bmatrix} 0 & 2/5 & 4/5 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix}$$

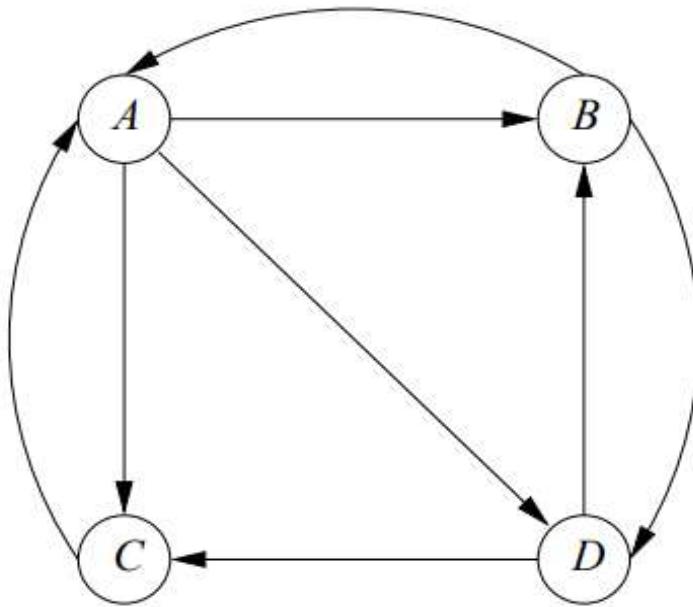


Figure 5.15: Repeat of example Web graph

Suppose that our topic is represented by the teleport set  $S = \{B, D\}$ . Then the vector  $(1 - \beta)\mathbf{e}_S/|S|$  has  $1/10$  for its second and fourth components and 0 for the other two components. The reason is that  $1 - \beta = 1/5$ , the size of  $S$  is 2, and  $\mathbf{e}_S$  has 1 in the components for  $B$  and  $D$  and 0 in the components for  $A$  and  $C$ . Thus, the equation that must be iterated is

$$\mathbf{v}' = \begin{bmatrix} 0 & 2/5 & 4/5 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix} \mathbf{v} + \begin{bmatrix} 0 \\ 1/10 \\ 0 \\ 1/10 \end{bmatrix}$$

$$\begin{bmatrix} 0/2 \\ 1/2 \\ 0/2 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 2/10 \\ 3/10 \\ 2/10 \\ 3/10 \end{bmatrix}, \begin{bmatrix} 42/150 \\ 41/150 \\ 26/150 \\ 41/150 \end{bmatrix}, \begin{bmatrix} 62/250 \\ 71/250 \\ 46/250 \\ 71/250 \end{bmatrix}, \dots, \begin{bmatrix} 54/210 \\ 59/210 \\ 38/210 \\ 59/210 \end{bmatrix}$$

Notice that because of the concentration of surfers at  $B$  and  $D$ , these nodes get a higher PageRank than they did in Example 5.2. In that example,  $A$  was the node of highest PageRank.  $\square$

### 5.3.3 Using Topic-Sensitive PageRank

In order to integrate topic-sensitive PageRank into a search engine, we must:

1. Decide on the topics for which we shall create specialized PageRank vectors.
2. Pick a teleport set for each of these topics, and use that set to compute the topic-sensitive PageRank vector for that topic.

3. Find a way of determining the topic or set of topics that are most relevant for a particular search query.
4. Use the PageRank vectors for that topic or topics in the ordering of the responses to the search query.

We have mentioned one way of selecting the topic set: use the top-level topics of the Open Directory. Other approaches are possible, but there is probably a need for human classification of at least some pages.

The third step is probably the trickiest, and several methods have been proposed. Some possibilities:

- (a) Allow the user to select a topic from a menu.
- (b) Infer the topic(s) by the words that appear in the Web pages recently searched by the user, or recent queries issued by the user. We need to discuss how one goes from a collection of words to a topic, and we shall do so in Section 5.3.4
- (c) Infer the topic(s) by information about the user, e.g., their bookmarks or their stated interests on Facebook.

### 5.3.4 Inferring Topics from Words

The question of classifying documents by topic is a subject that has been studied for decades, and we shall not go into great detail here.

Suffice it to say that topics are characterized by words that appear surprisingly often in documents on that topic.

For example, neither fullback nor measles appear very often in documents on the Web. But fullback will appear far more often than average in pages about sports, and measles will appear far more often than average in pages about medicine.

If we examine the entire Web, or a large, random sample of the Web, we can get the background frequency of each word.

Suppose we then go to a large sample of pages known to be about a certain topic, say the pages classified under sports by the Open Directory.

Examine the frequencies of words in the sports sample, and identify the words that appear significantly more frequently in the sports sample than in the background.

In making this judgment, we must be careful to avoid some extremely rare word that appears in the sports sample with relatively higher frequency.

This word is probably a misspelling that happened to appear only in one or a few of the sports pages.

Thus, we probably want to put a floor on the number of times a word appears, before it can be considered characteristic of a topic.

Once we have identified a large collection of words that appear much more frequently in the sports sample than in the background, and we do the same for all the topics on our list, we can examine other pages and classify them by topic.

Here is a simple approach. Suppose that  $S_1, S_2, \dots, S_k$  are the sets of words that have been determined to be characteristic of each of the topics on our list.

Let  $P$  be the set of words that appear in a given page  $P$ . Compute the Jaccard similarity between  $P$  and each of the  $S_i$ 's.

Classify the page as that topic with the highest Jaccard similarity. Note that all Jaccard similarities may be very low, especially if the sizes of the sets  $S_i$  are small.

Thus, it is important to pick reasonably large sets  $S_i$  to make sure that we cover all aspects of the topic represented by the set.

We can use this method, or a number of variants, to classify the pages the user has most recently retrieved.

We could say the user is interested in the topic into which the largest number of these pages fall.

Or we could blend the topicsensitive PageRank vectors in proportion to the fraction of these pages that fall into each topic, thus constructing a single PageRank vector that reflects the user's current blend of interests.

We could also use the same procedure on the pages that the user currently has bookmarked, or combine the bookmarked pages with the recently viewed pages

## Link Spam

- Page rank technique used by Google made Term Spam attack ineffective.
- So, spammers started to fool pageRank algorithms by creating techniques to artificially increase the page rank of their spam pages. These techniques are collectively called as Link Spam.

We will see how spammers create Link spam & then we'll discuss methods to overcome spamming such as :  
1) Trust Rank  
2) Measurement of spam math.

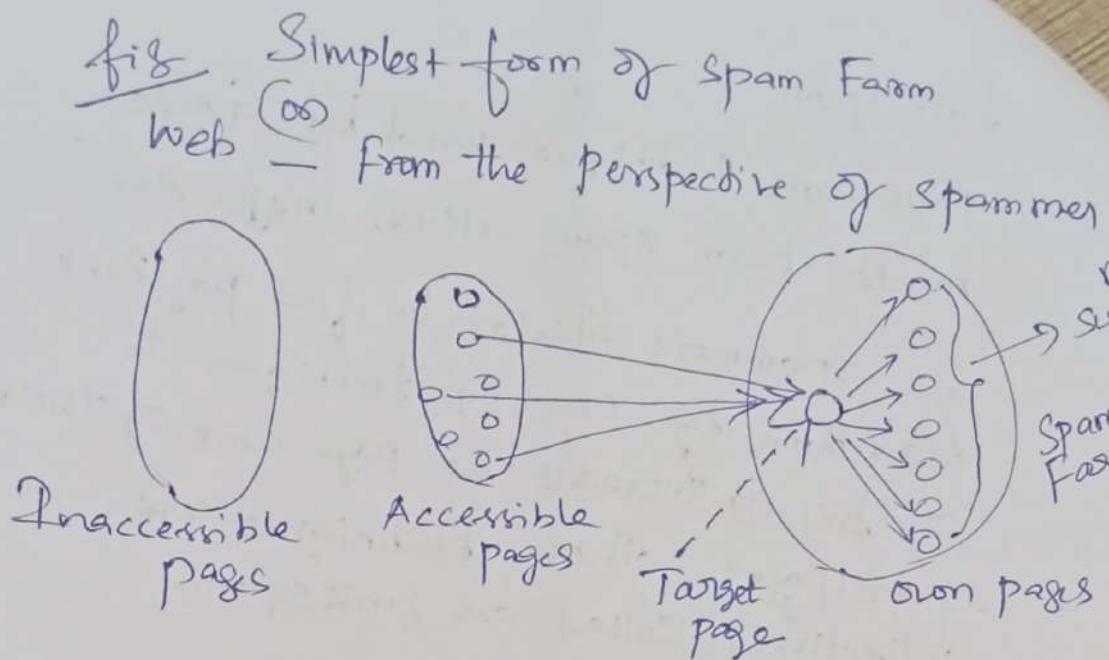
- (1) Architecture of spam farm
- (2) Analysis of spam farm
- (3) Combating Link spam
- (4) TrustRank
- (5) Spam math

1

### Architecture of Spam Farm

Defn : A collection of pages whose purpose is to increase the page rank of a certain page (or) pages is called spam farm.

fig Simplest form of Spam Farm

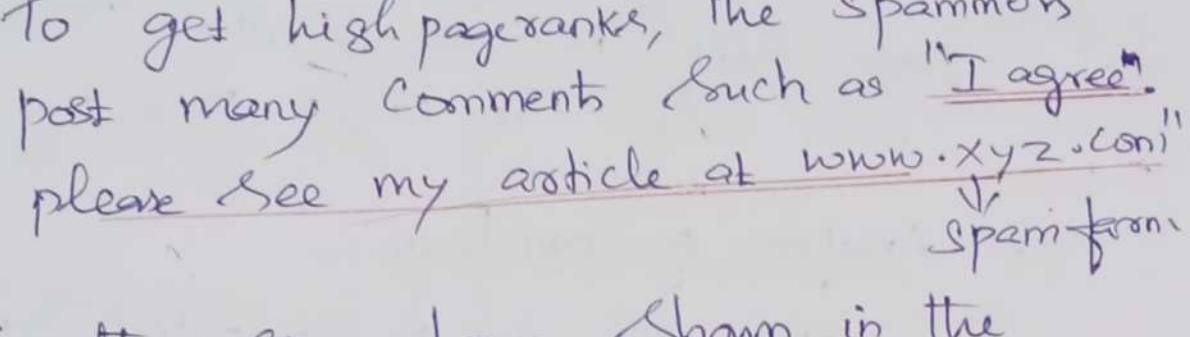


from the point of view of a spammer,

web is divided into 3 parts:

1. Inaccessible pages — pages that the Spammer cannot affect. Most of the web is in this part only.
2. Accessible pages — those pages that can be affected by the spammer. (even though they are not controlled by spammer)
3. own pages : the pages that are owned & controlled by the spammer.

\* Spam farm consists of Spammer's own pages, organized in a special way. (note in Rts of figure)  
with some links from the accessible pages to the Spammer's pages. without such links, the spam farm will be useless bcoz it will not be crawled by search engine.

- Eg: Many blogs (or) newspapers in web invite others to post their comments on the site.  
To get high pageranks, the spammers post many comments such as "I agree".  
please see my article at www.xyz.com.  


↓  
Spam farm
- For the spam farm shown in the figure, there is one page  $t$  (target page) at which the spammer tries to place the page rank.
- Link structure -
  - \* from  $t$  to supporting pages:
    - The target page  $t$  has links pointing to each of the supporting pages.
    - This distributes some of  $t$ 's page rank to the supporting pages.
  - \* From supporting pages to  $t$ :
    - Each supporting page links back only to the target page  $t$ .
    - This ensures that any page rank accumulated by the supporting pages is funneled back to  $t$ .

By linking only within this ~~closed~~  
structure of pages Spam farm

It ensures that almost all the  
Page rank is eventually concentrated  
back to the target page  $t$ .

② Analysis of Spam Farm.

- Suppose that page rank is computed using a damping parameter,  $\beta$  (around 0.85).
- $\beta$  is the fraction of a page's pagerank that gets distributed to its successors at the next round.
- Let  $n$  be the total pages in web some of which is a spam farm with target page  $t$  &  $m$  supporting pages.
  - (A) Let  $x$  be the amount of pagerank contributed by the accessible pages.  
 $x$  is the sum of the page rank contributions from these accessible pages.

- (B) Contribution from supporting pages  
Each of the  $m$  supporting pages links only to  $t$ .  
Let  $y$  denote page rank of  $t$ .

The page rank of each supporting page =  $\beta \frac{y}{m} + \frac{1-\beta}{n}$

- ⇒ First term,  $\frac{\beta y}{m}$  is the contribution from the target page t.  $\beta y$  is the portion of t's page rank distributed to m supporting pages.
- ⇒ Second term,  $\frac{1-\beta}{n}$  is the share of the supporting page from the fraction  $1-\beta$  of the page rank that is divided equally among all pages on web.

Page rank of target Page(t)

Page rank(y) of target Page(t) is derived from:

1. Contribution x from outside pages that link to t.
2. Contribution from supporting pages =

$$\beta \times m \times \left( \beta \frac{y}{m} + \frac{1-\beta}{n} \right)$$

(Since.  
Each supporting page contributes  $\beta$  times its PageRank to t).

$$y = \textcircled{1} + \textcircled{2}$$

$$= x + \beta m \left( \frac{\beta y}{m} + \frac{1-\beta}{n} \right)$$

$$y = x + \beta^2 y + \beta(1-\beta) \left( \frac{m}{n} \right)$$

$$y - \beta^2 y = x + \beta(1-\beta) \left( \frac{m}{n} \right)$$

$$y(1-\beta^2) = x + \beta(1-\beta) \left( \frac{m}{n} \right)$$

$$y = \frac{x}{1-\beta^2} + \frac{\beta(1-\beta)}{(1+\beta)(1-\beta)} \left( \frac{m}{n} \right)$$

$$= \frac{x}{1-\beta^2} + \frac{\beta}{1+\beta} \left( \frac{m}{n} \right)$$

$$\boxed{y = \frac{x}{1-\beta^2} + c \left( \frac{m}{n} \right)}$$
 where  $c = \frac{\beta}{1+\beta}$

$$\text{for } \beta = 0.85$$

$$\Rightarrow \frac{1}{1-\beta^2} = \frac{1}{1-(0.85)^2} = 3.6 \quad [3.6 \times 100 = 360] \quad \begin{matrix} \hookrightarrow \text{external} \\ \text{contribution} \\ \hookrightarrow y \end{matrix}$$

$$\Rightarrow c = \frac{\beta}{1+\beta} = \frac{0.85}{1.85} = 0.46 \quad [0.46 \times 100 = 46\%] \quad \begin{matrix} \hookrightarrow \text{spam} \\ \text{from} \\ \text{contribution} \\ \hookrightarrow y \end{matrix}$$

Where  $y$  is the page rank of  $A$

## combating Link spam

- It is essential for search engines to detect & eliminate link spam.
- 2 approaches to Link spam elimination
  - (1) Look for structures such as spam farm where one page links to a very large no. of pages, each of which links back to it.
    - Search engines will search for such Spam farm structures and eliminate them from their index.
    - In turn, spammers will try different structures of spam farms.
    - No end of war between spammers and the search engines.
  - (2) Another approach is to eliminate link spam
    - this approach doesn't locate the spamfarm.
    - A search engine can modify its definition of page rank to lower the rank of link-spam pages automatically.
    - we shall consider 2 formulas:
      - (2a) Trust rank — a variation of topic-sensitive page rank designed to lower the score of spam pages.

(2b) Spam mass - a calculation that identifies the pages that are likely to be spam. It allows the search engines to eliminate those pages (or) to lower their page rank strongly.

#### (4) Trust Rank

- It is a topic sensitive page rank, where the topic is a set of pages believed to be trustworthy (not spam).
- The concept is that while a spam page can be easily linked with a trustworthy page, but it is unlikely that a trustworthy page would link to a spam page.
- The borderline area is a site with blogs (or) other opportunities that allows users to post their comment which enables the spammers to establish a link to their spam pages.
- To implement Trust Rank, we have to develop a suitable teleport set of trustworthy pages.

2 approaches:

4a) Humans examine a set of pages & decide which of them are trustworthy.

e.g.) we can pick pages of highest pagerank to check whether they are trustworthy.

4b) pick a domain whose membership is controlled. It is difficult for the spammers to bring their spam pages

link into these domains.

e.g.) we pick .edu domain as university pages are unlikely to be spam farms.

Likewise, we can pick .mil, .gov domains.

To get a good distribution of trustworthy web pages, we should include such domains (.edu, .gov etc) from all countries.

We can say that search engines are following the second approach, so page rank really is a form of Trustrank.

(5)

## Spam Mass

- For each page, we measure the fraction of its Page Rank that comes from spam.
- We will compute both :
  - a) Ordinary pagerank &
  - b) TrustRank based on some teleport set of trustworthy pages.
- Suppose page P has :

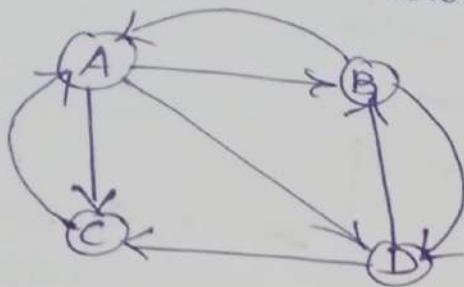
$$\begin{aligned} \text{PageRank} &\rightarrow \gamma \\ \text{TrustRank} &\rightarrow t. \end{aligned}$$

Then, Spam mass of page P =  $\frac{(\gamma - t)}{\gamma}$ .

- \* A negative value (or ~~small~~<sup>positive</sup> small value) for spam mass means that page P is probably not a spam page.
- If spam mass value is close to 1, then the page P is probably a spam page.
- Search engines will eliminate pages with high spam mass from their index. This way, they deal with link spam without having to identify the structures used by spammers.

Eg

Let us consider the web graph below.



Page rank computed for the above graph is

$$\begin{bmatrix} \frac{3}{9} \\ \frac{2}{9} \\ \frac{2}{9} \\ \frac{2}{9} \end{bmatrix}$$

Topic-sensitive page rank for the above graph is

$$\begin{bmatrix} \frac{54}{210} \\ \frac{59}{210} \\ \frac{38}{210} \\ \frac{59}{210} \end{bmatrix}$$

With the teleport set  $S = \{B, D\}$

B & D are the trusted pages.

The following table

shows the pagerank, Trustrank & spam mass for the 4 nodes.

Node	Pagerank $(\alpha)$	Trustrank $(\gamma)$	Spammass $(\alpha - \gamma)/\gamma$
A	$\frac{3}{9}$	$\frac{54}{210}$	0.229
B	$\frac{2}{9}$	$\frac{59}{210}$	-0.264
C	$\frac{2}{9}$	$\frac{38}{210}$	0.186
D	$\frac{2}{9}$	$\frac{59}{210}$	-0.264

Table: Calculation of spam mass

Conclusion from the above table:

- \* Spam mass of pages B & D are negative, so they are not spam.
- \* Spam mass of pages A and C are small positive values that are closer to 0 & not to 1. So, A & C are also not spam.

## Inferring words from pages

fullback → sports

measles → medicines

① Findout the background frequency of words

↓  
freq. in entire web

(or) representative sample

② Find the word frequency in topic sensitive pages

③ Compare

Indicator → Set of topic sensitive words for each topic.

$$\frac{|P \cap S_i|}{|P \cup S_i|} = \frac{\text{Size of intersection of sets}}{\text{Size of union of sets}}$$

Find Jaccard Similarity  
Highest Jaccard similarity  
↓  
Page belongs to  
that topic

"Jaccard Similarity"       $P = \text{Page} \Rightarrow \{w_1, w_2, \dots\}$

$S_1, S_2, S_3, \dots, S_i$

## Link Spam

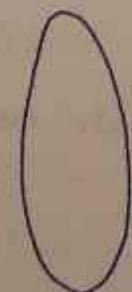
PageRank to combat TermSpam

Spam farm to fool PageRank

Techniques to fool SpamFarm → Link Spam

Collection of pages created by spammers to artificially increase the page rank of a target page(s).

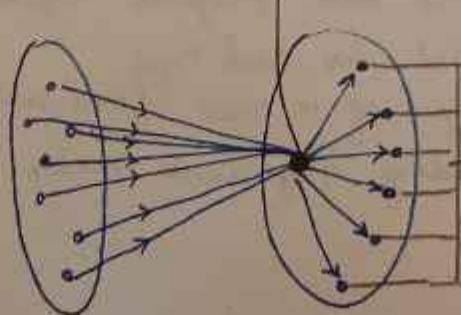
Architecture  
of spamfarm



Inaccessible  
pages

Set of pages that can't  
be attacked by spammers

Target Page ( $t$ )



Accessible  
pages

Can be attacked  
by spammers

Own pages

(Spam farm)

$m$  Supporting pages

Accessible pages  $\rightarrow$  Official website

Spammer shares their comments with links  
to their page  $\Rightarrow$  Link to target page

Link structure  
from the POV of spammer

- ①  $t$  to  $m$  pages
- ②  $m$  pages to  $t$



06-09-2024

### Link Spam - Architecture of Spam Farm

Analysis of spam farm

Combating link spam

Trust rank

Spam mass

### Analysis of Spam Farm

Target  $\rightarrow$  Boosts up

So you want to find out the page rank.

PageRank ( $y$ ) of target page ' $t$ '

$y \Rightarrow$  Contribution of PR from accessible pages

+  
Contribution of PR from supporting pages

Let ' $t$ ' be the target page for which the spammer wishes to increase PageRank.

Let ' $y$ ' be the PageRank of ' $t$ '.

Let  $m$  denote no. of support pages

Let  $B$  denote the fraction of PageRank that will be passed from a page to its successors.

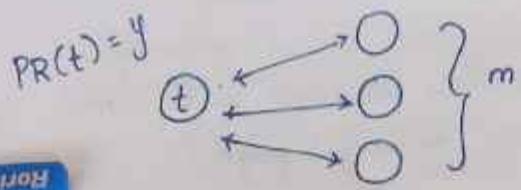
$y = x + \text{Contribution of PR from } m \text{ supporting pages}$



PR of single supporting page =  $\frac{By}{m} + \underbrace{\frac{1-B}{n}}$

$n \rightarrow$  No. of web pages

Contribution of the supporting pages  
get from entire web.



$$\text{PR of } m \text{ supporting pages} = \beta^m \left( \frac{\beta y}{m} + \frac{1-\beta}{n} \right)$$

$$\text{PR of } m \text{ supporting pages} = m \left( \frac{\beta y}{m} + \frac{1-\beta}{n} \right)$$

$$\text{Contribution of PR of } m \text{ supporting pages} = \beta^m \left( \frac{\beta y}{m} + \frac{1-\beta}{n} \right)$$

$$\text{In } ① \quad y = x + \beta^m \left( \frac{\beta y}{m} + \frac{1-\beta}{n} \right)$$

$$= x + \frac{\beta^2 m y}{m} + \beta(1-\beta) \left( \frac{m}{n} \right)$$

$$y = x + \beta^2 y + \beta(1-\beta) \left( \frac{m}{n} \right)$$

$$y - \beta^2 y = x + \beta(1-\beta) \left( \frac{m}{n} \right)$$

$$y(1-\beta^2) = x + \beta(1-\beta) \left( \frac{m}{n} \right)$$

$$y = \frac{x}{1-\beta^2} + \frac{\beta(1-\beta) \left( \frac{m}{n} \right)}{1-\beta^2} = \frac{x}{1-\beta^2} + \frac{\beta(1-\beta)}{(1+\beta)(1-\beta)} \left( \frac{m}{n} \right)$$

$$y = \frac{x}{1-\beta^2} + \frac{\beta}{1+\beta} \left( \frac{m}{n} \right)$$

$$y = \frac{x}{1-\beta^2} + c \left( \frac{m}{n} \right)$$

where  $c = \frac{\beta}{1+\beta}$

Let's take  $\beta = 0.85$

$$\frac{\beta}{1+\beta} = \frac{0.85}{1+0.85} = 0.46 = 46\%$$

$$\frac{1}{1-\beta^2} = \frac{1}{1-(0.85)^2} = 3.6 = 360\%$$

These two values:  
To know how much  
the spammer made the  
PR to increase

Beneficiary: Spammer  $\rightarrow$  Insight

He's getting PR of  $y$  which is getting amplified by the available  
PR (360 & 46)

## Combating Link Spam

- Find and eliminate spam farm
- (i) Find the structure of spam farm
- (ii) Modify PR: lower its score of spam page

- a) Trust Rank Techniques
- b) Spam mass

### Trust Rank

Topic sensitive PR

To lower PR of spam pages

Borderline area:

Loophole for spammers

From a teleport set of trustworthy pages.

a) Team  $\Rightarrow$  Manual Inspection

b) Choose domains: ( $\cdot\text{edu}, \cdot\text{gov}, \cdot\text{mil}$   
followed by search engines)

### Spam mass

Score calculated for a page 'p'

Based on the score, you can classify whether the page is spam page / non-spam page.

Page

Spam

Non-spam

Search engine eliminates

for every page 'p', you need to find

↳ Original rank 'r'

↳ Trust rank 't'

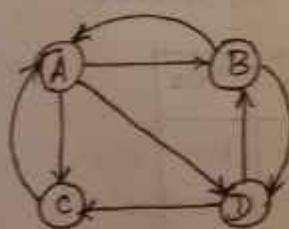
$$\boxed{\text{Spam mass} = \frac{r-t}{r}}$$

If this score is neg (or) small positive number, then spam mass is very less. Hence it will be an original page.

$\therefore$  Page is non-spam

For a positive value nearer to 1  $\rightarrow$  probably a spam page

eg



Original PR

$$\begin{bmatrix} 3/9 \\ 2/9 \\ 2/9 \\ 2/9 \end{bmatrix}$$

Trust rank t

$$\begin{bmatrix} 54/210 \\ 59/210 \\ 38/210 \\ 59/210 \end{bmatrix}$$

Nodes	Og PR y	Trust Rank t	Spam mass
A	3/9	54/210	$\frac{9/9 - 54/210}{3/9} = 0.24$ small +ve
B	3/9	59/210	= -0.05 -ve
C	2/9	38/210	= 0.18 small +ve
D	2/9	59/210	= -ve

Small +ve no. / Negative no.  $\rightarrow$  Non-Spam page

Homeo BK

### Unit - II

#### Recommendation Systems

↳ A model for Recommendation Systems  
eg: Netflix

- ↳ Content based Systems (Properties Based)
- ↳ Collaborative filtering Systems (Based on similarities b/w users and items)

- Utility matrix
- Long tail phenomenon
- Applications
- Populating utility matrix

#### ① Utility matrix

↳ Users & Items  
(U, I) pair  
will have a value

	I <sub>1</sub>	I <sub>2</sub>
U <sub>1</sub>	(high)	(low)
U <sub>2</sub>		

Matrix will be sparse

Entries of most values in matrix will be unknown.

e.g. Movie Rating

Utility matrix

5  $\rightarrow$  High rating  
user likes  
the movie

User Movie	HP <sub>1</sub>	HP <sub>2</sub>	HP <sub>3</sub>	SW <sub>1</sub>	SW <sub>2</sub>
A	5		4		1
B	4		3		
C				4	5
D	2		1		

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC:RECOMMENDATION SYSTEMS

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

# RECOMMENDATION SYSTEMS

There is an extensive class of Web applications that involve predicting user responses to options. Such a facility is called a recommendation system.

Two good examples of recommendation systems are:

1. Offering news articles to on-line newspaper readers, based on a prediction of reader interests.
2. Offering customers of an on-line retailer suggestions about what they might like to buy, based on their past history of purchases and/or product searches.

Recommendation systems use a number of different technologies. We can classify these systems into two broad groups.

# RECOMMENDATION SYSTEMS

- **Content-based systems** examine properties of the items recommended. For instance, if a Netflix user has watched many cowboy movies, then recommend a movie classified in the database as having the “cowboy” genre.
- **Collaborative filtering systems** recommend items based on similarity measures between users and/or items. The items recommended to a user are those preferred by similar users.

# A Model for Recommendation Systems

In this section we introduce a model for recommendation systems, based on a utility matrix of preferences. We introduce the concept of a “long-tail phenomenon”.

1. Utility Matrix
2. Long tail phenomenon
3. Applications of Recommendation systems
4. Populating the utility matrix

## Utility Matrix

In a recommendation-system application there are two classes of entities, which we shall refer to as users and items.

Users have preferences for certain items, and these preferences must be teased out of the data.

The data itself is represented as a utility matrix, giving for each user-item pair, a value that represents what is known about the degree of preference of that user for that item.

Values come from an ordered set, e.g., integers 1–5 representing the number of stars that the user gave as a rating for that item.

We assume that the matrix is sparse, meaning that most entries are “unknown.” An unknown rating implies that we have no explicit information about the user’s preference for the item.

**Example 9.1:** In Fig. 9.1 we see an example utility matrix, representing users' ratings of movies on a 1–5 scale, with 5 the highest rating. Blanks represent the situation where the user has not rated the movie. The movie names are HP1, HP2, and HP3 for *Harry Potter* I, II, and III, TW for *Twilight*, and SW1, SW2, and SW3 for *Star Wars* episodes 1, 2, and 3. The users are represented by capital letters *A* through *D*.

	HP1	HP2	HP3	TW	SW1	SW2	SW3
<i>A</i>	4			5	1		
<i>B</i>	5	5	4				
<i>C</i>				2	4	5	
<i>D</i>		3					3

Figure 9.1: A utility matrix representing ratings of movies on a 1–5 scale

Notice that most user-movie pairs have blanks, meaning the user has not rated the movie. In practice, the matrix would be even sparser, with the typical user rating only a tiny fraction of all available movies

The goal of a recommendation system is to predict the blanks in the utility matrix. For example, would user A like SW2?

There is little evidence from the tiny matrix in Fig. 9.1. We might design our recommendation system to take into account properties of movies, such as their producer, director, stars, or even the similarity of their names.

If so, we might then note the similarity between SW1 and SW2, and then conclude that since A did not like SW1, they were unlikely to enjoy SW2 either.

Alternatively, with much more data, we might observe that the people who rated both SW1 and SW2 tended to give them similar ratings. Thus, we could conclude that A would also give SW2 a low rating, similar to A's rating of SW1.

It is not necessary to predict every blank entry in a utility matrix. Rather, it is only necessary to discover some entries in each row that are likely to be high.

In most applications, the recommendation system does not offer users a ranking of all items, but rather suggests a few that the user should value highly.

It may not even be necessary to find all items with the highest expected ratings, but only to find a large subset of those with the highest ratings.

## The Long Tail

Physical delivery systems are characterized by a scarcity of resources. Brick-and-mortar stores have limited shelf space, and can show the customer only a small fraction of all the choices that exist.

On the other hand, on-line stores can make anything that exists available to the customer.

Thus, a physical bookstore may have several thousand books on its shelves, but Amazon offers millions of books.

A physical newspaper can print several dozen articles per day, while on-line news services offer thousands per day.

Recommendation in the physical world is fairly simple. First, it is not possible to tailor the store to each individual customer.

Thus, the choice of what is made available is governed only by the aggregate numbers.

Typically, a bookstore will display only the books that are most popular, and a newspaper will print only the articles it believes the most people will be interested in.

In the first case, sales figures govern the choices, in the second case, editorial judgement serves.

The distinction between the physical and on-line worlds has been called the long tail phenomenon, and it is suggested in Fig. 9.2.

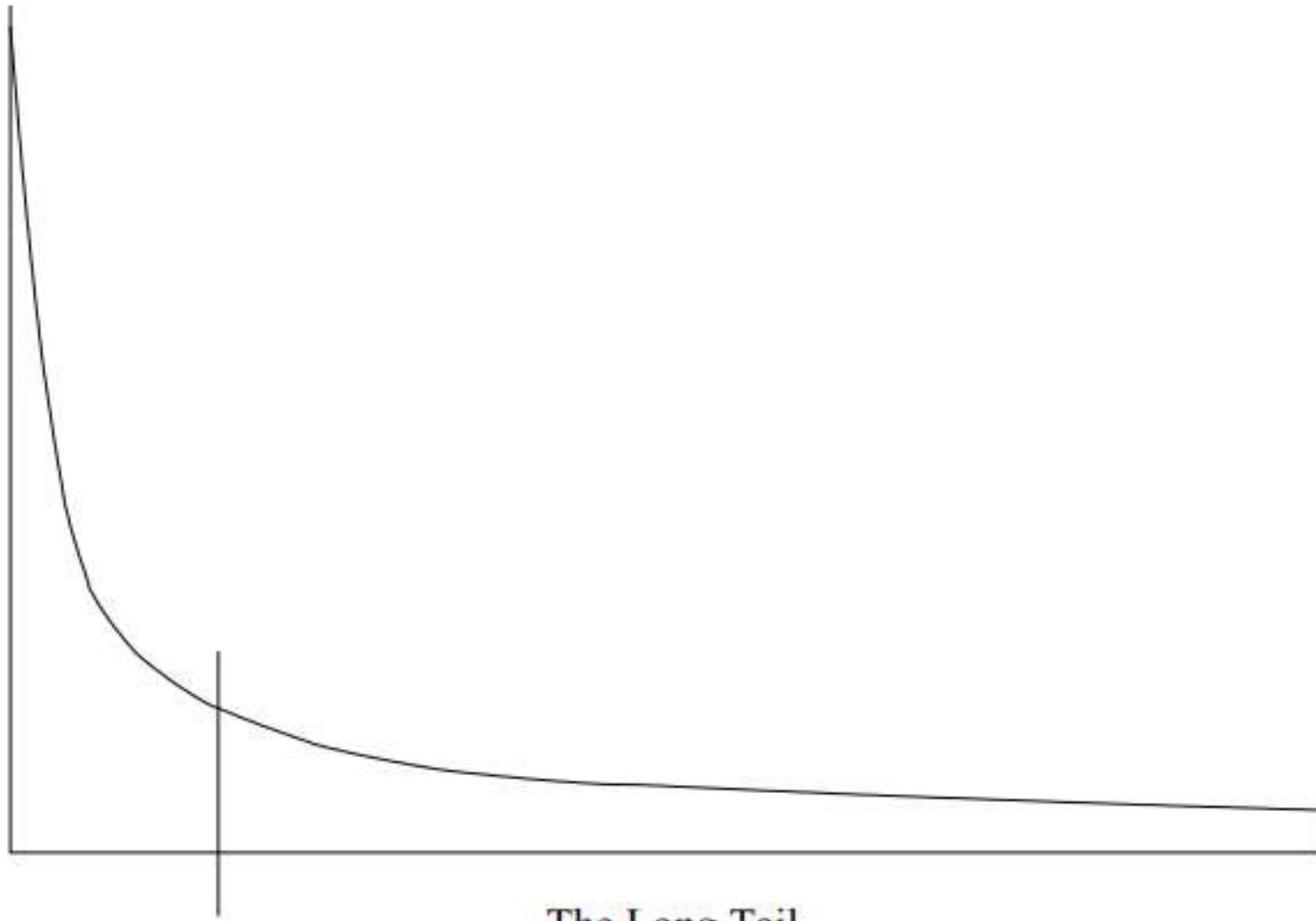


Figure 9.2: The long tail: physical institutions can only provide what is popular, while on-line institutions can make everything available

The vertical axis represents popularity (the number of times an item is chosen). The items are ordered on the horizontal axis according to their popularity.

Physical institutions provide only the most popular items to the left of the vertical line, while the corresponding on-line institutions provide the entire range of items: the tail as well as the popular items.

The long-tail phenomenon forces on-line institutions to recommend items to individual users.

It is not possible to present all available items to the user, the way physical institutions can. Neither can we expect users to have heard of each of the items they might like

# **Applications of Recommendation Systems**

We have mentioned several important applications of recommendation systems, but here we shall consolidate the list in a single place.

1. Product Recommendations: Perhaps the most important use of recommendation systems is at on-line retailers.

We have noted how Amazon or similar on-line vendors strive to present each returning user with some suggestions of products that they might like to buy.

These suggestions are not random, but are based on the purchasing decisions made by similar customers or on other techniques we shall discuss in this chapter.

## 2. Movie Recommendations:

Netflix offers its customers recommendations of movies they might like. These recommendations are based on ratings provided by users, much like the ratings suggested in the example utility matrix of Fig. 9.1.

The importance of predicting ratings accurately is so high, that Netflix offered a prize of one million dollars for the first algorithm that could beat its own recommendation system by 10%.

The prize was finally won in 2009, by a team of researchers called “Bellkor’s Pragmatic Chaos,” after over three years of competition.

### 3. News Articles:

News services have attempted to identify articles of interest to readers, based on the articles that they have read in the past.

The similarity might be based on the similarity of important words in the documents, or on the articles that are read by people with similar reading tastes.

The same principles apply to recommending blogs from among the millions of blogs available, videos on YouTube, or other sites where content is provided regularly

## Populating the Utility Matrix

Without a utility matrix, it is almost impossible to recommend items. However, acquiring data from which to build a utility matrix is often difficult.

There are two general approaches to discovering the value users place on items.

1. We can ask users to rate items. Movie ratings are generally obtained this way, and some on-line stores try to obtain ratings from their purchasers. Sites providing content, such as some news sites or YouTube also ask users to rate items.

This approach is limited in its effectiveness, since generally users are unwilling to provide responses, and the information from those who do may be biased by the very fact that it comes from people willing to provide ratings.

2. We can make inferences from users' behavior. Most obviously, if a user buys a product at Amazon, watches a movie on YouTube, or reads a news article, then the user can be said to "like" this item.

Note that this sort of rating system really has only one value: 1 means that the user likes the item.

Often, we find a utility matrix with this kind of data shown with 0's rather than blanks where the user has not purchased or viewed the item.

However, in this case 0 is not a lower rating than 1; it is no rating at all. More generally, one can infer interest from behavior other than purchasing. For example, if an Amazon customer views information about an item, we can infer that they are interested in the item, even if they don't buy it

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC:CONTENT –BASED RECOMMENDATION SYSTEMS

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

# CONTENT – BASED RECOMMENDATION SYSTEMS

Two basic architectures for a recommendation system:

1. **Content-Based systems** focus on properties of items. Similarity of items is determined by measuring the similarity in their properties.
2. **Collaborative-Filtering systems** focus on the relationship between users and items. Similarity of items is determined by the similarity of the ratings of those items by the users who have rated both items.

# CONTENT – BASED RECOMMENDATION SYSTEMS

Sub-topics to be covered under Content-based  
Recommendation systems:

- ❖ **Item Profiles**
- ❖ **Discovering features of documents**
- ❖ **Obtaining Item features from tags**
- ❖ **Representing Item Profiles**
- ❖ **User Profiles**
- ❖ **Recommending items to users based on content**
- ❖ **Classification Algorithms**

## Item Profiles

**Construct a profile for each item.** A profile is a **record** or collection of records representing **important characteristics of that item**.

In simple cases, the profile consists of some characteristics of the item that are **easily discovered**. For example, consider the **features of a movie** that might be relevant to a recommendation system.

1. The set of **actors** of the movie. Some **viewers prefer movies with their favorite actors**.
2. The **director**. Some viewers have a preference for the work of certain directors.
3. The **year** in which the movie was made. Some viewers **prefer old movies; others watch only the latest releases**.
4. The **genre or general type** of movie. Some viewers like only **comedies, others dramas or romances**.

## Item Profiles

There are many other features of movies that could be used as well.

Except for the last, genre, the information is readily available from descriptions of movies.

Genre is a vaguer concept. However, movie reviews generally assign a genre from a set of commonly used terms. For example the Internet Movie Database (IMDB) assigns a genre or genres to every movie.

Many other classes of items also allow us to obtain features from available data, even if that data must at some point be entered by hand.

For instance, products often have descriptions written by the manufacturer, giving features relevant to that class of product (e.g., the screen size and cabinet color for a TV).

## **Item Profiles**

Books have descriptions similar to those for movies, so we can obtain features such as author, year of publication, and genre.

Music products such as CD's and MP3 downloads have available features such as artist, composer, and genre.

## Discovering Features of Documents

There are other classes of items where it is not immediately apparent what the values of features should be.

We shall consider two of them:

- 1) document collections
- 2) images.

Documents present special problems, and we shall discuss the technology for extracting features from documents in this section.

There are many kinds of documents for which a recommendation system can be useful.

For example, there are many **news articles published each day**, and we cannot read all of them. A recommendation system can suggest articles on topics a user is interested in, but how can we distinguish among topics?

## Discovering Features of Documents

**Web pages are also a collection of documents.** Can we suggest pages a user might want to see? Likewise, **blogs could be recommended to interested users**, if we could classify blogs by topics.

we could classify blogs by topics. Unfortunately, these classes of documents **do not tend to have readily available information giving features**.

A substitute that has been useful in practice is the **identification of words that characterize the topic of a document**.

### How we do the identification?

First, eliminate stop words – the several hundred most common words, which tend to say little about the topic of a document.

## Discovering Features of Documents

For the remaining words, compute the TF.IDF score for each word in the document.

The ones with the highest scores are the words that characterize the document.

We may then take as the features of a document the n words with the highest TF.IDF scores.

It is possible to pick n to be the same for all documents, or to let n be a fixed percentage of the words in the document.

We could also choose to make all words whose TF.IDF scores are above a given threshold to be a part of the feature set.

## Discovering Features of Documents

Now, documents are represented by sets of words. Intuitively, **we expect these words to express the subjects or main ideas of the document.**

For example, in a news article, **we would expect the words with the highest TF.IDF score to include the names of people discussed in the article, unusual properties of the event described, and the location of the event.**

To measure the similarity of two documents, there are several natural distance measures we can use:

1. We could use the Jaccard distance between the sets of words
2. We could use the cosine distance between the sets, treated as vectors.

## Obtaining Item features from tags

Let us consider a **database of images as an example of a way that features have been obtained for items.**

The problem with images is that their data, typically **an array of pixels, does not tell us anything useful about their features.**

We can calculate **simple properties of pixels**, such as the **average amount of red** in the picture, but few users are looking for red pictures or especially like red pictures.

There have been a **number of attempts to obtain information about features of items by inviting users to tag the items** by entering words or phrases that describe the item.

## Obtaining Item features from tags

Thus, one picture with a lot of red might be tagged “Tiananmen Square,” while another is tagged “sunset at Malibu.”

The distinction is not something that could be discovered by existing image-analysis programs.

Almost any kind of data can have its features described by tags. One of the earliest attempts to tag massive amounts of data was the site **del.icio.us, later bought by Yahoo!, which invited users to tag Web pages.**

The goal of this tagging was to make a new method of search available, where users entered a set of tags as their search query, and the system retrieved the Web pages that had been tagged that way.

## Obtaining Item features from tags

However, it is also possible to use the tags as a recommendation system.

If it is observed that a user retrieves or bookmarks many pages with a certain set of tags, then we can recommend other pages with the same tags.

The problem with tagging as an approach to feature discovery is that the process only works if users are willing to take the trouble to create the tags, and there are enough tags that occasional erroneous ones will not bias the system too much.

## Representing Item Profiles

Our ultimate goal for content-based recommendation is to create both an item profile consisting of feature-value pairs and a user profile summarizing the preferences of the user, based on their row of the utility matrix.

We already suggested how an item profile could be constructed. We imagined a vector of 0's and 1's, where a 1 represented the occurrence of a high-TF.IDF word in the document.

Since features for documents were all words, it was easy to represent profiles this way.

We shall try to generalize this vector approach to all sorts of features. It is easy to do so for features that are sets of discrete values.

## Representing Item Profiles

For example, if one feature of movies is the set of actors, then imagine that there is a component for each actor, with 1 if the actor is in the movie, and 0 if not.

Likewise, we can have a component for each possible director, and each possible genre.

All these features can be represented using only 0's and 1's.

There is another class of features that is not readily represented by Boolean vectors: those features that are numerical.

For instance, we might take the average rating for movies to be a feature and this average is a real number.

## Representing Item Profiles

It does not make sense to have one component for each of the possible average ratings, and doing so would cause us to lose the structure implicit in numbers.

That is, two ratings that are close but not identical should be considered more similar than widely differing ratings.

Likewise, numerical features of products, such as screen size or disk capacity for PC's, should be considered similar if their values do not differ greatly.

Numerical features should be represented by single components of vectors representing items. These components hold the exact value of that feature.

## Representing Item Profiles

There is no harm if some components of the vectors are Boolean and others are real-valued or integer-valued.

We can still compute the cosine distance between vectors, although if we do so, we should give some thought to the appropriate scaling of the non-Boolean components, so that they neither dominate the calculation nor are they irrelevant.

## Representing Item Profiles

**Example 9.2:** Suppose the only features of movies are the set of actors and the average rating. Consider two movies with five actors each. Two of the actors are in both movies. Also, one movie has an average rating of 3 and the other an average of 4. The vectors look something like

$$\begin{array}{ccccccccc} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 3\alpha \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 4\alpha \end{array}$$

However, there are in principle an infinite number of additional components, each with 0's for both vectors, representing all the possible actors that neither movie has. Since cosine distance of vectors is not affected by components in which both vectors have 0, we need not worry about the effect of actors that are in neither movie.

## Representing Item Profiles

The last component shown represents the average rating. We have shown it as having an unknown scaling factor  $\alpha$ . In terms of  $\alpha$ , we can compute the cosine of the angle between the vectors. The dot product is  $2 + 12\alpha^2$ , and the lengths of the vectors are  $\sqrt{5 + 9\alpha^2}$  and  $\sqrt{5 + 16\alpha^2}$ . Thus, the cosine of the angle between the vectors is

$$\frac{2 + 12\alpha^2}{\sqrt{25 + 125\alpha^2 + 144\alpha^4}}$$

## Representing Item Profiles

If we choose  $\alpha = 1$ , that is, we take the average ratings as they are, then the value of the above expression is 0.816.

If we use  $\alpha = 2$ , that is, we double the ratings, then the cosine is 0.940. That is, the vectors appear much closer in direction than if we use  $\alpha = 1$ .

Likewise, if we use  $\alpha = 1/2$ , then the cosine is 0.619, making the vectors look quite different.

We cannot tell which value of  $\alpha$  is “right,” but we see that the choice of scaling factor for numerical features affects our decision about how similar items are.

## User Profiles

We not only need to create vectors describing items; we need to **create vectors with the same components that describe the user's preferences.**

We have the **utility matrix** representing the connection between users and items.

Recall the **nonblank matrix entries could be just 1's representing user purchases or a similar connection, or they could be arbitrary numbers representing a rating or degree of affection that the user has for the item.**

With this information, **the best estimate we can make regarding which items the user likes is some aggregation of the profiles of those items**

If the utility matrix has only 1's, then the natural aggregate is the average of the components of the vectors representing the item profiles for the items in which the utility matrix has 1 for that user.

Example 9.3 : Suppose items are movies, represented by Boolean profiles with components corresponding to actors.

Also, the utility matrix has a 1 if the user has seen the movie and is blank otherwise.

If 20% of the movies that user U likes have Julia Roberts as one of the actors, then the user profile for U will have 0.2 in the component for Julia Roberts.

If the utility matrix is not Boolean, e.g., ratings 1–5, then we can weight the vectors representing the profiles of items by the utility value.

It makes sense to normalize the utilities by subtracting the average value for a user.

That way, we get negative weights for items with a below-average rating, and positive weights for items with above-average ratings.

That effect will prove useful when we discuss in Section 9.2.6 how to find items that a user should like.

## User Profiles

Example 9.4 : Consider the same movie information as in Example 9.3, but now suppose **the utility matrix has nonblank entries that are ratings in the 1–5 range.**

Suppose user **U gives an average rating of 3**. There are three movies with Julia Roberts as an actor, and those movies got ratings of 3, 4, and 5.

Then in the user profile of U, the component for Julia Roberts will have value that is the average of 3 – 3, 4 – 3, and 5 – 3, that is, a value of 1.

On the other hand, user V gives an average rating of 4, and has also rated three movies with Julia Roberts (it doesn't matter whether or not they are the same three movies U rated).

## User Profiles

User V gives these three movies ratings of 2, 3, and 5.

The user profile for V has, in the component for Julia Roberts, the average of 2 – 4, 3 – 4, and 5 – 4, that is, the value  $-2/3$ .

## Recommending Items to Users based on content

**With profile vectors for both users and items**, we can estimate the degree to which a user would prefer an item by computing the cosine distance between the user's and item's vectors.

As in Example 9.2, we may wish to scale various components whose values are not Boolean.

The random-hyperplane and locality-sensitive-hashing techniques can be used to place (just) item profiles in buckets. In that way, given a user to whom we want to recommend some items, we can apply the same two techniques – random hyperplanes and LSH – to determine in which buckets we must look for items that might have a small cosine distance from the user.

## Recommending Items to Users based on content

Example 9.5 :

Now, consider Example 9.4.

There, we observed that the vector for a user will have positive numbers for actors that tend to appear in movies the user likes and negative numbers for actors that tend to appear in movies the user doesn't like.

## Recommending Items to Users based on content

Consider a movie with many actors the user likes, and only a few or none that the user doesn't like.

The cosine of the angle between the user's and movie's vectors will be a large positive fraction.

That implies an angle close to 0, and therefore a small cosine distance between the vectors.

Next, consider a movie with about as many actors that the user likes as those the user doesn't like.

## Recommending Items to Users based on content

In this situation, the cosine of the angle between the user and movie is around 0, and therefore the angle between the two vectors is around 90 degrees.

Finally, consider a movie with mostly actors the user doesn't like.

In that case, the cosine will be a large negative fraction, and the angle between the two vectors will be close to 180 degrees – the maximum possible cosine distance.

## Classification Algorithms

**A completely different approach to a recommendation system** using item profiles and utility matrices is to treat the problem as one of **machine learning**.

Regard the given data as a training set, and **for each user, build a classifier that predicts the rating of all items.**

There are a **great number of different classifiers**, and it is not our purpose to teach this subject here.

However, you should be aware of the option of developing a classifier for recommendation, so we shall discuss one common classifier – decision trees – briefly.

## Classification Algorithms

A decision tree is a collection of nodes, arranged as a binary tree.

The leaves render decisions; in our case, the decision would be “likes” or “doesn’t like.”

Each interior node is a condition on the objects being classified; in our case the condition would be a predicate involving one or more features of an item.

To classify an item, we start at the root, and apply the predicate at the root to the item.

If the predicate is true, go to the left child, and if it is false, go to the right child. Then repeat the same process at the node visited, until a leaf is reached. That leaf classifies the item as liked or not.

## Classification Algorithms

Construction of a decision tree requires selection of a predicate for each interior node.

There are many ways of picking the best predicate, but they all try to arrange that one of the children gets all or most of the positive examples in the training set (i.e, the items that the given user likes, in our case) and the other child gets all or most of the negative examples (the items this user does not like).

Once we have selected a predicate for a node N, we divide the items into the two groups:

those that satisfy the predicate and those that do not.

## Classification Algorithms

For each group, we again find the predicate that best separates the positive and negative examples in that group.

These predicates are assigned to the children of N.

This process of dividing the examples and building children can proceed to any number of levels.

We can stop, and create a leaf, if the group of items for a node is homogeneous; i.e., they are all positive or all negative examples

## Classification Algorithms

However, we may wish to stop and create a leaf with the majority decision for a group, even if the group contains both positive and negative examples.

The reason is that the statistical significance of a small group may not be high enough to rely on.

For that reason a variant strategy is to create an ensemble of decision trees, each using different predicates, but allow the trees to be deeper than what the available data justifies. Such trees are called overfitted.

To classify an item, apply all the trees in the ensemble, and let them vote on the outcome. We shall not consider this option here, but give a simple hypothetical example of a decision tree.

## Classification Algorithms

Example 9.6 : Suppose our items are news articles, and features are the highTF.IDF words (keywords) in those documents.

Further suppose there is a user U who likes articles about baseball, except articles about the New York Yankees.

The row of the utility matrix for U has 1 if U has read the article and is blank if not.

We shall take the 1's as “like” and the blanks as “doesn't like.” Predicates will be Boolean expressions of keywords.

Since U generally likes baseball, we might find that the best predicate for the root is “homerun” OR (“batter” AND “pitcher”).

## Classification Algorithms

Items that satisfy the predicate will tend to be positive examples (articles with 1 in the row for U in the utility matrix), and items that fail to satisfy the predicate will tend to be negative examples (blanks in the utility-matrix row for U).

Figure 9.3 shows the root as well as the rest of the decision tree.

Suppose that the group of articles that do not satisfy the predicate includes sufficiently few positive examples that we can conclude all of these items are in the “don’t-like” class.

We may then put a leaf with decision “don’t like” as the right child of the root.

## Classification Algorithms

However, the articles that satisfy the predicate includes a number of articles that user U doesn't like; these are the articles that mention the Yankees.

Thus, at the left child of the root, we build another predicate.

We might find that the predicate “Yankees” OR “Jeter” OR “Teixeira” is the best possible indicator of an article about baseball and about the Yankees.

Thus, we see in Fig. 9.3 the left child of the root, which applies this predicate.

## Classification Algorithms

Both children of this node are leaves, since we may suppose that the items satisfying this predicate are predominantly negative and those not satisfying it are predominantly positive.

Unfortunately, classifiers of all types tend to take a long time to construct. For instance, if we wish to use decision trees, we need one tree per user.

Constructing a tree not only requires that we look at all the item profiles, but we have to consider many different predicates, which could involve complex combinations of features.

Thus, this approach tends to be used only for relatively small problem sizes.

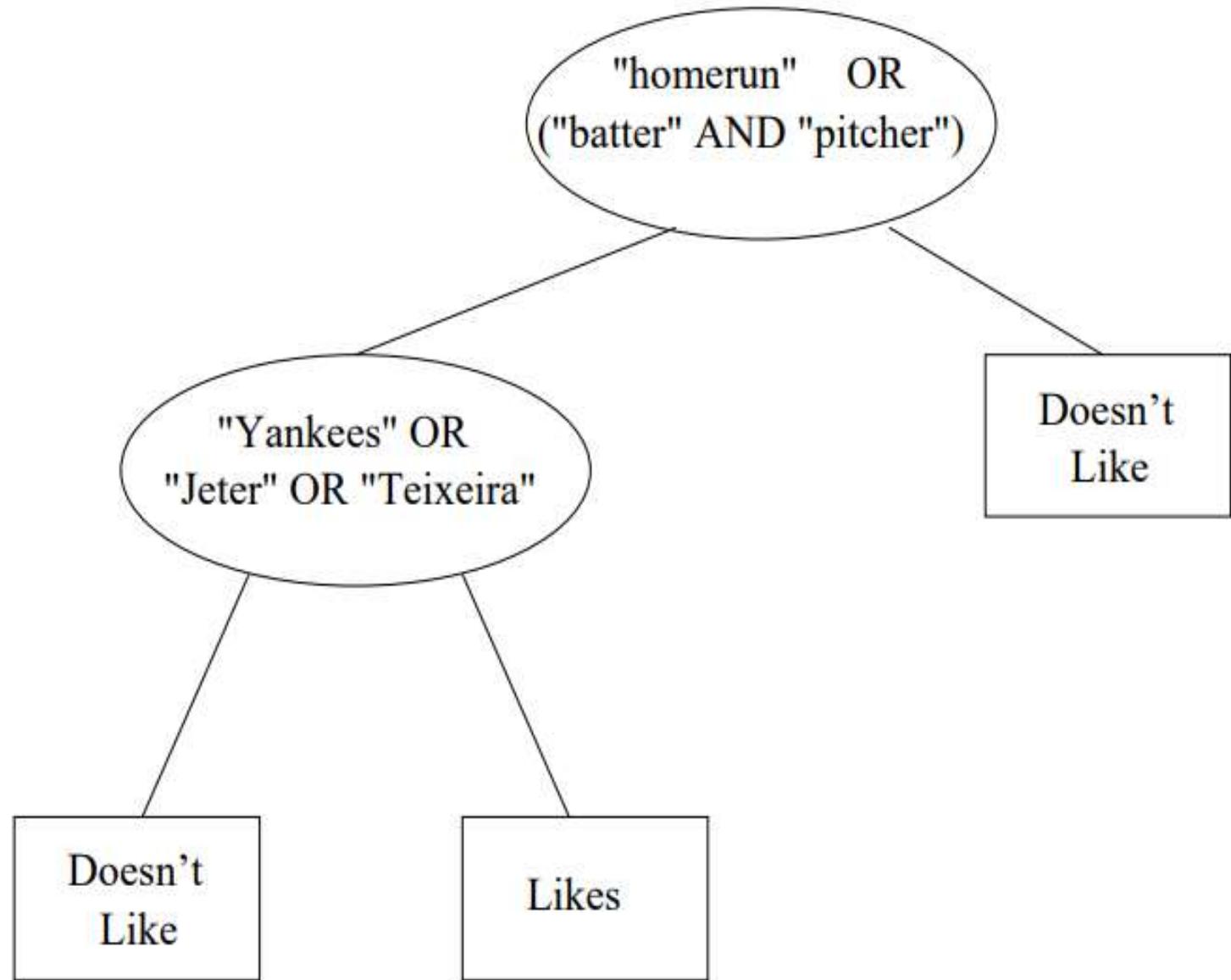


Figure 9.3: A decision tree

## Collaborative Filtering

1. Measuring similarity
  2. Quality of similarity
  3. Clustering of users and items

User	HP1	HP2	HP3	TW	SW1	SW2	CW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

Given an user U → to recommend

$$\begin{array}{rcl} \text{Movies} & \rightarrow & \\ \frac{4+5+1}{3} & = 10/3 \\ \frac{5+5+4}{3} & = 14/3 \\ \frac{2+4+5}{3} & = 11/3 \\ \frac{3+3}{2} & = 3 \end{array}$$

→ Average

Given an user  $U \rightarrow$  to recommend an item

Find similar users → with same taste as  $\cup$

Recommend similar user's preference to user U.

## Two distance measures

① Jaccard distance → X wrong Does not work well with  
 ② Cosine similarity stating's ⇒ Do Rounding off  
 to 1,0 -

## Jaccard Similarity

Correct answer

$$JS[A, C] = \frac{|A \cap C|}{|A \cup C|} = \frac{2}{4} = 0.5$$

Jaccard Distance =

$$1 - JS = 1 - 0.5 = 0.5$$

$$JD(A, C) = 0.5$$

$$JS(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{1}{5} = 0.2$$

$$JD(A, B) = 1 - JS(A, B) = 1 - 0.2 = 0.8$$

$$\text{JSD}(A, B) = 0.8$$

## Cosine Distance

$$\text{Cosine } (A, c) = \frac{A \cdot c}{|A| \cdot |c|}$$

$$A = \begin{bmatrix} 4 & 0 & 0 & 5 & 1 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 2 & 4 & 5 & 0 \end{bmatrix}$$

$$\therefore \frac{(5 \cdot 2) + (1 \cdot 4)}{\sqrt{4^2 + 5^2 + 1^2} \cdot \sqrt{2^2 + 4^2 + 5^2}}$$

$$\cos(\theta_c) = 0.38^{\circ}$$

$$\text{Cosine } (A, B) = 0.322$$

$\Rightarrow$  Cosine distance gives correct conclusion.

Rounding the Data

High Ratings  $\rightarrow 1$

Low Ratings  $\rightarrow 0$

User	HP <sub>1</sub>	HP <sub>2</sub>	HP <sub>3</sub>	TW	SW <sub>1</sub>	SW <sub>2</sub>	SW <sub>3</sub>
A	1				1		
B	1	1				1	1
C							
D			1				1

$$JS(A, C) = \frac{|A \cap C|}{|A \cup C|} = \frac{0}{4} = 0 \quad JD(A, C) = 1 - 0 = 1$$

$$JS(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{1}{4} = 0.25 \quad JD(A, B) = 1 - 0.25 = 0.75$$

Normalized ratings

Higher Ratings  $\Rightarrow +ve$

Lower Ratings  $\Rightarrow -ve$

Normalized Rating = Rating - Average rating

Distance  $\rightarrow + \rightarrow$  Vectors in same dir.

Distance  $\rightarrow - \rightarrow$  Opposite taste

$\downarrow$   
Vectors in different direction

" To improve  
Cosine distance "

User	HP <sub>1</sub>	HP <sub>2</sub>	HP <sub>3</sub>	TW	SW <sub>1</sub>	SW <sub>2</sub>	SW <sub>3</sub>
A	$\frac{4-10}{3}$ $= -2/3$			$\frac{5-10}{3}$ $= -5/3$	$\frac{1-10}{3}$ $= -7/3$		
B	$1/3$	$1/3$	$-2/3$				
C				$-5/3$	$1/3$	$1/3$	
D		0					0

$$\text{Cosine sim}(A, C) = \frac{(5/3)(-5/3) + (-7/3)(1/3)}{\sqrt{\frac{4}{9} + \frac{25}{9} + \frac{49}{9}} \sqrt{\frac{25}{9} + \frac{1}{9} + \frac{1}{9}}} = \frac{-3.55}{(2.94)(1.732)} = \frac{-3.55}{5.105} = 0.697$$

$$\text{Cosine dis}(A, C) = 1 - \text{CS}(A, C) = 1 - 0.697 = 0.303 //$$

## Quality of Similarity

1. User based : If one user is similar to other user
2. Item based : If one item is similar to other item.

① User U → Similar user U'

U' likes item I → Directly recommend I to U

② But not the same case with item based similarity

User U → likes item I

Item I' → Similar to I

Cannot directly recommend I to U

Deal with context, preference, novelty

Not only recommend just similar  
but the item should be new to  
user.

13.09.2024

## Clustering of Users & Items

User	HP <sub>1</sub>	HP <sub>2</sub>	HP <sub>3</sub>	TW	SW <sub>1</sub>	SW <sub>2</sub>	SW <sub>3</sub>
A	4			5			
B	5	5	4				
C				2	4	5	
D		3					3

Group similar items into Item cluster  
↓

User	HP Item cluster 1	TW Item cluster 2	SW Item Cluster 3
A	4	5	
B	14/3		
C		2	9/2
D	3		3

Find average ratings

Group similar users into User cluster  
↓

User cluster	IC <sub>1</sub>	IC <sub>2</sub>	IC <sub>3</sub>
UC <sub>1</sub>			
UC <sub>2</sub>			

→ "Cluster Cluster  
Utility Matrix"

To find the missing values, in original table

A - HP<sub>2</sub>

- In which cluster  $\text{HP}_2$  is present

In which A is present

$$\underline{U_{C_1}} - \underline{I_{C_1}}$$

Even if  $U_C - IC$  is empty, try to cluster the users again.

## ④ Dimensionality Reduction      U-V Decomposition

$n$  users,  $m$  items  $\Rightarrow$  Matrix M

Choose a smaller matrix  $U_{n \times d}$  and  $V_{d \times m}$

$P$  should closely approximate  $M$

$$P = U \cdot V$$

$$(n \times d) \quad (d \times m)$$

To find missing entries in M

$$P = n \times m$$

Do many iterations such that  
 $P$  should closely resemble  $M$ .

If RMSE does not  
reduce in further iterations

"STOP"

## Singular Value Decomposition (SVD)

UVD is a special case of SVD.

- 1. U-V decomposition      UV-D is a special
  - 2. Root Mean Square Error
  - 3. Incremental Calculation of UV-Decomposition
  - 4. Optimising an arbitrary element
  - 5. Building a complete U-V decomposition algorithm

01104

$$\begin{bmatrix} 5 & 2 & 4 & 4 & 3 \\ 3 & 1 & 2 & 4 & 1 \\ 2 & \square & 3 & 1 & 4 \\ 2 & 5 & 4 & 3 & 5 \\ 4 & 4 & 5 & 4 & \square \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \\ U_{31} & U_{32} \\ U_{41} & U_{42} \\ U_{51} & U_{52} \end{bmatrix} \begin{bmatrix} V_{11} & V_{12} & V_{13} & V_{14} & V_{15} \\ V_{21} & V_{22} & V_{23} & V_{24} & V_{25} \end{bmatrix}$$

no. of non-blank elements in matrix = 23 > 20

Not possible to find exact matrix M

## Dimensionality Reduction

①

- An entirely different approach to estimate the blank entries in the utility matrix is to assume that : the utility matrix is actually the product of 2 long, thin matrices.
- We'll discuss one such approach to discover such matrices :  $U \times V$  Decomposition  

↓  
It is an instance of a More General theory called Singular-Value Decomposition (SVD).

- (1) UV-Decomposition
- (2) Root-Mean-Square (RMS) Error
- (3) Incremental Computation of a UV-Decomposition
- (4) Optimizing an Arbitrary element
- (5) Building a complete UV-Decomposition algorithm

### (1) UV-Decomposition

- It is a type of matrix factorization used in recommendation systems to predict missing (blank) values in a utility matrix.
- Imagine you have a utility matrix  $M$  with  $n$  rows (User) &  $m$  columns (Items).
- Each entry in the matrix represents how much a user likes a particular item given as a rating (1-5).

Purpose of  
U-V-decomposition

- Some entries will be filled & others will be blank (unrated).
- The Goal of U-V-decomposition is to approximate <sup>this</sup> matrix  $M$  by breaking it into 2 smaller matrices  $U$  &  $V$ .
- [It allows to reduce the complexity of the data.]  
Instead of working with a large matrix  $M$  we work with 2 smaller matrices  $U$  &  $V$ .]
- Matrix  $U$  has  $n$  rows ( $n$  users) &  $d$  columns ( $d$  items) → some items
- Matrix  $V$  has  $d$  rows ( $d$  users) &  $m$  columns ( $m$  items) → some users
- By multiplying  $U$  &  $V$ , we get an approximation of the original Matrix  $M$ .
- Product  $UV$  should closely approximate the known ratings in  $M$ .
- More importantly, for the blank entries in  $M$ , the product  $UV$  gives an estimated rating based on the patterns found in data.

## Eg: UV Decomposition of Matrix M

$$\begin{bmatrix} 5 & 2 & 4 & 4 & 3 \\ 3 & 1 & 2 & 4 & 1 \\ 2 & \boxed{\quad} & 3 & 1 & 4 \\ 2 & \cancel{5} & 4 & 3 & 5 \\ 4 & 4 & 5 & 4 & \boxed{\quad} \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \\ v_{31} & v_{32} \\ v_{41} & v_{42} \\ v_{51} & v_{52} \end{bmatrix} \times \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} & v_{15} \\ v_{21} & v_{22} & v_{23} & v_{24} & v_{25} \end{bmatrix}$$

Matrix V  
(2x5)

Matrix U (5x2)

- 5x5 matrix M
- all entries of M are known except 2 entries
- we wish to decompose M into

$U \times V$

-  $U \rightarrow 5 \times 2$  matrix

-  $V \rightarrow 2 \times 5$  "

The matrices M, U & V are shown above.  
The variables M, U & V are to be determined

- Entries of U & V are

Note: No. of entries in M (23) > combined no. of entries in U & V (20)

$\therefore U \times V$  will not exactly agree on the non-blank entries of M.

## (2) Root Mean Square Error (RMSE)

To know whether the UV decomposition correctly approximates the original matrix M,

we can use the RMSE measure.

If RMSE is low, then UV gives a

good approximation of original matrix M.

So, we try to achieve a minimum RMSE.

Steps to find RMSE: (Do the following steps for every nonblank entry in  $UV$ )

- (1) Subtract every nonblank entry in  $UV$  from the corresponding entry in  $M$

Difference = Entry in  $M$  - Entry in  $UV$

- (2) Square the difference and sum

- (3) Take mean of those squares

$$\text{Mean of squares} = \frac{\text{Sum of squares}}{\text{No. of nonblank entries in } M}$$

- (4) Take square root of the mean.

This is the Root mean square

→ In general, last 2 steps are omitted because

Minimizing the sum of squares is same

as minimizing the square root of the mean square.

Eg:- Suppose  $U$  &  $V$  are having all entries as 1

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}_{5 \times 2} \text{ Motion } U \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}_{2 \times 5} \text{ Motion } V = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}_{5 \times 5} \text{ UV}$$

We will find the RMSE

using  $UV$  matrix & the original matrix  $M$  by following steps (1) to (4) given above.

(3)

$$\text{Matrix } UXV = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

$$\text{Original Matrix } M = \begin{bmatrix} 5 & 2 & 4 & 4 & 3 \\ 3 & 1 & 2 & 4 & 1 \\ 2 & -3 & 1 & 4 & \\ 2 & 5 & 4 & 3 & 5 \\ 4 & 5 & 4 & & \end{bmatrix}$$

Step 1

1<sup>st</sup> row:  $(5-2)^2 + (2-2)^2 + (4-2)^2 + (4-2)^2 + (3-2)^2$   
 $3^2 + 0 + 2^2 + 2^2 + 1^2$   
 $= 9 + 4 + 4 + 1$   
 $= 18$

2<sup>nd</sup> row:  $(3-2)^2 + (1-2)^2 + (2-2)^2 + (4-2)^2 + (1-2)^2$   
 $= 1 + 1 + 0 + 4 + 1$   
 $= 7$

Now we find for other rows:

Sum the sums of all 5 rows

$$18 + 7 + 6 + 8 + 21 = 75$$

Find mean:  $\frac{75}{\text{No. of nonblank entries in } M} = \frac{75}{23}$

$$\text{Mean} = \frac{75}{23}$$

$$\left. \begin{array}{l} \text{Root mean square error (RMSE)} \\ \text{RMSE} = \sqrt{\frac{75}{23}} \end{array} \right\} = \sqrt{\frac{75}{23}}$$

$$\boxed{\text{RMSE} = 1.806}$$

(3)

### Incremental Computation of a UV-Decomposition

To achieve a UV-decomposition with least RMSE;

- \* we start with some arbitrarily chosen  $U$  &  $V$  & Repeatedly adjust  $U$  &  $V$  to make RMSE small.

- \* we can do adjustments to a single element of  $U$  or  $V$ . However, complex adjustments are also possible.

Eg: we shall start with  $U$  &  $V$  with all entries as 1.

$$U_{11} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad V = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

We decide to alter  $U_{11}$  to reduce RMSE.

Let  $U_{11} = n$

Now, the updated  $UV$  is:

$$\begin{bmatrix} n & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} n+1 & n+1 & n+1 & n+1 & n+1 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix} = UXV$$

only the 1st row of  $UXV$  has changed

So, we compute RMSE by the effect of 1<sup>st</sup> row:

i.e) Entry of  $UV$  - Entry of  $UXV$

$$(x+1) - (5-(x+1))^2 + (2-(x+1))^2 + (4-(x+1))^2 + (3-(x+1))^2$$

The above sum is simplified as below:

(4)

$$(5-x-1)^2 + (2-x-1)^2 + (4-x-1)^2 + (4-x-1)^2 + (3-x-1)^2$$

$$\Rightarrow (4-x)^2 + (1-x)^2 + (3-x)^2 + (3-x)^2 + (2-x)^2$$

We want the value of  $x$  to minimize the above sum. So, we take the derivative & set that equal to 0:

$$-2 \times ((4-x) + (1-x) + (3-x) + (3-x) + (2-x)) = 0$$

$$\text{or} \quad -2(4+1+3+3+2 - 5x) = 0$$

$$-2(13 - 5x) = 0$$

$$13 - 5x = 0$$

$$13 = 5x \quad \text{or} \quad 5x = 13$$

$$x = 13/5$$

$$\boxed{x = 2.6}$$

Put  $x$  value in  $U_{11}$ ,

$$\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3.6 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

(The best value for  $U_{11}$   
is found to be 2.6)

Now, 1st row component in RMSE is:

$$(5-3.6)^2 + (2-3.6)^2 + (4-3.6)^2 + (4-3.6)^2 + (3-3.6)^2$$

$$= 5.2$$

Earlier it was 18, now it is reduced to 5.2  
(Refer page 3)

Let us choose  $V_{11}$  & alter it ( $V_{11}=y$ ) to reduce RMSE

$$\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} y & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2.6y+1 & 3.6 & 3.6 & 3.6 & 3.6 \\ y+1 & 2 & 2 & 2 & 2 \\ y+1 & 2 & 2 & 2 & 2 \\ y+1 & 2 & 2 & 2 & 2 \\ y+1 & 2 & 2 & 2 & 2 \end{bmatrix}$$

Hence, only the 1st column of the product is affected by  $y$ .

So, we find the sum of the squares of the differences b/w the entries in the first column of  $M$  &  $UV$  only.

$$= (5 - (2.6y+1))^2 + (3 - (y+1))^2 + (2 - (y+1))^2 + (1 - (y+1))^2$$

$$\Rightarrow (4 - 2.6y)^2 + (2 - y)^2 + (1 - y)^2 + (1 - y)^2 + (3 - y)^2$$

To find a value of  $y$  that minimizes the sum, we differentiate the above eqn. & equate to 0.

$$-2 \times (2.6(4 - 2.6y) + (2 - y) + (1 - y) + (3 - y)) = 0$$

Solving for  $y$

$$\text{we get } y = \frac{17.4}{10.76} = 1.617$$

$$\boxed{y = 1.617}.$$

Replace  $y$  by 1.617 in the above  $UV$  matrices we get as follows:

$$\begin{bmatrix} 2.6 & & \\ & \vdots & \vdots \\ & \vdots & \vdots \end{bmatrix} \times \begin{bmatrix} 1.617 & & & & \\ & \vdots & \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} 5.204 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \end{bmatrix} \quad (5)$$

#### (A) Optimizing an arbitrary element

Now, let us develop the General formula for the optimum value of an element.

→ Let  $M$  be a  $n \times m$  utility matrix with some entries blank

→ Let  $U$  be a matrix ( $n \times d$ )  $V$  be a matrix ( $d \times m$ ) for some  $d$ .

$$\text{Let } P = UV$$

→ Let  $m_{ij}$ ,  $u_{ij}$  &  $v_{ij}$  denote the entries in row  $i$  & column  $j$  of  $M$ ,  $U$  &  $V$  matrices.

→  $P_{ij}$  denote the entry in row  $i$ , col  $j$  of  $P$ .

→ Suppose we choose an entry  $u_{ij}$  to vary in order to minimize RMSE b/w  $M$  &  $UV$ .

→ Varying  $u_{ij}$  affects only the elements in row  $i$  of the Product  $P = UV$ .

So, we are concerned with that row only

$$P_{ij} = \sum_{k=1}^d u_{ik} \cdot v_{kj}$$

for all values of  $j$

&  $m_{ij}$  is a nonblank entry.

Sum of the squares of the difference  
between entries in  $M$  &  $UV$  for  
row  $i$  & col  $j$ :

$$(m_{ij} - p_{ij})^2 = \left( m_{ij} - \sum_{k=1}^d u_{ik} \cdot v_{kj} \right)^2$$

In this way, we try to minimize RMSE  
by optimizing an arbitrary element.

### (5) Building a complete UV-Decomposition algorithm

We have tools to search for the global optimum decomposition of a utility matrix  $M$ .  
Four areas of importance are:

- (a) Preprocessing of Matrix  $M$
- (b) Initializing  $U$  &  $V$
- (c) Ordering the optimization of the elements  $U$  &  $V$ .
- (d) Ending the attempt at optimization

#### (a) Preprocessing

\* The important factors in determining the missing elements of the matrix  $M$  include:

Differences in rating scales  
of men.

\* In preprocessing step, we remove the influence of such factors by any of the following logic:

(b)

(a) From each non-blank element  $m_{ij}$ , subtract the average rating of user  $i$ . In the resulting matrix, subtract average rating of item  $j$  from each entry.

(a)

(b)

From each non-blank element  $m_{ij}$ , first subtract average rating of item  $j$  & then subtract average rating of user  $i$  in the modified matrix.

(b)

(c) Do normalization by subtracting from  $m_{ij}$ , the average of the (average rating of user  $i$  & item  $j$ ).

In this option, if you choose option (b), then, before making predictions, we have to undo the normalization.

## (b) Initialization

- It is essential to introduce some randomness in the way we seek an optimum solution.

- This is to ensure that we have many local minima in the hope of reaching the global minimum.

A simple starting point for  $U$  &  $V$  is:

Set each element of  $U$  &  $V$  to  $\boxed{\sqrt{a/d}}$ .

Where  $a \rightarrow$  average of the non-blank entries in  $M$

$d \rightarrow$  one of the dimensions of the  $U$  &  $V$  matrices.

→ To get different values for  $U$  &  $V$ , we can modify  $\sqrt{w/d}$  randomly for each of these elements.

(1) Add a normally distributed value (with mean 0) to each element & some std.deviation

(2) Add a value uniformly chosen from a range  $-c$  to  $+c$  for some  $c$ .

### Performing the optimization

From a given starting value of  $U$  &  $V$ , if we wish to achieve a good improvement in RMSE, then we need to pick an order in which we visit the elements of  $U$  &  $V$ .

i) Pick the elements row-by-row for the elements of  $U$  &  $V$

ii) Visit them in a round-robin fashion.

Note that just because we optimized an element once does not mean we should not optimize it again.

So, we can adjust the elements repeatedly until no further improvements are possible.

## converging to a Minimum

- Ideally at some point the RMSE becomes 0
- But in practice, RMSE cannot be reduced to 0 as the no. of non-blank entries in  $M$  are more than those in  $U \& V$  together.
- So, we have to track the amount of improvement in the RMSE in each round of optimization. If the improvement falls below threshold, stop the process of improving RMSE.  $\frac{(RMSE)}{RMSE_{\text{initial}}}$  has converged to the minimum value possible.

## Avoiding Overfitting

Although RMSE may be small on the given data, it doesn't do well in predicting the future data.

This problem is called Overfitting.

Techniques to cope up with Overfitting problem:

- (1) Stop revisiting elements of  $U \& V$  well before the process has converged.
- (2) Take several different

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC:Social Networks As Graphs

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

# Social Networks As Graphs

## (a)What is a Social Network?

When we think of a social network, we think of Facebook, Twitter, Google+, or another website that is called a “social network,” and indeed this kind of network is representative of the broader class of networks called “social.”

The essential characteristics of a social network are:

1. There is a collection of entities that participate in the network. Typically, these entities are people, but they could be something else entirely.
2. There is at least one relationship between entities of the network. On Facebook or its ilk, this relationship is called friends. Sometimes the relationship is all-or-nothing;

two people are either friends or they are not. However, in other examples of social networks, the relationship has a degree.

This degree could be discrete; e.g., friends, family, acquaintances, or none as in Google+. It could be a real number; an example would be the fraction of the average day that two people spend talking to each other.

3. There is an assumption of nonrandomness or locality. This condition is the hardest to formalize, but the intuition is that relationships tend to cluster.

That is, if entity A is related to both B and C, then there is a higher probability than average that B and C are related.

## **(b) Social Networks as Graphs**

Social networks are naturally modeled as graphs, which we sometimes refer to as a social graph.

The entities are the nodes, and an edge connects two nodes if the nodes are related by the relationship that characterizes the network.

If there is a degree associated with the relationship, this degree is represented by labeling the edges. Often, social graphs are undirected, as for the Facebook friends graph. But they can be directed graphs, as for example the graphs of followers on Twitter or Google+.

Example 10.1 : Figure 10.1 is an example of a tiny social network. The entities are the nodes A through G. The relationship, which we might think of as “friends,” is represented by the edges. For instance, B is friends with A, C, and D. Is this graph really typical of a social network, in the sense that it exhibits locality of relationships? First, note that the graph has nine edges out of the  $\binom{7}{2} = 21$  pairs of nodes that could have had an edge between them.

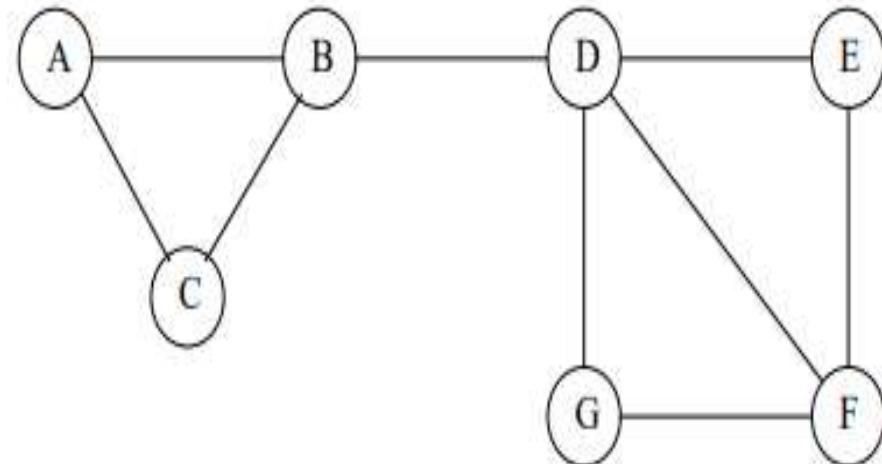


Figure 10.1: Example of a small social network

Suppose X, Y , and Z are nodes of Fig. 10.1, with edges between X and Y and also between X and Z.

What would we expect the probability of an edge between Y and Z to be? If the graph were large, that probability would be very close to the fraction of the pairs of nodes that have edges between them, i.e.,  $9/21 = .429$  in this case.

Since we already know there are edges (X, Y ) and (X, Z), there are only seven edges remaining. Those seven edges could run between any of the 19 remaining pairs of nodes. Thus, the probability of an edge (Y, Z) is  $7/19 = .368$ .

Now, we must compute the probability that the edge (Y, Z) exists in Fig. 10.1, given that edges (X, Y ) and (X, Z) exist.

What we shall actually count is pairs of nodes that could be Y and Z, without worrying about which node is Y and which is Z.

If X is A, then Y and Z must be B and C, in some order. Since the edge (B, C) exists, A contributes one positive example (where the edge does exist) and no negative examples (where the edge is absent).

The cases where X is C, E, or G are essentially the same. In each case, X has only two neighbors, and the edge between the neighbors exists. Thus, we have seen four positive examples and zero negative examples so far.

Now, consider X = F. F has three neighbors, D, E, and G. There are edges between two of the three pairs of neighbors, but no edge between G and E.

Thus, we see two more positive examples and we see our first negative example. If  $X = B$ , there are again three neighbors, but only one pair of neighbors, A and C, has an edge.

Thus, we have two more negative examples, and one positive example, for a total of seven positive and three negative. Finally, when  $X = D$ , there are four neighbors. Of the six pairs of neighbors, only two have edges between them.

Thus, the total number of positive examples is nine and the total number of negative examples is seven. We see that in Fig. 10.1, the fraction of times the third edge exists is thus  $9/16 = .563$ .

This fraction is considerably greater than the  $.368$  expected value for that fraction. We conclude that Fig. 10.1 does indeed exhibit the locality expected in a social network.

## **(c) Varieties of Social Networks**

There are many examples of social networks other than “friends” networks. Here, let us enumerate some of the other examples of networks that also exhibit locality of relationships.

**(c.1) Telephone Networks** Here the nodes represent phone numbers, which are really individuals. There is an edge between two nodes if a call has been placed between those phones in some fixed period of time, such as last month, or “ever.”

The edges could be weighted by the number of calls made between these phones during the period. Communities in a telephone network will form from groups of people that communicate frequently:

groups of friends, members of a club, or people working at the same company, for example.

## (c.2)Email Networks

The nodes represent email addresses, which are again individuals. An edge represents the fact that there was at least one email in at least one direction between the two addresses.

Alternatively, we may only place an edge if there were emails in both directions. In that way, we avoid viewing spammers as “friends” with all their victims. Another approach is to label edges as weak or strong.

Strong edges represent communication in both directions, while weak edges indicate that the communication was in one direction only.

The communities seen in email networks come from the same sorts of groupings we mentioned in connection with telephone networks.

A similar sort of network involves people who text other people through their cell phones.

### **(c.3)Collaboration Networks**

Nodes represent individuals who have published research papers. There is an edge between two individuals who published one or more papers jointly.

Optionally, we can label edges by the number of joint publications. The communities in this network are authors working on a particular topic.

An alternative view of the same data is as a graph in which the nodes are papers. Two papers are connected by an edge if they have at least one author in common. Now, we form communities that are collections of papers on the same topic.

## Other Examples of Social Graphs

Many other phenomena give rise to graphs that look something like social graphs, especially exhibiting locality.

Examples include: information networks (documents, web graphs, patents), infrastructure networks (roads, planes, water pipes, powergrids), biological networks (genes, proteins, food-webs of animals eating each other), as well as other types, like product co-purchasing networks (e.g., Groupon).

**(d) Graphs With Several Node Types** There are other social phenomena that involve entities of different types. We just discussed under the heading of “collaboration networks,” several kinds of graphs that are really formed from two types of nodes.

Authorship networks can be seen to have author nodes and paper nodes. In the discussion above, we built two social networks by eliminating the nodes of one of the two types, but we do not have to do that. We can rather think of the structure as a whole.

For a more complex example, users at a site like deli.cio.us place tags on Web pages.

There are thus three different kinds of entities: users, tags, and pages.

We might think that users were somehow connected if they tended to use the same tags frequently, or if they tended to tag the same pages.

Similarly, tags could be considered related if they appeared on the same pages or were used by the same users, and pages could be considered similar if they had many of the same tags or were tagged by many of the same users.

The natural way to represent such information is as a  $k$ -partite graph for some  $k > 1$ . We know bipartite graphs, the case  $k = 2$ . In general, a  $k$ -partite graph consists of  $k$  disjoint sets of nodes, with no edges between nodes of the same set.

Figure 10.2 is an example of a tripartite graph (the case  $k = 3$  of a  $k$ -partite graph). There are three sets of nodes, which we may think of as users  $\{U_1, U_2\}$ , tags  $\{T_1, T_2, T_3, T_4\}$ , and Web pages  $\{W_1, W_2, W_3\}$ . Notice that all edges connect nodes from two different sets.

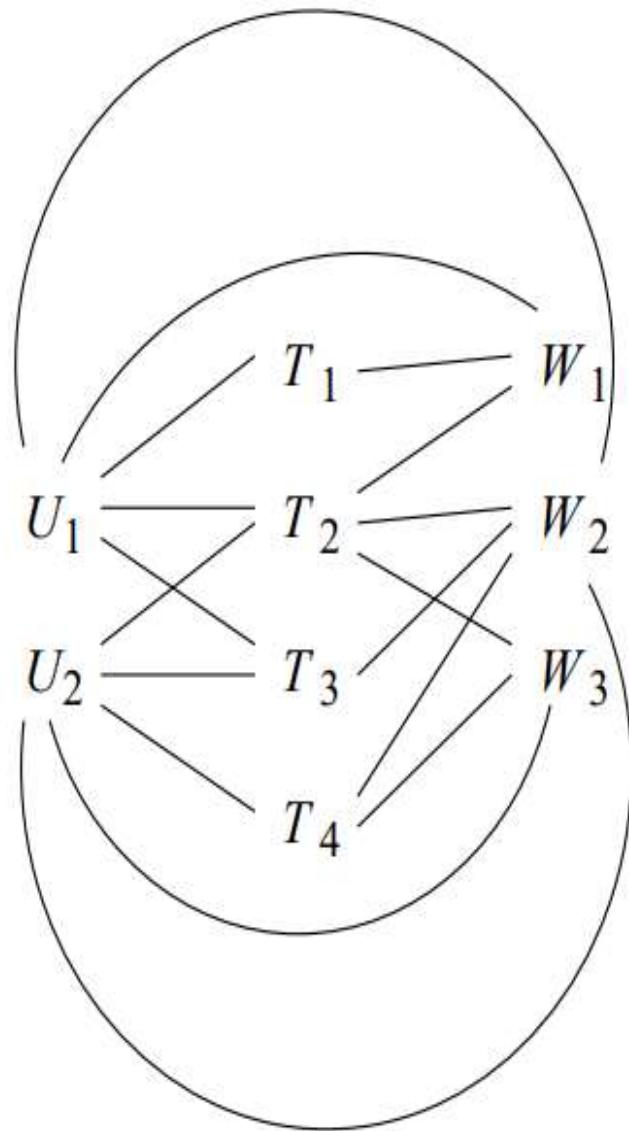


Figure 10.2: A tripartite graph representing users, tags, and Web pages

We may assume this graph represents information about the three kinds of entities. For example, the edge  $(U1, T2)$  means that user  $U1$  has placed the tag  $T2$  on at least one page.

Note that the graph does not tell us a detail that could be important: who placed which tag on which page? To represent such ternary information would require a more complex representation, such as a database relation with three columns corresponding to users, tags, and pages.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC: Clustering of Social Network Graphs

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

## **Clustering of Social Network Graphs**

An important aspect of social networks is that they contain communities of entities that are connected by many edges. These typically correspond to groups of friends at school or groups of researchers interested in the same topic, for example.

In this section, we shall consider clustering of the graph as a way to identify communities.

### **(a) Distance Measures for Social-Network Graphs**

If we were to apply standard clustering techniques to a social-network graph, our first step would be to define a distance measure.

When the edges of the graph have labels, these labels might be usable as a distance measure, depending on what they represented.

But when the edges are unlabeled, as in a “friends” graph, there is not much we can do to define a suitable distance.

Our first instinct is to assume that nodes are close if they have an edge between them and distant if not. Thus, we could say that the distance  $d(x, y)$  is 0 if there is an edge  $(x, y)$  and 1 if there is no such edge.

We could use any other two values, such as 1 and  $\infty$ , as long as the distance is closer when there is an edge.

Neither of these two-valued “distance measures” – 0 and 1 or 1 and  $\infty$  – is a true distance measure. The reason is that they violate the triangle inequality when there are three nodes, with two edges between them.

That is, if there are edges (A, B) and (B, C), but no edge (A, C), then the distance from A to C exceeds the sum of the distances from A to B to C. We could fix this problem by using, say, distance 1 for an edge and distance 1.5 for a missing edge. But the problem with two-valued distance functions is not limited to the triangle inequality.

### **(b) Applying Standard Clustering Methods**

There are two general approaches to clustering: hierarchical (agglomerative) and point-assignment.

Hierarchical clustering of a social-network graph starts by combining some two nodes that are connected by an edge.

Successively, edges that are not between two nodes of the same cluster would be chosen randomly to combine the clusters to which their two nodes belong.

Drawbacks of Hierarchical clustering

---

## **Disadvantages**

---

Hard to define levels for clusters. Sensitivity to noise and outliers

---

Rigid, cannot correct later for erroneous decisions made earlier.

---

No ability to make corrections when the splitting/merging decision is taken.

---

Lack of interpretability in terms of the cluster descriptors.

---

It cannot perform well on a large database

Now, consider a point-assignment approach to clustering social networks. Again, the fact that all edges are at the same distance will introduce a number of random factors that will lead to some nodes being assigned to the wrong cluster.

So, both the standard clustering methods are not suitable for clustering social network graphs.

**(c) Betweenness** Since there are problems with standard clustering methods, several specialized clustering techniques have been developed to find communities in social networks.

In this section we shall consider one of the simplest, based on finding the edges that are least likely to be inside a community.

Define the betweenness of an edge  $(a, b)$  to be the number of pairs of nodes  $x$  and  $y$  such that the edge  $(a, b)$  lies on the shortest path between  $x$  and  $y$ .

To be more precise, since there can be several shortest paths between  $x$  and  $y$ , edge  $(a, b)$  is credited with the fraction of those shortest paths that include the edge  $(a, b)$ .

As in golf, a high score is bad. It suggests that the edge  $(a, b)$  runs between two different communities; that is,  $a$  and  $b$  do not belong to the same community.

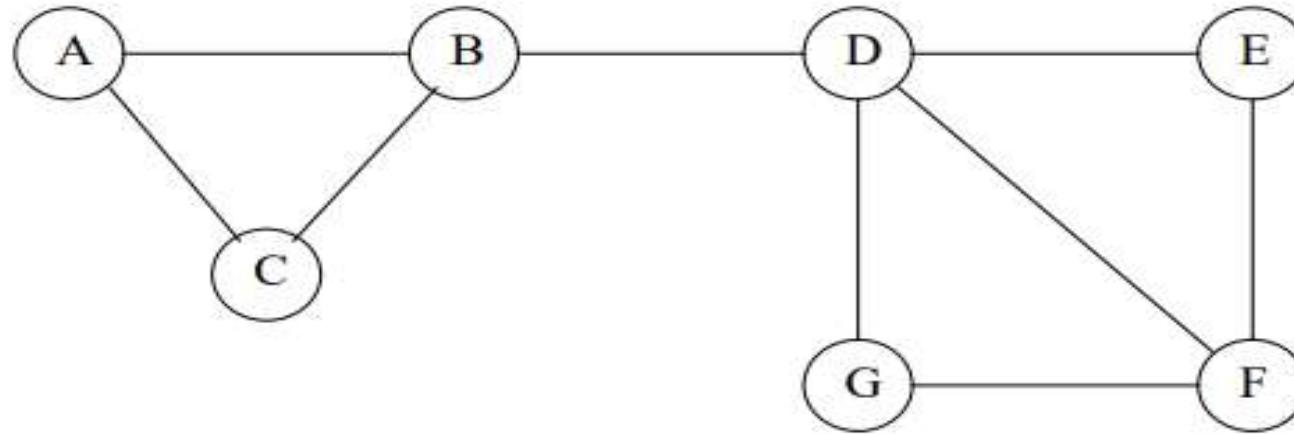


Figure 10.3: Repeat of Fig. 10.1

In Fig. 10.3 the edge (B, D) has the highest betweenness, as should surprise no one. In fact, this edge is on every shortest path between any of A, B, and C to any of D, E, F, and G.

Its betweenness is therefore  $3 \times 4 = 12$ . In contrast, the edge (D, F) is on only four shortest paths: those from A, B, C, and D to F.

In Fig. 10.3 the edge (B, D) has the highest betweenness, as should surprise no one. In fact, this edge is on every shortest path between any of A, B, and C to any of D, E, F, and G.

Its betweenness is therefore  $3 \times 4 = 12$ . In contrast, the edge (D, F) is on only four shortest paths: those from A, B, C, and D to F.

#### **(d)The Girvan-Newman Algorithm**

In order to exploit the betweenness of edges, we need to calculate the number of shortest paths going through each edge.

We shall describe a method called the Girvan-Newman (GN) Algorithm, which visits each node X once and computes the number of shortest paths from X to each of the other nodes that go through each of the edges.

The algorithm begins by performing a breadth-first search (BFS) of the graph, starting at the node X.

Note that the level of each node in the BFS presentation is the length of the shortest path from X to that node. Thus, the edges that go between nodes at the same level can never be part of a shortest path from X.

Edges between levels are called DAG edges (“DAG” stands for directed, acyclic graph). Each DAG edge will be part of at least one shortest path from root X.

If there is a DAG edge (Y, Z), where Y is at the level above Z (i.e., closer to the root), then we shall call Y a parent of Z and Z a child of Y , although parents are not necessarily unique in a DAG as they would be in a tree.

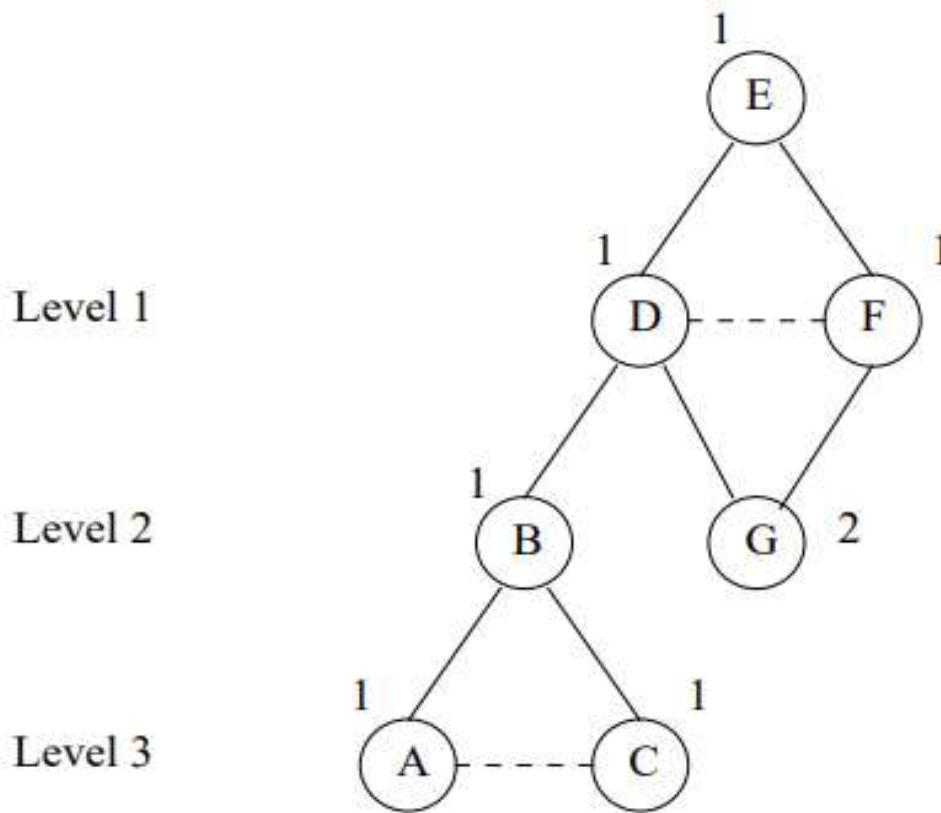


Figure 10.4: Step 1 of the Girvan-Newman Algorithm

**Example 10.6:** Figure 10.4 is a breadth-first presentation of the graph of Fig. 10.3, starting at node  $E$ . Solid edges are DAG edges and dashed edges connect nodes at the same level.  $\square$

The second step of the GN algorithm is to label each node by the number of shortest paths that reach it from the root. Start by labeling the root 1. Then, from the top down, label each node  $Y$  by the sum of the labels of its parents.

In Fig. 10.4 are the labels for each of the nodes. First, label the root E with 1. At level 1 are the nodes D and F. Each has only E as a parent, so they too are labeled 1. Nodes B and G are at level 2.

B has only D as a parent, so B's label is the same as the label of D, which is 1.

However, G has parents D and F, so its label is the sum of their labels, or 2.

Finally, at level 3, A and C each have only parent B, so their labels are the label of B, which is 1.

The third and final step is to calculate for each edge  $e$  the sum over all nodes  $Y$  of the fraction of shortest paths from the root  $X$  to  $Y$  that go through  $e$ .

This calculation involves computing this sum for both nodes and edges, from the bottom.

Each node other than the root is given a credit of 1, representing the shortest path to that node.

This credit may be divided among nodes and edges above, since there could be several different shortest paths to the node.

The rules for the calculation are as follows:

1. Each leaf in the DAG (a leaf is a node with no DAG edges to nodes at levels below) gets a credit of 1.
2. Each node that is not a leaf gets a credit equal to 1 plus the sum of the credits of the DAG edges from that node to the level below

3. A DAG edge  $e$  entering node  $Z$  from the level above is given a share of the credit of  $Z$  proportional to the fraction of shortest paths from the root to  $Z$  that go through  $e$ . Formally, let the parents of  $Z$  be  $Y_1, Y_2, \dots, Y_k$ . Let  $p_i$  be the number of shortest paths from the root to  $Y_i$ ; this number was computed in Step 2 and is illustrated by the labels in Fig. 10.4. Then the credit for the edge  $(Y_i, Z)$  is the credit of  $Z$  times  $p_i$  divided by  $\sum_{j=1}^k p_j$ .

After performing the credit calculation with each node as the root, we sum the credits for each edge. Then, since each shortest path will have been discovered twice – once when each of its endpoints is the root – we must divide the credit for each edge by 2.

**Example 10.8:** Let us perform the credit calculation for the BFS presentation of Fig. 10.4. We shall start from level 3 and proceed upwards. First,  $A$  and  $C$ , being leaves, get credit 1. Each of these nodes have only one parent, so their credit is given to the edges  $(B, A)$  and  $(B, C)$ , respectively.

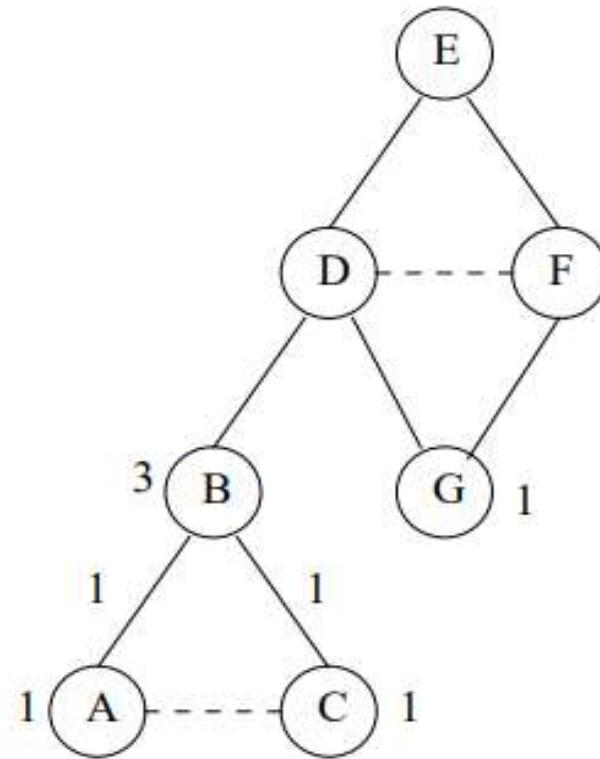


Figure 10.5: Final step of the Girvan-Newman Algorithm – levels 3 and 2

At level 2,  $G$  is a leaf, so it gets credit 1.  $B$  is not a leaf, so it gets credit equal to 1 plus the credits on the DAG edges entering it from below. Since both these edges have credit 1, the credit of  $B$  is 3. Intuitively 3 represents the fact that all shortest paths from  $E$  to  $A$ ,  $B$ , and  $C$  go through  $B$ . Figure 10.5 shows the credits assigned so far.

Now, let us proceed to level 1. B has only one parent, D, so the edge (D, B) gets the entire credit of B, which is 3. However, G has two parents, D and F.

We therefore need to divide the credit of 1 that G has between the edges (D, G) and (F, G). In what proportion do we divide?

If you examine the labels of Fig. 10.4, you see that both D and F have label 1, representing the fact that there is one shortest path from E to each of these nodes.

Thus, we give half the credit of G to each of these edges; i.e., their credit is each  $1/(1 + 1) = 0.5$ . Had the labels of D and F in Fig. 10.4 been 5 and 3, meaning there were five shortest paths to D and only three to F, then the credit of edge (D, G) would have been  $5/8$  and the credit of edge (F, G) would have been  $3/8$ .

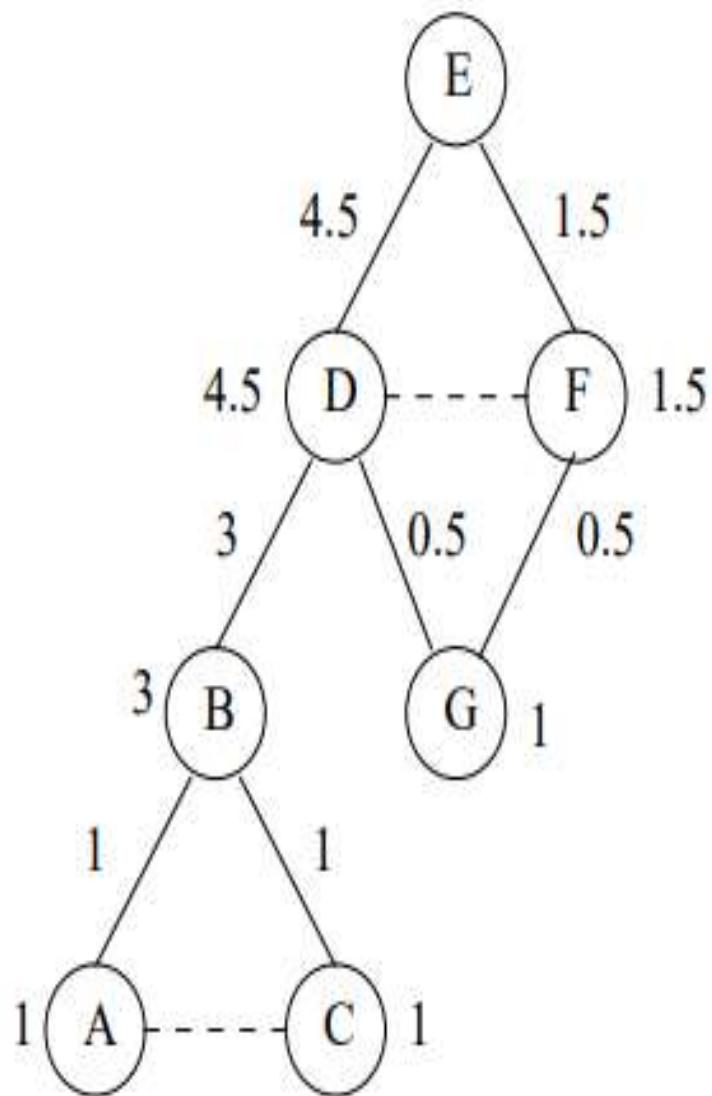


Figure 10.6: Final step of the Girvan-Newman Algorithm – completing the credit calculation

## **(e) Using Betweenness to Find Communities**

The betweenness scores for the edges of a graph behave something like a distance measure on the nodes of the graph. Let us start with our running example, the graph of Fig. 10.1. We see it with the betweenness for each edge in Fig. 10.7.

The calculation of the betweenness will be left to the reader. The only tricky part of the count is to observe that between E and G there are two shortest paths, one going through D and the other through F. Thus, each of the edges (D, E), (E, F), (D, G), and (G, F) are credited with half a shortest path.

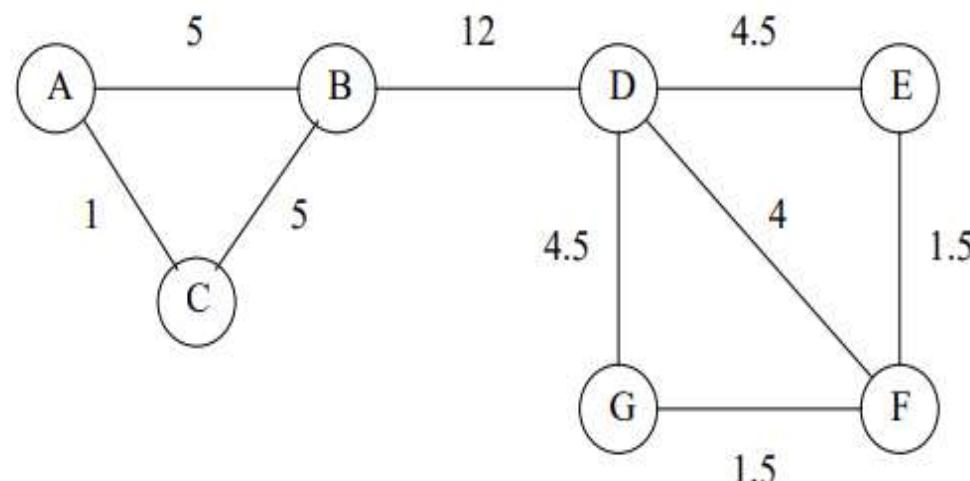


Figure 10.7: Betweenness scores for the graph of Fig. 10.1

Clearly, edge (B, D) has the highest betweenness, so it is removed first. That leaves us with exactly the communities we observed make the most sense, namely: {A, B, C} and {D, E, F, G}.

However, we can continue to remove edges. Next to leave are (A, B) and (B, C) with a score of 5, followed by (D, E) and (D, G) with a score of 4.5.

Then, (D, F), whose score is 4, would leave the graph. We see in Fig. 10.8 the graph that remains.

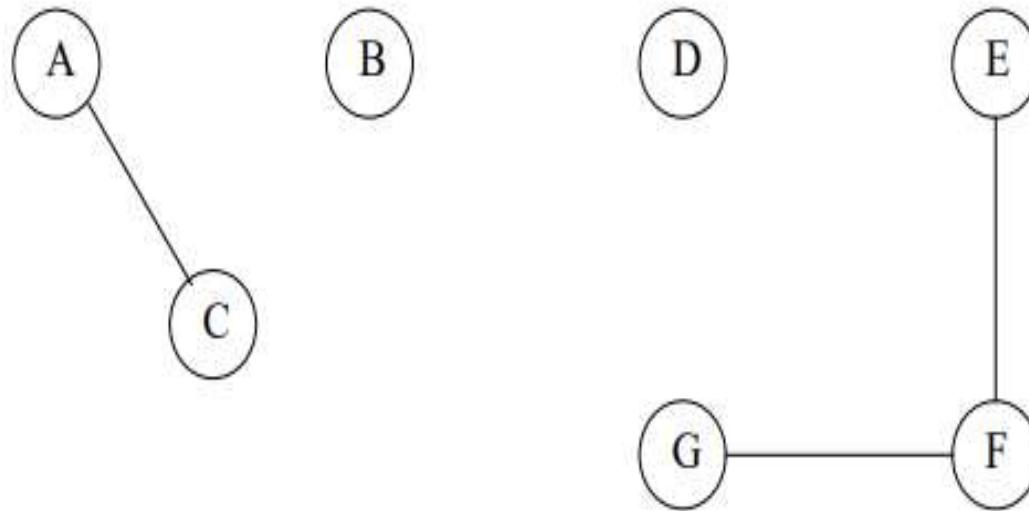


Figure 10.8: All the edges with betweenness 4 or more have been removed

The “communities” of Fig. 10.8 look strange.

One implication is that A and C are more closely knit to each other than to B. That is, in some sense B is a “traitor” to the community {A, B, C} because he has a friend D outside that community.

Likewise, D can be seen as a “traitor” to the group {D, E, F, G}, which is why in Fig. 10.8, only E, F, and G remain connected.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC: Direct Discovery of Communities

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

## **Direct Discovery of Communities**

In the previous topic , “clustering of social network graphs”, we searched for communities by partitioning all the individuals in a social network.

While this approach is relatively efficient, it does have several limitations. It is not possible to place an individual in two different communities, and everyone is assigned to a community.

In this section, we shall see a technique for discovering communities directly by looking for subsets of the nodes that have a relatively large number of edges among them.

Interestingly, the technique for doing this search on a large graph involves finding large frequent itemsets.

## (a) Finding Cliques

Our first thought about how we could find sets of nodes with many edges between them is to start by finding a large clique (a set of nodes with edges between any two of them).

However, that task is not easy. Not only is finding maximal cliques NP-complete, but it is among the hardest of the NP-complete problems in the sense that even approximating the maximal clique is hard.

Further, it is possible to have a set of nodes with almost all edges between them, and yet have only relatively small cliques.

Example : Suppose our graph has nodes numbered  $1, 2, \dots, n$  and there is an edge between two nodes  $i$  and  $j$  unless  $i$  and  $j$  have the same remainder when divided by  $k$ .

Then the fraction of possible edges that are actually present is approximately  $(k - 1)/k$ .

There are many cliques of size  $k$ , of which  $\{1, 2, \dots, k\}$  is but one example. Yet there are no cliques larger than  $k$ .

To see why, observe that any set of  $k + 1$  nodes has two that leave the same remainder when divided by  $k$ . This point is an application of the “pigeonhole principle.”

Since there are only  $k$  different remainders possible, we cannot have distinct remainders for each of  $k + 1$  nodes.

Thus, no set of  $k + 1$  nodes can be a clique in this graph.

## **(b)Complete Bipartite Graphs**

A complete bipartite graph consists of  $s$  nodes on one side and  $t$  nodes on the other side, with all  $st$  possible edges between the nodes of one side and the other present.

We denote this graph by  $K_{s,t}$ . You should draw an analogy between complete bipartite graphs as subgraphs of general bipartite graphs and cliques as subgraphs of general graphs.

In fact, a clique of  $s$  nodes is often referred to as a complete graph and denoted  $K_s$ , while a complete bipartite subgraph is sometimes called a bi-clique.

We know it is not possible to guarantee that a graph with many edges necessarily has a large clique, it is possible to guarantee that a bipartite graph with many edges has a large complete bipartite subgraph.

We can regard a complete bipartite subgraph (or a clique if we discovered a large one) as the nucleus of a community and add to it nodes with many edges to existing members of the community.

If the graph itself is  $k$ -partite, then we can take nodes of two types and the edges between them to form a bipartite graph.

In this bipartite graph, we can search for complete bipartite subgraphs as the nuclei of communities.

For instance, we could focus on the tag and page nodes of a tripartite graph (Users-Tags-Web pages eg. in Fig. 10.2) and try to find communities of tags and Web pages.

Such a community would consist of related tags and related pages that deserved many or all of those tags

However, we can also use complete bipartite subgraphs for community finding in ordinary graphs where nodes all have the same type. Divide the nodes into two equal groups at random.

If a community exists, then we would expect about half its nodes to fall into each group, and we would expect that about half its edges would go between groups.

Thus, we still have a reasonable chance of identifying a large complete bipartite subgraph in the community.

To this nucleus we can add nodes from either of the two groups, if they have edges to many of the nodes already identified as belonging to the community

## (c) Finding Complete Bipartite Subgraphs

Suppose we are given a large bipartite graph  $G$ , and we want to find instances of  $K_{s,t}$  within it.

It is possible to view the problem of finding instances of  $K_{s,t}$  within  $G$  as one of finding frequent itemsets.

For this purpose, let the “items” be the nodes on one side of  $G$ , which we shall call the left side. We assume that the instance of  $K_{s,t}$  we are looking for has  $t$  nodes on the left side, and we shall also assume for efficiency that  $t \leq s$ .

The “baskets” correspond to the nodes on the other side of  $G$  (the right side). The members of the basket for node  $v$  are the nodes of the left side to which  $v$  is connected.

Finally, let the support threshold be  $s$ , the number of nodes that the instance of  $K_{s,t}$  has on the right side.

We can now state the problem of finding instances of  $K_{s,t}$  as that of finding frequent itemsets  $F$  of size  $t$ .

That is, if a set of  $t$  nodes on the left side is frequent, then they all occur together in at least  $s$  baskets.

But the baskets are the nodes on the right side. Each basket corresponds to a node that is connected to all  $t$  of the nodes in  $F$ .

Thus, the frequent itemset of size  $t$  and  $s$  of the baskets in which all those items appear form an instance of  $K_{s,t}$ .

## Example :

Recall the bipartite graph of Fig. 10.10.

The left side is the nodes  $\{1, 2, 3, 4\}$  and the right side is  $\{a, b, c, d\}$ .

The latter are the baskets, so basket  $a$  consists of “items” 1 and 4; that is,  $a = \{1, 4\}$ .

Similarly,  $b = \{2, 3\}$ ,  $c = \{1\}$  and  $d = \{3\}$

If  $s = 2$  and  $t = 1$ , we must find itemsets of size 1 that appear in at least two baskets.  $\{1\}$  is one such itemset, and  $\{3\}$  is another.

However, in this tiny example there are no itemsets for larger, more interesting values of  $s$  and  $t$ , such as  $s = t = 2$ .

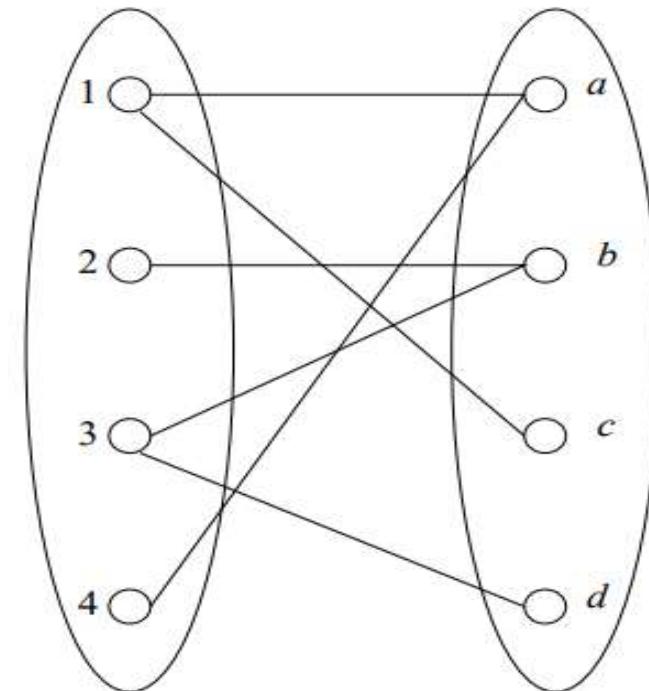


Figure 10.10: The bipartite graph from Fig. 8.1

## (d) Why Complete Bipartite Graphs Must Exist

We must now turn to the matter of demonstrating that any bipartite graph with a sufficiently high fraction of the edges present will have an instance of  $K_{s,t}$ .

In what follows, assume that the graph  $G$  has  $n$  nodes on the left and another  $n$  nodes on the right.

Assume the two sides have the same number of nodes simplifies the calculation, but the argument generalizes to sides of any size.

Finally, let  $d$  be the average degree of all nodes.

The argument involves counting the number of frequent itemsets of size  $t$  that a basket with  $d$  items contributes to.

When we sum this number over all nodes on the right side, we get the total frequency of all the subsets of size  $t$  on the left.

When we divide by  $\binom{n}{t}$ , we get the average frequency of all itemsets of size  $t$ . At least one must have a frequency that is at least average, so if this average is at least  $s$ , we know an instance of  $K_{s,t}$  exists.

Now, we provide the detailed calculation. Suppose the degree of the  $i$ th node on the right is  $d_i$ ; that is,  $d_i$  is the size of the  $i$ th basket. Then this basket contributes to  $\binom{d_i}{t}$  itemsets of size  $t$ . The total contribution of the  $n$  nodes on the right is  $\sum \binom{d_i}{t}$ . The value of this sum depends on the  $d_i$ 's, of course. However, we know that the average value of  $d_i$  is  $d$ .

Thus, in what follows, we shall assume that all nodes have the average degree  $d$ . So doing minimizes the total contribution to the counts for the itemsets, and thus makes it least likely that there will be a frequent itemset (itemset with support  $s$  or more) of size  $t$ . Observe the following:

- The total contribution of the  $n$  nodes on the right to the counts of the itemsets of size  $t$  is  $n \binom{d}{t}$ .
- The number of itemsets of size  $t$  is  $\binom{n}{t}$ .
- Thus, the average count of an itemset of size  $t$  is  $n \binom{d}{t} / \binom{n}{t}$ ; this expression must be at least  $s$  if we are to argue that an instance of  $K_{s,t}$  exists.

If we expand the binomial coefficients in terms of factorials, we find

$$n \binom{d}{t} / \binom{n}{t} = n d! (n - t)! t! / ((d - t)! t! n!) =$$

$$n(d)(d - 1) \cdots (d - t + 1) / (n(n - 1) \cdots (n - t + 1))$$

To simplify the formula above, let us assume that  $n$  is much larger than  $d$ , and  $d$  is much larger than  $t$ . Then  $d(d - 1) \cdots (d - t + 1)$  is approximately  $d^t$ , and  $n(n - 1) \cdots (n - t + 1)$  is approximately  $n^t$ . We thus require that

$$n(d/n)^t \geq s$$

**Example 10.13:** Suppose there is a community with 100 nodes on each side, and the average degree of nodes is 50; i.e., half the possible edges exist. This community will have an instance of  $K_{s,t}$ , provided  $100(1/2)^t \geq s$ . For example, if  $t = 2$ , then  $s$  can be as large as 25. If  $t = 3$ ,  $s$  can be 11, and if  $t = 4$ ,  $s$  can be 6.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC: Partitioning of Graphs

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

## Partitioning of Graphs

We examine another approach to organizing social-network graphs.

We use some important tools from matrix theory (“spectral methods”) to formulate the problem of partitioning a graph to minimize the number of edges that connect different components.

The goal of minimizing the “cut” size needs to be understood carefully before proceeding.

For instance, if you just joined Facebook, you are not yet connected to any friends.

## Partitioning of Graphs

We do not want to partition the friends graph with you in one group and the rest of the world in the other group, even though that would partition the graph without there being any edges that connect members of the two groups.

This cut is not desirable because the two components are too unequal in size.

**10.4.1 What Makes a Good Partition?** Given a graph, we would like to divide the nodes into two sets so that the cut, or set of edges that connect nodes in different sets is minimized.

However, we also want to constrain the selection of the cut so that the two sets are approximately equal in size. The next example illustrates the point.

**Example 10.14:** Recall our running example of the graph in Fig. 10.1. There, it is evident that the best partition puts  $\{A, B, C\}$  in one set and  $\{D, E, F, G\}$  in the other. The cut consists only of the edge  $(B, D)$  and is of size 1. No nontrivial cut can be smaller.

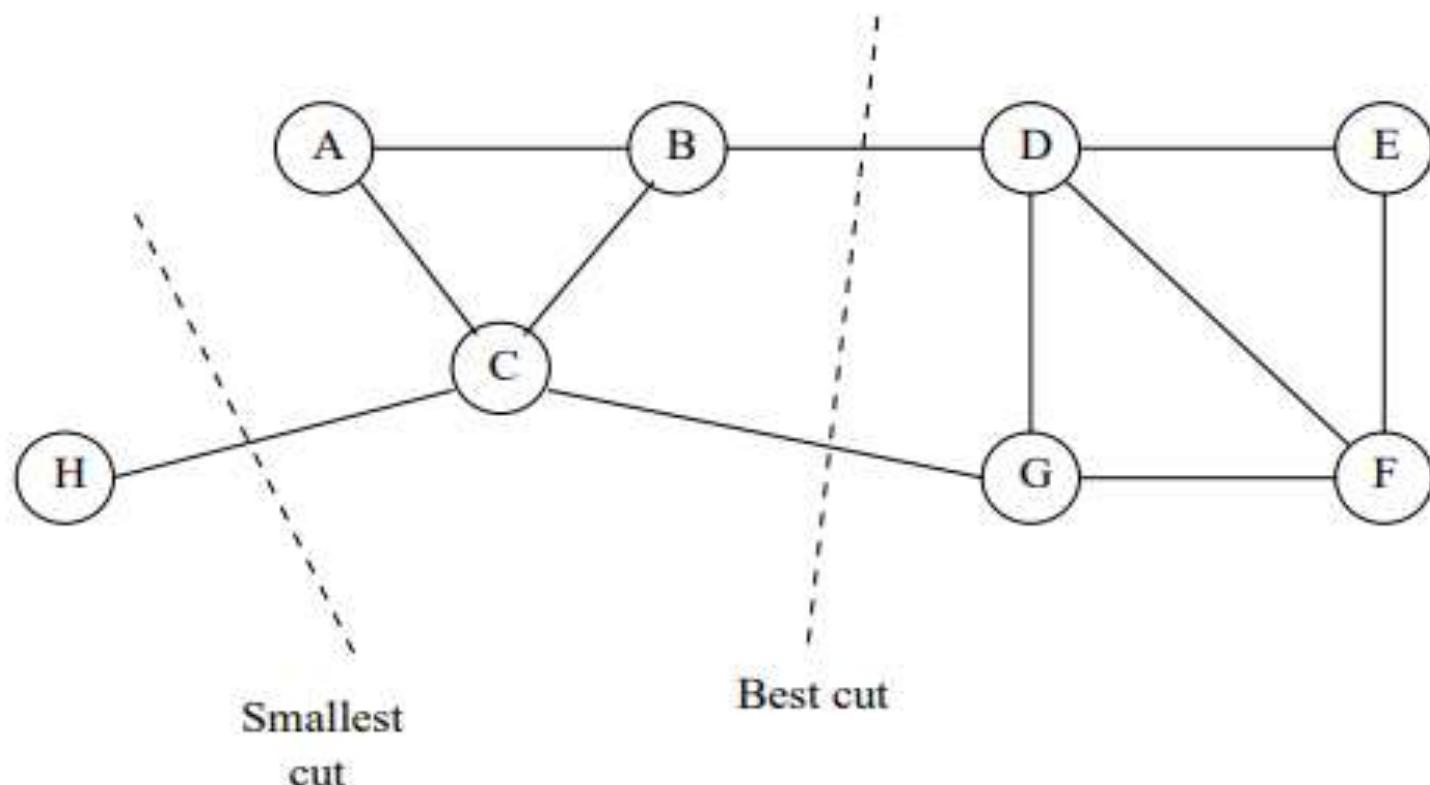


Figure 10.11: The smallest cut might not be the best cut

In Fig. 10.11 is a variant of our example, where we have added the node  $H$  and two extra edges,  $(H, C)$  and  $(C, G)$ . If all we wanted was to minimize the size of the cut, then the best choice would be to put  $H$  in one set and all the other nodes in the other set. But it should be apparent that if we reject partitions where one set is too small, then the best we can do is to use the cut consisting of edges  $(B, D)$  and  $(C, G)$ , which partitions the graph into two equal-sized sets  $\{A, B, C, H\}$  and  $\{D, E, F, G\}$ .  $\square$

## Normalized Cuts

A proper definition of a “good” cut must balance the size of the cut itself against the difference in the sizes of the sets that the cut creates. One choice that serves well is the “normalized cut.”

First, define the volume of a set  $S$  of nodes, denoted  $\text{Vol}(S)$ , to be the number of edges with at least one end in  $S$ . Suppose we partition the nodes of a graph into two disjoint sets  $S$  and  $T$ .

Let  $\text{Cut}(S, T)$  be the number of edges that connect a node in  $S$  to a node in  $T$ . Then the normalized cut value for  $S$  and  $T$  is

$$\frac{\text{Cut}(S, T)}{\text{Vol}(S)} + \frac{\text{Cut}(S, T)}{\text{Vol}(T)}$$

Example : Again consider the graph of Fig. 10.11. If we choose

$S = \{H\}$  and

$T = \{A, B, C, D, E, F, G\}$ , then  $\text{Cut}(S, T) = 1$ .

$\text{Vol}(S) = 1$ , because there is only one edge connected to H.

$\text{Vol}(T) = 11$ , because all the edges have at least one end at a node of T .

Thus, the normalized cut for this partition is  $1/1 + 1/11 = 1.09$ .

Now, consider the preferred cut for this graph consisting of the edges (B, D) and (C, G).

Then  $S = \{A, B, C, H\}$  and

$T = \{D, E, F, G\}$ .

$\text{Cut}(S, T) = 2$ ,  $\text{Vol}(S) = 6$ , and  $\text{Vol}(T) = 7$ .

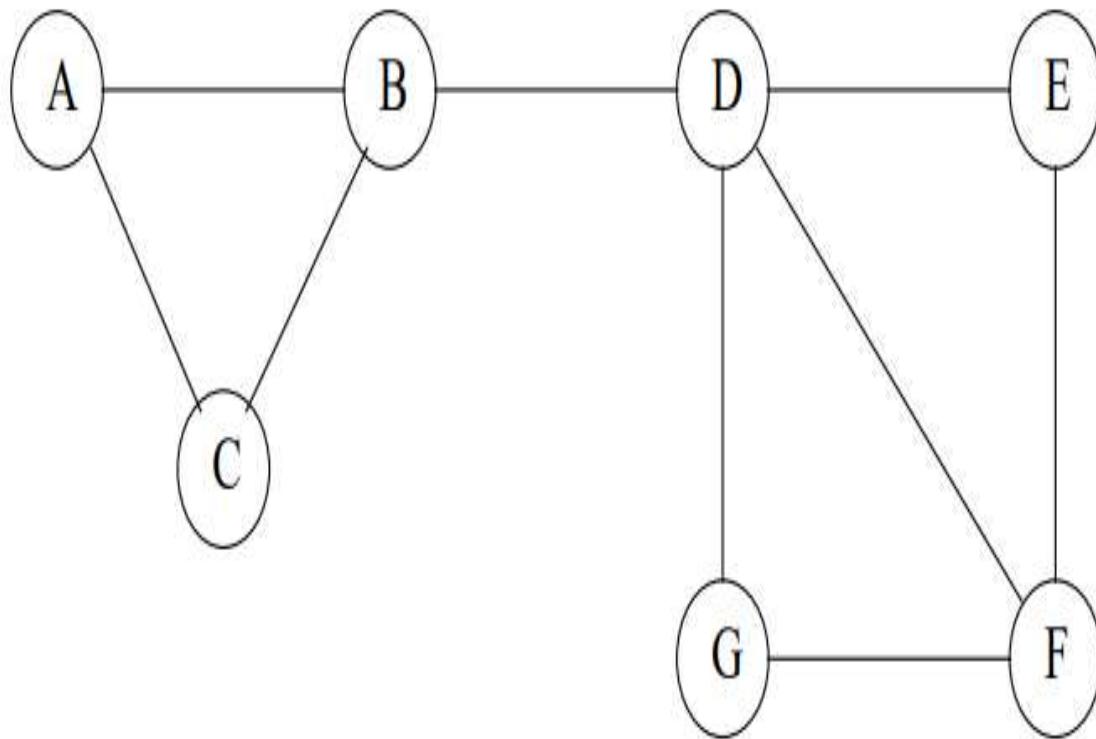
The normalized cut for this partition is thus only  $2/6 + 2/7 = 0.62$ .

## **Some Matrices That Describe Graphs**

To develop the theory of how matrix algebra can help us find good graph partitions,

We first need to learn about three different matrices that describe aspects of a graph.

The first should be familiar: **the adjacency matrix that has a 1 in row i and column j if there is an edge between nodes i and j, and 0 otherwise.**



0	1	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
0	1	0	0	1	1	1
0	0	0	1	0	1	0
0	0	0	1	1	0	1
0	0	0	1	0	1	0

Adjacency Matrix for the graph

Figure 10.12: Repeat of the graph of Fig. 10.1

Note that the rows and columns correspond to the nodes A, B, . . . , G in that order.

For example, **the edge (B, D) is reflected by the fact that the entry in row 2 and column 4 is 1 and so is the entry in row 4 and column 2**

The second matrix we need is the degree matrix for a graph. This graph has nonzero entries only on the diagonal. The entry for row and column  $i$  is the degree of the  $i$ th node.

The degree matrix for the graph is shown below. For instance, the entry in row 4 and column 4 is 4 because node D has edges to four other nodes. The entry in row 4 and column 5 is 0, because that entry is not on the diagonal.

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

Degree matrix for the preceding graph

Suppose our graph has adjacency matrix A and degree matrix D. Our third matrix, called the Laplacian matrix, is  $L = D - A$ , the difference between the degree matrix and the adjacency matrix.

That is, the Laplacian matrix L has the same entries as D on the diagonal. Off the diagonal, at row i and column j, L has  $-1$  if there is an edge between nodes i and j and 0 if not.

The Laplacian matrix for the graph is shown below. Notice that each row and each column sums to zero, as must be the case for any Laplacian matrix.

$$\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 4 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 3 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix}$$

Laplacian matrix for the preceding graph

## Eigenvalues of the Laplacian Matrix

We can get a good idea of the best way to partition a graph from the eigenvalues and eigenvectors of its Laplacian matrix.

The smallest eigenvalue for every Laplacian matrix is 0, and its corresponding eigenvector is  $[1, 1, \dots, 1]$ .

To see why, let  $L$  be the Laplacian matrix for a graph of  $n$  nodes, and let  $1$  be the column vector of all 1's with length  $n$ .

We claim  $L1$  is a column vector of all 0's. There is a simple way to find the second-smallest eigenvalue for any matrix, such as the Laplacian matrix, that is symmetric (the entry in row  $i$  and column  $j$  equals the entry in row  $j$  and column  $i$ ).

There is a simple way to find the second-smallest eigenvalue for any matrix, such as the Laplacian matrix, that is *symmetric* (the entry in row  $i$  and column  $j$  equals the entry in row  $j$  and column  $i$ ). While we shall not prove this fact, the second-smallest eigenvalue of  $L$  is the minimum of  $\mathbf{x}^T L \mathbf{x}$ , where  $\mathbf{x} = [x_1, x_2, \dots, x_n]$  is a column vector with  $n$  components, and the minimum is taken under the constraints:

1. The length of  $\mathbf{x}$  is 1; that is  $\sum_{i=1}^n x_i^2 = 1$ .
2.  $\mathbf{x}$  is orthogonal to the eigenvector associated with the smallest eigenvalue.

Moreover, the value of  $\mathbf{x}$  that achieves this minimum is the second eigenvector.

When  $L$  is a Laplacian matrix for an  $n$ -node graph, we know something more. The eigenvector associated with the smallest eigenvalue is  $\mathbf{1}$ . Thus, if  $\mathbf{x}$  is orthogonal to  $\mathbf{1}$ , we must have

$$\mathbf{x}^T \mathbf{1} = \sum_{i=1}^n x_i = 0$$

In addition for the Laplacian matrix, the expression  $\mathbf{x}^T L \mathbf{x}$  has a useful equivalent expression. Recall that  $L = D - A$ , where  $D$  and  $A$  are the degree and adjacency matrices of the same graph. Thus,  $\mathbf{x}^T L \mathbf{x} = \mathbf{x}^T D \mathbf{x} - \mathbf{x}^T A \mathbf{x}$ .

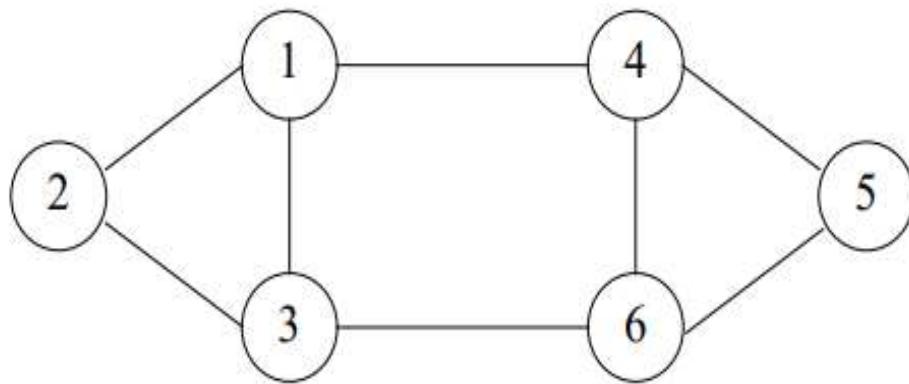


Figure 10.16: Graph for illustrating partitioning by spectral analysis

**Example 10.19:** Let us apply the above technique to the graph of Fig. 10.16. The Laplacian matrix for this graph is shown in Fig. 10.17. By standard methods or math packages we can find all the eigenvalues and eigenvectors of this matrix. We shall simply tabulate them in Fig. 10.18, from lowest eigenvalue to

highest. Note that we have not scaled the eigenvectors to have length 1, but could do so easily if we wished.

$$\begin{bmatrix} 3 & -1 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 3 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & -1 & -1 & 3 \end{bmatrix}$$

Figure 10.17: The Laplacian matrix for Fig. 10.16

The second eigenvector has three positive and three negative components. It makes the unsurprising suggestion that one group should be  $\{1, 2, 3\}$ , the nodes with positive components, and the other group should be  $\{4, 5, 6\}$ .  $\square$

Eigenvalue	0	1	3	3	4	5
Eigenvector	1	1	-5	-1	-1	-1
	1	2	4	-2	1	0
	1	1	1	3	-1	1
	1	-1	-5	-1	1	1
	1	-2	4	-2	-1	0
	1	-1	1	3	1	-1

Figure 10.18: Eigenvalues and eigenvectors for the matrix of Fig. 10.17

## 10.4.5 Alternative Partitioning Methods

The method we saw just before gives us a good partition of the graph into two pieces that have a small cut between them.

There are several ways we can use the same eigenvectors to suggest other good choices of partition.

First, we are not constrained to put all the nodes with positive components in the eigenvector into one group and those with negative components in the other.

We could set the threshold at some point other than zero. For instance, suppose we modified the above example so that the threshold was not zero, but  $-1.5$ .

Then the two nodes 4 and 6, with components  $-1$  in the second eigenvector of Fig. 10.18, would join 1, 2, and 3, leaving five nodes in one component and only node 5 in the other.

That partition would have a cut of size two, as did the choice based on the threshold of zero, but the two components have radically different sizes, so we would tend to prefer our original choice.

The third eigenvector, with a threshold of 0, puts nodes 1 and 4 in one group and the other four nodes in the other. That is not a bad partition, but its cut size is four, compared with the cut of size two that we get from the second eigenvector.

If we use both the second and third eigenvectors, we put nodes 2 and 3 in one group, because their components are positive in both eigenvectors.

Nodes 5 and 6 are in another group, because their components are negative in the second eigenvector and positive in the third.

Node 1 is in a group by itself because it is positive in the second eigenvector and negative in the third, while node 4 is also in a group by itself because its component is negative in both eigenvectors.

This partition of a six-node graph into four groups is too fine a partition to be meaningful.

But at least the groups of size two each have an edge between the nodes, so it is as good as we could ever get for a partition into groups of these sizes.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC: Finding Overlapping Communities

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

# Finding Overlapping Communities

We have concentrated on clustering a social graph to find communities. But communities are in practice rarely disjoint.

In this section, we explain a method for taking a social graph and fitting a model to it that best explains how the probability that two individuals are connected by an edge (are “friends”) increases as they become members of more communities in common.

An important tool in this analysis is “maximum-likelihood estimation”.

## The Nature of Communities

To begin, let us consider what we would expect two overlapping communities to look like.

Our data is a social graph, where nodes are people and there is an edge between two nodes if the people are “friends.”

Let us imagine that this graph represents students at a school, and there are two clubs in this school:  
the Chess Club and the Spanish Club.

It is reasonable to suppose that each of these clubs forms a community, as does any other club at the school.

It is also reasonable to suppose that two people in the Chess Club are more likely to be friends in the graph because they know each other from the club.

Likewise, if two people are in the Spanish Club, then there is a good chance they know each other, and are likely to be friends.

What if two people are in both clubs? They now have two reasons why they might know each other, and so we would expect an even greater probability that they will be friends in the social graph.

Our conclusion is that we expect edges to be dense within any community, but we expect edges to be even denser in the intersection of two communities, denser than that in the intersection of three communities, and so on. The idea is suggested by Figure.

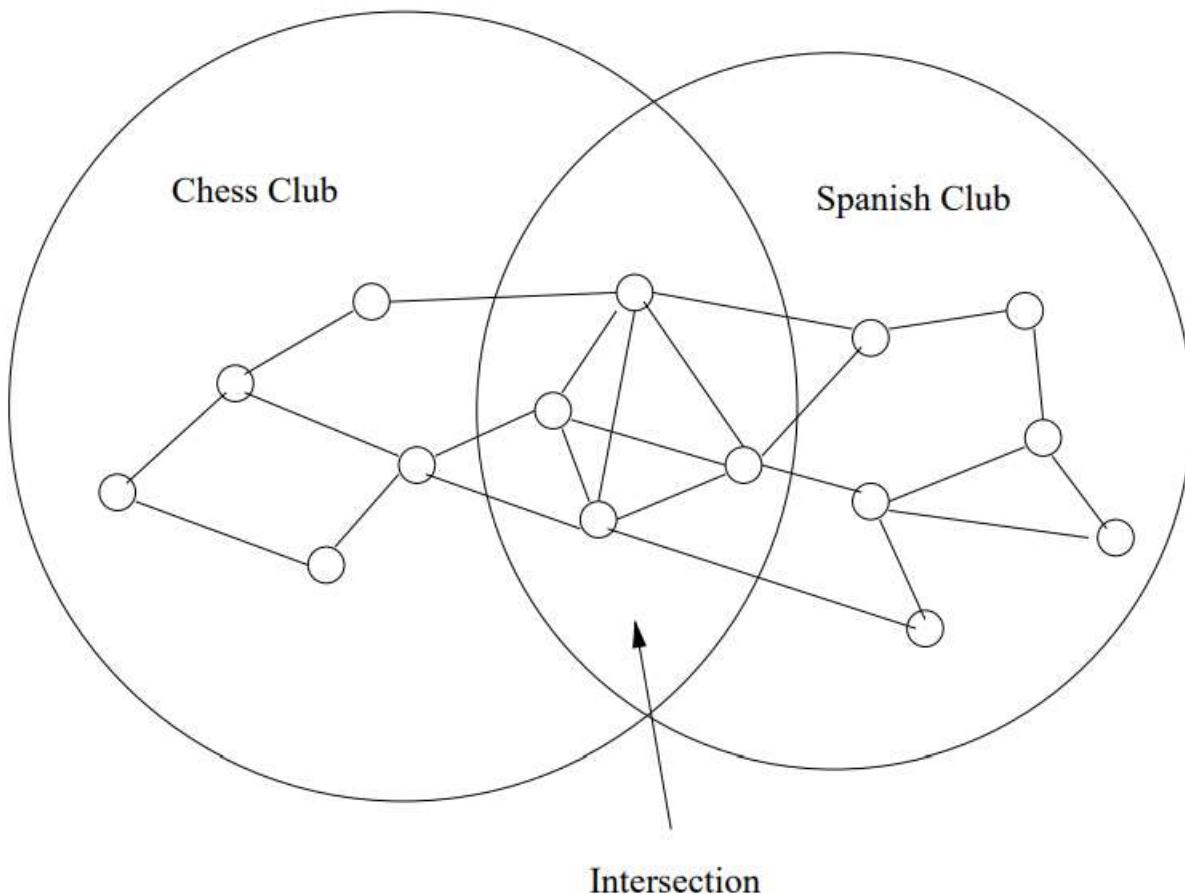


Figure 10.19: The overlap of two communities is denser than the nonoverlapping parts of these communities

## Maximum-Likelihood Estimation

Before we see the algorithm for finding communities that have overlap, let us learn a useful modeling tool called maximum-likelihood estimation, or MLE.

The idea behind MLE is that we make an assumption about the generative process (the model) that creates instances of some artifact, for example, “friends graphs.”

The model has parameters that determine the probability of generating any particular instance of the artifact;

this probability is called the likelihood of those parameter values.

We assume that the value of the parameters that gives the largest value of the likelihood is the correct model for the observed artifact.

An example should make the MLE principle clear. For instance, we might wish to generate random graphs.

We suppose that each edge is present with probability  $p$  and not present with probability  $1-p$ , with the presence or absence of each edge chosen independently.

The only parameter we can adjust is  $p$ .

For each value of  $p$  there is a small but nonzero probability that the graph generated will be exactly the one we see.

Following the MLE principle, we shall declare that the true value of  $p$  is the one for which the probability of generating the observed graph is the highest.

**Example 10.21:** Consider the graph of Fig. 10.19. There are 15 nodes and 23 edges. As there are  $\binom{15}{2} = 105$  pairs of 15 nodes, we see that if each edge is chosen with probability  $p$ , then the probability (likelihood) of generating exactly the graph of Fig. 10.19 is given by the function  $p^{23}(1-p)^{82}$ . No matter what value  $p$  has between 0 and 1, that is an incredibly tiny number. But the function does have a maximum, which we can determine by taking its derivative and setting that to 0. That is:

$$23p^{22}(1-p)^{82} - 82p^{23}(1-p)^{81} = 0$$

We can group terms to rewrite the above as

$$p^{22}(1-p)^{81}(23(1-p) - 82p) = 0$$

The only way the right side can be 0 is if  $p$  is 0 or 1, or the last factor,

$$(23(1-p) - 82p)$$

is 0. When  $p$  is 0 or 1, the value of the likelihood function  $p^{23}(1-p)^{82}$  is minimized, not maximized, so it must be the last factor that is 0. That is, the

likelihood of generating the graph of Fig. 10.19 is maximized when

$$23 - 23p - 82p = 0$$

or  $p = 23/105$ .

That outcome is hardly a surprise. It says the most likely value for  $p$  is the observed fraction of possible edges that are present in the graph. However, when we use a more complicated mechanism to generate graphs or other artifacts, the value of the parameters that produce the observed artifact with maximum likelihood is far from obvious.  $\square$

## The Affiliation-Graph Model

We shall now introduce a reasonable mechanism, called the *affiliation-graph model*, to generate social graphs from communities. Once we see how the parameters of the model influence the likelihood of seeing a given graph, we can address how one would solve for the values of the parameters that give the maximum likelihood. The mechanism, called *community-affiliation graphs*.

1. There is a given number of communities, and there is a given number of individuals (nodes of the graph).
2. Each community can have any set of individuals as members. That is, the memberships in the communities are parameters of the model.
3. Each community  $C$  has a probability  $p_C$  associated with it, the probability that two members of community  $C$  are connected by an edge because they are both members of  $C$ . These probabilities are also parameters of the model.
4. If a pair of nodes is in two or more communities, then there is an edge between them if any of the communities of which both are members justifies that edge according to rule (3).

We must compute the likelihood that a given graph with the proper number of nodes is generated by this mechanism.

The key observation is how the edge probabilities are computed, given an assignment of individuals to communities and values of the  $p_C$ 's.

Consider an edge  $(u, v)$  between nodes  $u$  and  $v$ . Suppose  $u$  and  $v$  are members of communities  $C$  and  $D$ , but not any other communities.

Then the probability that there is no edge between  $u$  and  $v$  is the product of the probabilities that there is no edge due to community  $C$  and no edge due to community  $D$ .

That is, with probability  $(1 - p_C)(1 - p_D)$  there is no edge  $(u, v)$  in the graph, and of course the probability that there is such an edge is 1 minus that.

More generally, if  $u$  and  $v$  are members of a nonempty set of communities  $M$  and not any others, then  $p_{uv}$ , the probability of an edge between  $u$  and  $v$  is given by:

$$p_{uv} = 1 - \prod_{C \text{ in } M} (1 - p_C)$$

As an important special case, if  $u$  and  $v$  are not in any communities together, then we take  $p_{uv}$  to be  $\epsilon$ , some very tiny number.

If we know which nodes are in which communities, then we can compute the likelihood of the given graph for these edge probabilities.

$$\prod_{(u,v) \text{ in } E} p_{uv} \quad \prod_{(u,v) \text{ not in } E} (1 - p_{uv})$$

**Example 10.22:** Consider the tiny social graph in Fig. 10.20. Suppose there are two communities  $C$  and  $D$ , with associated probabilities  $p_C$  and  $p_D$ . Also, suppose that we have determined (or are using as a temporary hypothesis) that  $C = \{w, x, y\}$  and  $D = \{w, y, z\}$ . To begin, consider the pair of nodes  $w$  and  $x$

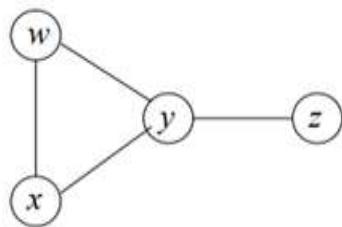


Figure 10.20: A social graph

□  $M_{wx} = \{C\}$ ; that is, this pair is in community  $C$  but not in community  $D$ . Therefore,  $p_{wx} = 1 - (1 - p_C) = p_C$ .

Similarly,  $x$  and  $y$  are only together in  $C$ ,  $y$  and  $z$  are only together in  $D$ , and likewise  $w$  and  $z$  are only together in  $D$ . Thus, we find  $p_{xy} = p_C$  and  $p_{yz} = p_{wz} = p_D$ . Now the pair  $w$  and  $y$  are together in both communities, so  $p_{wy} = 1 - (1 - p_C)(1 - p_D) = p_C + p_D - p_C p_D$ . Finally,  $x$  and  $z$  are not together in either community, so  $p_{xz} = \epsilon$ .

Now, we can compute the likelihood of the graph in Fig. 10.20, given our assumptions about membership in the two communities.

This likelihood is the product of the probabilities associated with each of the four pairs of nodes whose edges appear in the graph, times one minus the probability for each of the two pairs whose edges are not there.

That is we want

$$p_{wx}p_{wy}p_{xy}p_{yz}(1 - p_{wz})(1 - p_{xz})$$

Substituting the expressions we developed above for each of these probabilities, we get

$$(p_C)^2 p_D (p_C + p_D - p_C p_D) (1 - p_D) (1 - \epsilon)$$

Note that  $\epsilon$  is very small, so the last factor is essentially 1 and can be dropped.

We must find the values of  $p_C$  and  $p_D$  that maximize the above expression. First, notice that all factors are either independent of  $p_C$  or grow with  $p_C$ . The only hard step in this argument is to remember that  $p_D \leq 1$ , so

$$p_C + p_D - p_C p_D$$

must grow positively with  $p_C$ . It follows that the likelihood is maximized when  $p_C$  is as large as possible; that is,  $p_C = 1$ .

Next, we must find the value of  $p_D$  that maximizes the expression, given that  $p_C = 1$ . The expression becomes  $p_D(1 - p_D)$ , and it is easy to see that this expression has its maximum at  $p_D = 0.5$ . That is, given  $C = \{w, x, y\}$  and  $D = \{w, y, z\}$ , the maximum likelihood for the graph in Fig. 10.20 occurs when members of  $C$  are certain to have an edge between them and there is a 50% chance that joint membership in  $D$  will cause an edge between the members.

## Avoiding the Use of Discrete Membership Changes

Consider the situation where membership of individuals in communities is discrete; either you are a member of the community or not.

We can think of “strength of membership” of individuals in communities. Intuitively, the stronger the membership of two individuals in the same community, the more likely it is that this community will cause them to have an edge between them.

In this model, we can adjust the strength of membership for an individual in a community continuously, just as we can adjust the probability associated with a community in the affiliation graph model.

That improvement allows us to use standard methods, such as gradient descent, to maximize the expression for likelihood. In the improved model, we have

1. Fixed sets of communities and individuals, as before.
2. For each community  $C$  and individual  $x$ , there is a *strength of membership* parameter  $F_{xC}$ . These parameters can take any nonnegative value, and a value of 0 means the individual is definitely not in the community.
3. The probability that community  $C$  causes there to be an edge between nodes  $u$  and  $v$  is

$$p_C(u, v) = 1 - e^{-F_{uC} F_{vC}}$$

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC: Simrank

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

## Simrank

Simrank is another approach to analyzing social-network graphs.

This technique, called “simrank,” applies best to graphs with several types of nodes, although it can in principle be applied to any graph.

The purpose of simrank is to measure the similarity between nodes of the same type, and it does so by seeing where random walkers on the graph wind up when starting at a particular node.

Because calculation must be carried out once for each starting node, it is limited in the sizes of graphs that can be analyzed completely in this manner.

## **(a)Random Walkers on a Social Graph**

We can think of a person “walking” on a social network. The graph of a social network is generally undirected, while the Web graph is directed.

However, the difference is unimportant. A walker at a node N of an undirected graph will move with equal probability to any of the neighbors of N (those nodes with which N shares an edge).

Suppose, for example, that such a walker starts out at node T1 of Fig. shown in the next slide. At the first step, it would go either to U1 or W1. If to W1, then it would next either come back to T1 or go to T2.

If the walker first moved to U1, it would wind up at either T1, T2, or T3 next.

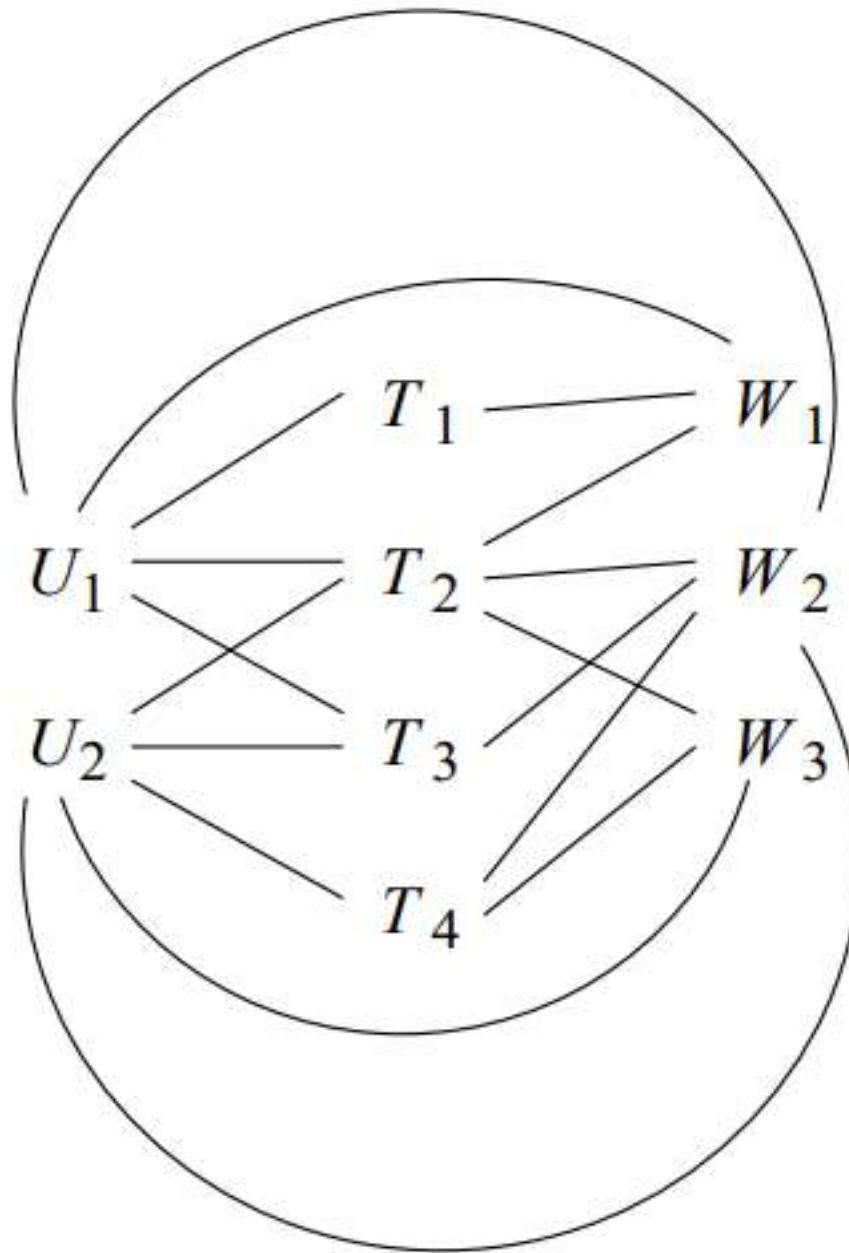


Figure 10.21: Repeat of Fig. 10.2

We conclude that, starting at T1, there is a good chance the walker would visit T2, at least initially, and that chance is better than the chance it would visit T3 or T4.

It would be interesting if we could infer that tags T1 and T2 are therefore related or similar in some way.

The evidence is that they have both been placed on a common Web page, W1, and they have also been used by a common tagger, U1.

However, if we allow the walker to continue traversing the graph at random, then the probability that the walker will be at any particular node does not depend on where it starts out.

## **(b)Random Walks with Restart**

We see from the observations above that it is not possible to measure similarity to a particular node by looking at the limiting distribution of the walker.

However, it is possible to introduce a small probability that the walker will stop walking at random.

We also studied that it is possible to select only a subset of Web pages as the teleport set, the pages that the walker would go to when they stopped surfing the Web at random.

Here, we take this idea to the extreme. As we are focused on one particular node N of a social network, and want to see where the random walker winds up on short walks from that node,

we modify the matrix of transition probabilities to have a small additional probability of transitioning to N from any node.

Formally, let  $M$  be the transition matrix of the graph  $G$ . That is, the entry in row  $i$  and column  $j$  of  $M$  is  $1/k$  if node  $j$  of  $G$  has degree  $k$ , and one of the adjacent nodes is  $i$ .

Otherwise, this entry is 0.

Figure below is an example of a very simple network involving three pictures, and two tags, “Sky” and “Tree” that have been placed on some of them.

Pictures 1 and 3 have both tags, while Picture 2 has only the tag “Sky.”

Intuitively, we expect that Picture 3 is more similar to Picture 1 than Picture 2 is, and an analysis using a random walker with restart at Picture 1 will support that intuition.

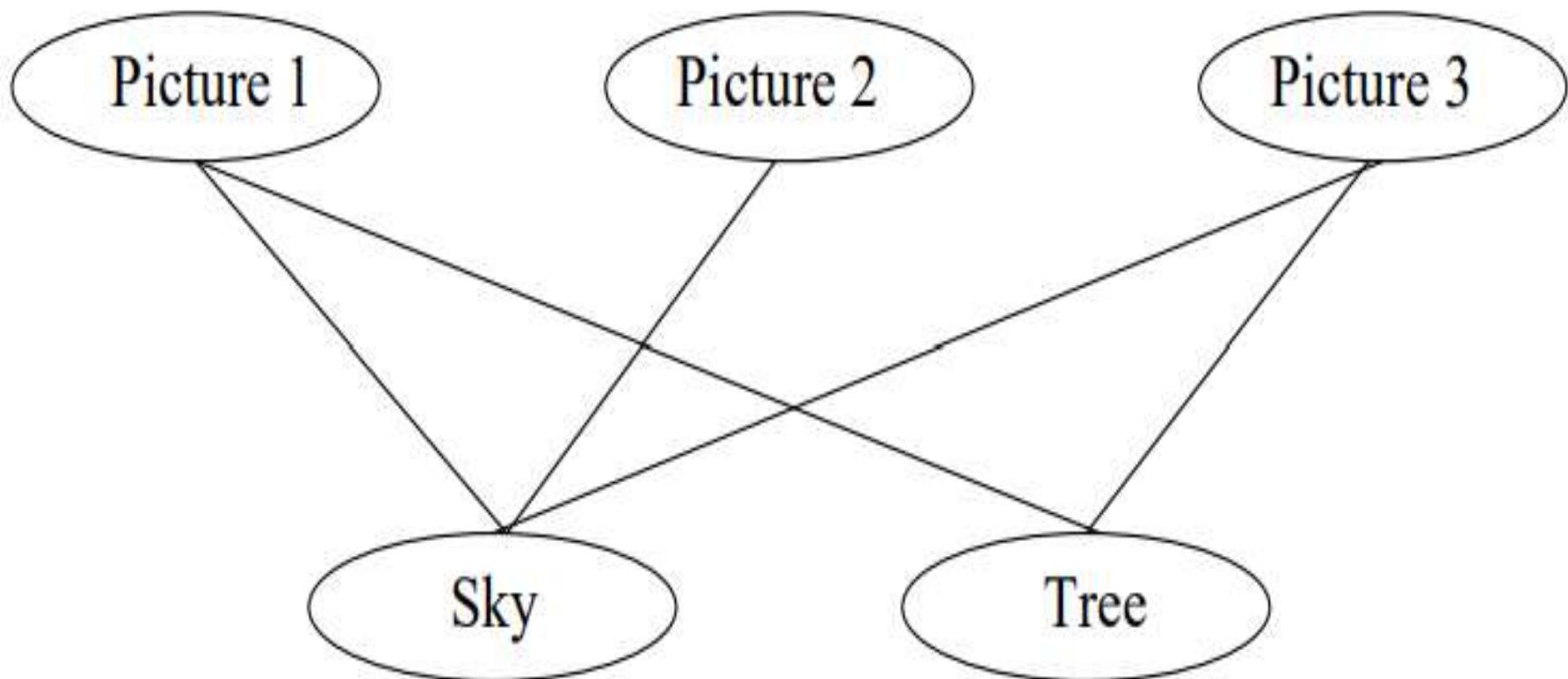


Figure 10.22: A simple bipartite social graph

Let us order the nodes as Picture 1, Picture 2, Picture 3, Sky, Tree. Then the transition matrix for the graph of Fig. 10.22 is

$$\begin{bmatrix} 0 & 0 & 0 & \frac{1}{3} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{2} & 1 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \end{bmatrix}$$

For example, the fourth column corresponds to the node “Sky,” and this node connects to each of the tree picture nodes.

It therefore has degree three, so the nonzero entries in its column must be  $1/3$ .

The picture nodes correspond to the first three rows and columns, so the entry  $1/3$  appears in the first three rows of column 4.

Since the “Sky” node does not have an edge to either itself or the “Tree” node, the entries in the last two rows of column 4 are 0.

As before, let us use  $\beta$  as the probability that the walker continues at random, so  $1 - \beta$  is the probability the walker will teleport to the initial node N.

Let  $e_N$  be the column vector that has 1 in the row for node N and 0's elsewhere.

Then if  $v$  is the column vector that reflects the probability the walker is at each of the nodes at a particular round, and  $v'$  is the probability the walker is at each of the nodes at the next round, then  $v'$  is related to  $v$  by:

$$v' = \beta M v + (1 - \beta) e_N$$

Assume  $M$  is the matrix &  
 $\beta = 0.8$ .

$$\begin{bmatrix} 0 & 0 & 0 & 1/3 & 1/2 \\ 0 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 1/3 & 1/2 \\ 1/2 & 1 & 1/2 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \end{bmatrix}$$

Also, assume that node  $N$  is for Picture 1; that is, we want to compute the similarity of other pictures to Picture 1. Then the equation for the new value

$\mathbf{v}'$  of the distribution that we must iterate is

$$\mathbf{v}' = \begin{bmatrix} 0 & 0 & 0 & 4/15 & 2/5 \\ 0 & 0 & 0 & 4/15 & 0 \\ 0 & 0 & 0 & 4/15 & 2/5 \\ 2/5 & 4/5 & 2/5 & 0 & 0 \\ 2/5 & 0 & 2/5 & 0 & 0 \end{bmatrix} \mathbf{v} + \begin{bmatrix} 1/5 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Since the graph of Fig. 10.22 is connected, the original matrix  $M$  is stochastic, and we can deduce that if the initial vector  $\mathbf{v}$  has components that sum to 1, then  $\mathbf{v}'$  will also have components that sum to 1. As a result, we can simplify the above equation by adding  $1/5$  to each of the entries in the first row of the matrix. That is, we can iterate the matrix-vector multiplication

$$\mathbf{v}' = \begin{bmatrix} 1/5 & 1/5 & 1/5 & 7/15 & 3/5 \\ 0 & 0 & 0 & 4/15 & 0 \\ 0 & 0 & 0 & 4/15 & 2/5 \\ 2/5 & 4/5 & 2/5 & 0 & 0 \\ 2/5 & 0 & 2/5 & 0 & 0 \end{bmatrix} \mathbf{v}$$

If we start with  $\mathbf{v} = \mathbf{e}_N$ , then the sequence of estimates of the distribution of the walker that we get is

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1/5 \\ 0 \\ 0 \\ 2/5 \\ 2/5 \end{bmatrix}, \begin{bmatrix} 35/75 \\ 8/75 \\ 20/75 \\ 6/75 \\ 6/75 \end{bmatrix}, \begin{bmatrix} 95/375 \\ 8/375 \\ 20/375 \\ 142/375 \\ 110/375 \end{bmatrix}, \begin{bmatrix} 2353/5625 \\ 568/5625 \\ 1228/5625 \\ 786/5625 \\ 690/5625 \end{bmatrix}, \dots, \begin{bmatrix} .345 \\ .066 \\ .145 \\ .249 \\ .196 \end{bmatrix}$$

We observe from the above that in the limit, the walker is more than twice as likely to be at Picture 3 than at Picture 2. This analysis confirms the intuition that Picture 3 is more like Picture 1 than Picture 2 is.  $\square$

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 3

TOPIC: Counting Triangles

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

# Counting Triangles

One of the most useful properties of social-network graphs is the count of triangles and other simple subgraphs.

We will look at the methods for estimating or getting an exact count of triangles in a very large graph.

We begin with a motivation for such counts and then give some methods for counting efficiently

## Why Count Triangles?

If we start with  $n$  nodes and add  $m$  edges to a graph at random, there will be an expected number of triangles in the graph.

We can calculate this number without too much difficulty. There are  $n C 3$  sets of three nodes, or approximately  $n^3/6$  sets of three nodes that might be a triangle.

The probability of an edge between any two given nodes being added is  $m/ nC2$ , or approximately  $(2m/n^2)^3$

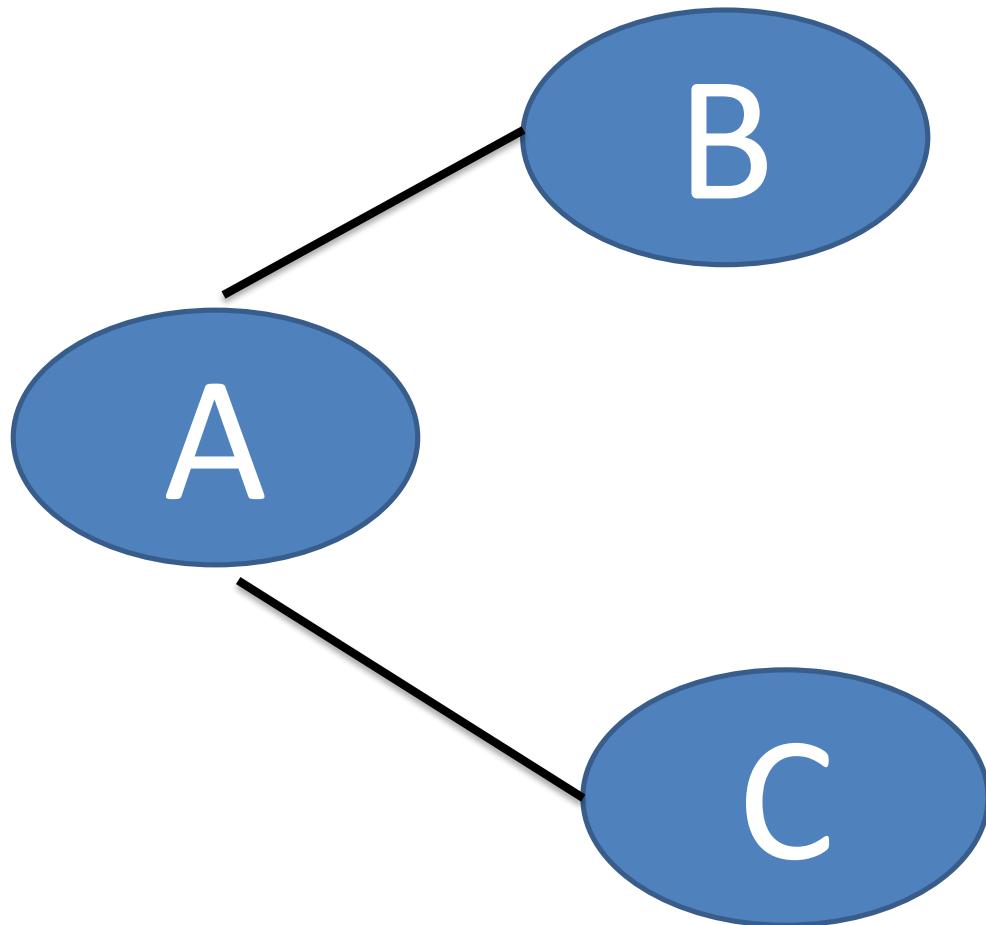
The probability that any set of three nodes has edges between each pair, if those edges are independently chosen to be present or absent is approximately  $(2m/n^2)^3 = 8m^3/n^6$ . Thus, the expected number of triangles in a graph of  $n$  nodes and  $m$  randomly selected edges is approximately  $(8m^3/n^6)(n^3/6) = \frac{4}{3}(m/n)^3$ .

If a graph is a social network with  $n$  participants and  $m$  pairs of “friends,” we would expect the number of triangles to be much greater than the value for a random graph.

The reason is that if A and B are friends, and A is also a friend of C, there should be a much greater chance than average that B and C are also friends.

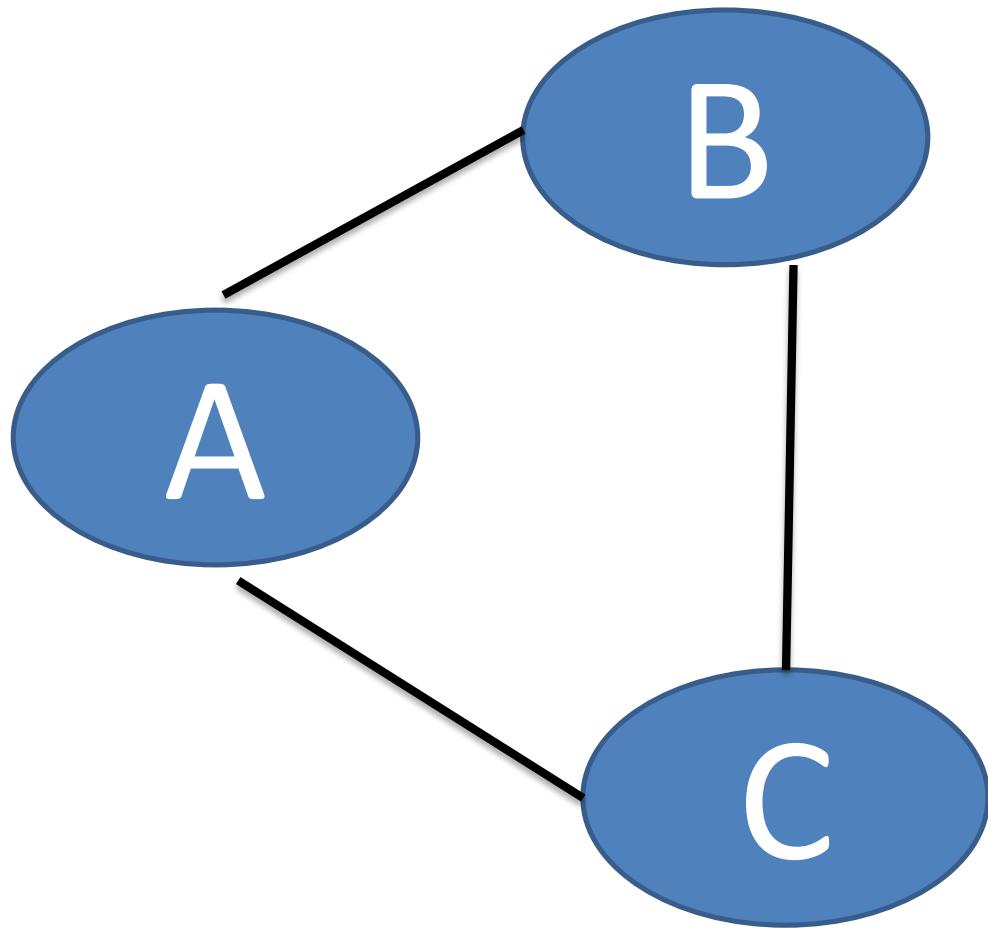
Thus, counting the number of triangles helps us to measure the extent to which a graph looks like a social network.

# Counting Triangles



Person A brings Person B and C into the Social Network.

# Counting Triangles



As the community is aging(matures), B and C interact with each other and become friends. Thus, a triangle is formed.

## Counting Triangles

Heavy hitter node: degree  $\sqrt{m}$

Heavy hitter triangle : all three nodes are heavy hitter nodes

### Algorithm for finding triangles

#### Preprocessing:

1. Compute the degree of each node
2. Construct an index (Hash table)

Key	value
$(v_1, v_2)$	edge exists or not

3. Construct another index (Hash table)

Key	value
$(v_1)$	adjacent nodes of $v_1$

## Counting Triangles

We shall order the nodes as follows. First, order nodes by degree.

Then, if  $v$  and  $u$  have the same degree, recall that both  $v$  and  $u$  are integers, so order them numerically.

**That is, we say  $v < u$**

**if and only if either**

- (i) The degree of  $v$  is less than the degree of  $u$ , or**
- (ii) The degrees of  $u$  and  $v$  are the same, and  $v < u$**

## Counting Triangles

### Heavy-Hitter Triangles:

There are only  $\sqrt{m}$  heavy-hitter nodes,

so we can consider all sets of three of these nodes.

There are  $O(m^{3/2})$  possible heavyhitter triangles,

and using the index on edges we can check if all three edges exist in  $O(1)$  time.

Therefore,  $O(m^{3/2})$  time is needed to find all the heavy-hitter triangles.

## Counting Triangles

### Other Triangles:

Therefore,  $O(m^{3/2})$  time is needed to find all the other triangles.

We now see that preprocessing takes  $O(m)$  time.

The time to find heavyhitter triangles is  $O(m^{3/2})$ , and so is the time to find the other triangles.

Thus, the total time of the algorithm is  $O(m^{3/2})$

# Optimality of the Triangle-Finding Algorithm

It turns out that the algorithm described above is the best possible.

To see why, consider a complete graph on  $n$  nodes.

This graph has  $m = n C 2$  edges

The number of triangles is  $nC3$  .

We know any algorithm will take  $\Omega(n^3)$  time on this graph.

## Counting Triangles

# Finding triangles using Map Reduce

Large graph-parallelism-speed up computation.

Finding triangles - Multiway joins-more efficient

Single MR job to count triangles

Idea

Nodes 1,2,3 ..n

Relation E represents edges

$E(A,B)$  is a tuple => there is an edge bw nodes A and B ( $A < B$ )

Using this relation,  
we can express the set of triangles of the graph  
whose edges are  $E$  by the natural join  
 $E(X, Y) \bowtie E(X, Z) \bowtie E(Y, Z)$

**SELECT e1.A, e2.B, e3.A**

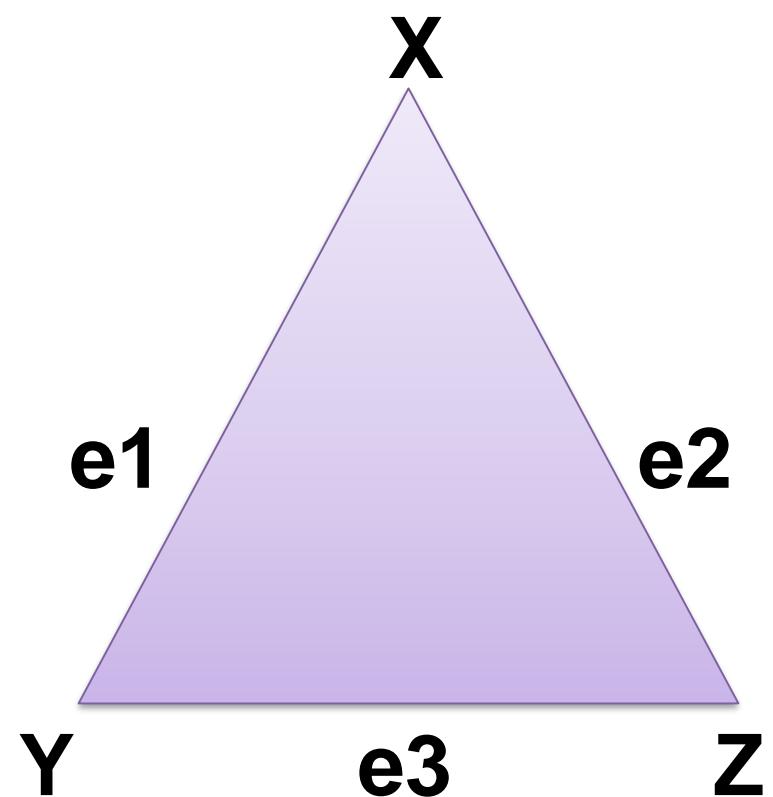
**FROM E e1, E e2, E e3**

**WHERE**

$e1.A = e2.A$  AND

$e1.B = e3.A$  AND

$e2.B = e3.B$



**Where A and B denote the two ends of an edge**

## **Map Tasks and Data Partitioning:**

In a MapReduce job, the input data is divided into chunks that are processed by different Map tasks. Each Map task processes its chunk independently.

## **Relation E:**

The relation E represents edges in a graph, typically formatted as tuples of nodes,

e.g.,  $E(u,v)$  indicates there is an edge between nodes u and v

### 3. Hashing for Buckets:

- Each node  $u$  and  $v$  is hashed to determine which bucket it belongs to.

The hash function  $h$  will generate a bucket number for each node:

- $h(u)$  gives the bucket for node  $u$ .
- $h(v)$  gives the bucket for node  $v$ .
- However, the node  $Z$  (which is required to complete the triangle) has not been defined, meaning the Map task cannot determine  $z$  directly.

#### 4. Sending to Reducers:

- Since the Map task cannot determine the exact bucket for  $Z$ , it must send the edge  $E(u, v)$  to all Reducers corresponding to every possible bucket for  $Z$ .

### **Redundant Data Transfer:**

The approach requires **sending the edge  $E(u,v)$  to multiple Reducers, potentially leading to inefficient data transfer and increased processing time**. However, this is necessary to ensure that all possible triangles are found in the Reduce phase.

### **Completing the Triangle:**

In the Reduce tasks, **edges from the Map phase will be processed**. The Reducers will look for combinations of edges that complete a triangle.

## Example

Let's say we have the following edges in  $E$ :

- $E(1, 2)$
- $E(2, 3)$
- $E(1, 3)$

Assuming:

- $h(1) = 0$
- $h(2) = 1$
- $h(3) = 2$

If the Map task processes the tuple  $E(1, 2)$ :

- It hashes 1 and 2 to buckets 0 and 1, respectively.
- The Map task must then send  $E(1, 2)$  to all Reducers corresponding to the buckets for  $Z$ , which might be 0, 1, and 2.
- This means  $E(1, 2)$  would be sent to three Reducers to ensure that any possible triangles involving nodes 1, 2, and any node  $Z$  (which could be either 1, 2, or 3) are processed.

## Using Fewer Reduce tasks

We should try to reduce the number of reduce tasks.

Then, we get a substantial decrease in the number of key-value pairs that must be communicated.

Instead of having to send each of the  $m$  edges to 3b Reduce tasks,

we need to send each edge to only  $b$  tasks.

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 4

TOPIC: Structured Data Queries with Hive

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

## Structured Data queries with Hive

**Apache Hive is a “data warehousing” framework built on top of Hadoop.**

Hive provides data analysts with a familiar **SQL-based interface to Hadoop**, which allows them to attach structured schemas to data in HDFS and access and analyze that data using SQL queries.

Hive has made it possible for developers who are fluent in SQL to leverage the scalability and resilience of Hadoop without requiring them to learn Java or the native MapReduce API

## Structured Data queries with Hive

Hive provides its own dialect of SQL called the **Hive Query Language, or HQL**.

**HQL supports many commonly used SQL statements**, including

1. **Data definition statements (DDLs)** (e.g., CREATE DATABASE/SCHEMA/TABLE),
2. **Data manipulation statements (DMSs)** (e.g., INSERT, UPDATE, LOAD), and
3. **Data retrieval queries (e.g., SELECT).**
4. Hive also supports **integration of custom user-defined functions**, which can be written in Java or any language supported by Hadoop Streaming, that extend the built-in functionality of HQL.

## Structured Data queries with Hive

Hive commands and HQL queries are compiled into an execution plan or a series of HDFS operations and/or MapReduce jobs, which are then executed on a Hadoop cluster.

Thus, Hive has inherited certain limitations from HDFS and MapReduce that constrain it from providing key online transaction processing (OLTP) features that one might expect from a traditional database management system.

### Limitations

In particular, because HDFS is a write-once, read-many (WORM) file system and does not provide in-place file updates,

Hive is not very efficient for performing row-level inserts, updates, or deletes.

## Structured Data queries with Hive

In fact, these **row-level updates are only recently supported as of Hive release 0.14.0.**

Additionally, **Hive queries entail higher-latency** due to the overhead required to generate and launch the compiled MapReduce jobs on the cluster;

**even small queries that would complete within a few seconds on a traditional RDBMS may take several minutes to finish in Hive.**

### Merits

On the plus side, Hive provides the **high-scalability and high-throughput** that you would expect from any Hadoop-based application.

## Structured Data queries with Hive

As a result, it is **very well suited to batch-level workloads for online analytical processing (OLAP) of very large datasets at the terabyte and petabyte scale.**

In this section, we explore some of Hive's primary features and write HQL queries to perform data analysis.

## Hive Command Line Interface(CLI)

Hive's installation comes packaged with a handy command-line interface (CLI), which we will use to interact with Hive and run our HQL statements.

To start the Hive CLI from the \$HIVE\_HOME:

```
~$ cd $HIVE_HOME  
/srv/hive$ bin/hive
```

This will initiate the CLI and bootstrap the logger (if configured) and Hive history file, and finally display a Hive CLI prompt:

```
hive>
```

At any time, you can exit the Hive CLI using the following command:

```
hive> exit;
```

**Hive can also run in non-interactive mode** directly from the command line by passing the **filename option, -f**, followed by the **path to the script to execute:**

```
~$ hive -f ~/hadoop-fundamentals/hive/init.hql
```

Additionally, the **quoted-query-string option, -e**, allows you to run inline commands from the command line:

```
~$ hive -e 'SHOW DATABASES;'
```

You can view the **full list of Hive options for the CLI by using the -H flag**:

```
~$ hive -H
```

Non-interactive mode is very handy for running saved scripts, but the CLI gives us the ability to easily debug and iterate on queries in Hive.

usage: hive	
-d,--define <key=value>	Variable substitution to apply to hive commands. e.g. -d A=B or --define A=B
--database <database>	Specify the database to use
-e <quoted-query-string>	SQL from command line
-f <filename>	SQL from files
-H,--help	Print help information
-h <hostname>	connecting to Hive Server on remote host
--hiveconf <property=value>	Use value for given property
--hivevar <key=value>	Variable substitution to apply to hive commands. e.g. --hivevar A=B
-i <filename>	Initialization SQL file
-p <port>	connecting to Hive Server on port number
-S,--silent	Silent mode in interactive shell
-v,--verbose	Verbose mode (echo executed SQL to the console)

## Hive Query Language (HQL)

We will learn HiveQL (HQL) statements to create a Hive database, load the database with data that resides in HDFS, and perform query-based analysis on the data.

Creating a database in Hive is **very similar to creating a database in a SQL-basedRDBMS**, by using the CREATE DATABASE or CREATE SCHEMA statement:

**hive> CREATE DATABASE log\_data;**

When Hive creates a new database, the schema definition data is stored in the Hive metastore(Central repository to store metadata about databases and tables).

*(Eg for metadata: name of the database, owner, location and timestamp)*

Hive will raise an error if the database already exists in the metastore;

we can check for the existence of the database by using  
IF NOT EXISTS:

**hive> CREATE DATABASE IF NOT EXISTS log\_data;**

We can then run SHOW DATABASES to verify that our database has been created.

Hive will return all databases found in the metastore, along with the default Hive database:

**hive> SHOW DATABASES;**

OK

default

log\_data

Time taken: 0.085 seconds, Fetched: 2 row(s)

Additionally, we can set our working database with the USE command:

**hive> USE log\_data;**

## Creating tables

Hive provides a SQL-like CREATE TABLE statement, which in its simplest form takes a table name and column definitions:

```
CREATE TABLE apache_log ( host STRING, identity STRING, user STRING, time STRING, request STRING, status STRING, size STRING, referer STRING, agent STRING );
```

However, because Hive data is stored in the file system, usually in HDFS or the local file system,

the CREATE TABLE command also **takes optional clauses** to specify the row format with the **ROW FORMAT clause** that tells Hive how to read each row in the file and map to our columns.

For example, we could indicate that the data is in a delimited file with fields delimited by the tab character:

```
hive> CREATE TABLE shakespeare ( lineno STRING,  
linetext STRING ) ROW FORMAT DELIMITED FIELDS  
TERMINATED BY '\t';
```

- **FIELDS TERMINATED BY '\t':**
  - This specifies that the fields (columns) in the data are separated by a tab character (\t). This means that when data is loaded into this table, each row of data must have its fields separated by tabs.
- **Example Data Format**
- The expected data format in the file for the shakespeare table would look like this (using tab characters for separation):

- 1 **The first line of the text**
- 2 **The second line of the text**
- 3 **Another line of text**

Once we've created the table, we can use DESCRIBE to verify our table definition:

```
hive> DESCRIBE apache_log;
```

OK

host	string	from deserializer
identity	string	from deserializer
user	string	from deserializer
time	string	from deserializer
request	string	from deserializer
status	string	from deserializer
size	string	from deserializer
referrer	string	from deserializer
agent	string	from deserializer

Time taken: 0.553 seconds. Fetched: 9 row(s)

Note that in this particular table, all columns are defined with the Hive primitive data type, string.

Hive supports many other primitive data types that will be familiar to SQL users and generally correspond to the primitive types supported by Java.

*Table 6-1. Hive primitive data types*

Type	Description	Example
TINYINT	8-bit signed integer, from -128 to 127	127
SMALLINT	16-bit signed integer, from -32,768 to 32,767	32,767
INT	32-bit signed integer	2,147,483,647
BIGINT	64-bit signed integer	9,223,372,036,854,775,807
FLOAT	32-bit single-precision float	1.99

DOUBLE	64-bit double-precision float	3.14159265359
--------	-------------------------------	---------------

BOOLEAN	True/false	true
STRING	2 GB max character string	hello world
TIMESTAMP	Nanosecond precision	1400561325

In addition to the primitive data types, Hive also supports complex data types, listed in [Table 6-2](#), that can store a collection of values.

*Table 6-2. Hive complex data types*

Type	Description	Example
ARRAY	Ordered collection of elements. The elements in the array must be of the same type.	recipients ARRAY<email:STRING>
MAP	Unordered collection of key/value pairs. Keys must be of primitive types and values can be of any type.	files MAP<filename:STRING, size:INT>
STRUCT	Collection of elements of any type.	address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>

This may seem awkward at first, because **relational databases generally don't support collection types, minimize data duplication.**

However, in a big data system like Hive where we are processing large volumes of unstructured data by sequentially scanning off disk, the **ability to read embedded collections provides a huge benefit in retrieval performance.**

## Loading data

With our **table created and schema defined, we are ready to load the data into Hive.**

It's important to note one important **distinction between Hive and traditional RDBMSs with regards to schema enforcement:**

Hive implements Schema on read approach whereas traditional RDBMS implements Schema on Write approach.

Traditional relational databases enforce the schema on writes by rejecting any data that does not conform to the schema as defined.

Feature	Schema on Read	Schema on Write
Definition	Schema is applied during data read.	Schema is defined before data write.
Data Structure	Supports semi-structured/unstructured data.	Requires structured data conforming to a schema.
Flexibility	High flexibility; schema can evolve.	Low flexibility; schema changes require migrations.
Data Integrity	Less strict; integrity enforced at query time.	Strict enforcement; integrity checked at data insertion.
Use Case	Ideal for big data and analytics.	Suited for transactional systems (OLTP).
Performance	May lead to slower queries due to schema evaluation at runtime.	Faster queries as schema is known at write time.
Example Systems	Hive, NoSQL databases (like MongoDB).	Traditional RDBMS (like MySQL, Oracle).

**Load operations are purely copy/move operations that move data files into locations corresponding to Hive tables.**

Data loading in Hive is done in **batch-oriented fashion** using a bulk **LOAD DATA command** or by inserting results from another query with the **INSERT command**.

To start, let's copy our Apache log data file to HDFS and then load it into the table we created earlier:

```
~$ hadoop fs –mkdir statistics
```

```
~$ hadoop fs –mkdir statistics/log_data
```

```
~$ hadoop fs –copyFromLocal  
~/hadoop-fundamentals/data/log_data/apache.log  
\ statistics/log_data/
```

You can verify that the apache.log file was successfully uploaded to HDFS with the tail command:

```
~$ hadoop fs –tail statistics/log_data/apache.log
```

Once the file has been uploaded to HDFS, return to the Hive CLI and use the log\_data database:

```
~$ $HIVE_HOME/bin/hive
```

```
hive> use log_data;
```

```
OK
```

```
Time taken: 0.221 seconds
```

We'll use the LOAD DATA command and specify the HDFS path to the logfile, writing the contents into the apache\_log table:

```
hive> LOAD DATA INPATH 'statistics/logs/data/apache.log' OVERWRITE INTO TABLE apache_log;
```

Loading data to table log\_data.apache\_log

**rmr: DEPRECATED: Please use 'rm -r' instead.**

Deleted

hdfs://localhost:9000/user/hive/warehouse/log\_data.db/apache\_log

Table log\_data.apache\_log

**stats: [numFiles=1, numRows=0, totalSize=52276758, rawDataSize=0]**

OK

Time taken: 0.902 seconds

**LOAD DATA** is Hive's bulk loading command. **INPATH** takes an argument to a path on the default file system (in this case, HDFS).

We can also specify a path on the local file system by using **LOCAL INPATH** instead. Hive proceeds to move the file into the warehouse location.

If the **OVERWRITE** keyword is used, then any existing data in the target table will be deleted and replaced by the data file input; otherwise, the new data is added to the table.

Once the data has been copied and loaded, Hive outputs some statistics on the loaded data; although the `num_rows` reported is 0, you can verify the actual count of rows by running a **SELECT COUNT** (output truncated)

```
hive> SELECT COUNT(1) FROM apache_log;
```

Total MapReduce jobs = 1

Launching Job 1 out of 1 ...

OK

726739

Time taken: 34.666 seconds, Fetched: 1 row(s)

As you can see, when we run this Hive query it actually executes a MapReduce job to perform the aggregation.

After the MapReduce job has executed, you should see that the apache\_log table now contains 726,739 rows.

# INT 404R01 BIG DATA ANALYTICS

# B.Tech CSE 'A'

## Year/Sem: IV/VII

# Unit 4

## TOPIC: HBase

Handled by,  
Dr.M.Devi Sri Nandhini  
AP III/School of Computing

## Hbase

In the previous section, we learned how we could use Hive to perform SQLbased analysis on large, structured datasets stored in HDFS.

However, we observed that while Hive provides a familiar data manipulation paradigm within Hadoop, it doesn't change the storage and processing paradigm.

Hive still utilizes HDFS and MapReduce in a batch-oriented fashion.

Recall that because HDFS is designed as a write-once, read-many (WORM) file system, it is optimized for sequential reads and not efficient for use cases that require frequent or fast row-level updates to the data.

This data access pattern is often called “**random access**” and the number of **applications** that require such **real-time, low-latency read/write access** are **growing rapidly**.

Take, for example, the explosion of **real-time sensor and telemetry applications, such as those used by NOAA to gather weather data from remote stations or by NASA’s Deep Space Network to record data transmissions from unmanned spacecraft.**

These applications **must store and process an enormous volume of event data** from potentially numerous transmission devices at an **extremely fast rate**,

Thus, for use cases that **require random, real-time read/write access to data**, we need to **look outside of standard MapReduce and Hive** for our data persistence and processing layer.

In a relational model, rows are sparse but columns are not.

That is, upon inserting a new row to a table, the database allocates storage for every column regardless of whether a value exists for that field or not.

However, in applications where data is represented as a collection of arbitrary fields or sparse columns, each row may use only a subset of available columns, which can make a standard relational schema both a wasteful and awkward fit.

## **NoSQL(Not only SQL and Columnar databases)**

NoSQL databases are designed to handle large-scale data storage and retrieval, with flexibility for semi-structured, unstructured, or rapidly changing data.

They are often used in big data applications and real-time web applications.

Common types of NoSQL databases include:

**Document stores** (e.g., MongoDB, CouchDB)

**Key-value stores** (e.g., Redis, DynamoDB)

**Column-family stores** (e.g., Apache Cassandra, HBase)

**Graph databases** (e.g., Neo4j)

**HBase is classified as a column-family or column-oriented database**, modeled on Google's BigTable architecture. This architecture allows HBase to provide:

- Random (row-level) read/write access**
- Strong consistency**
- “Schema-less” or flexible data modeling**

**HBase organizes data into tables that contain rows.**

Within a table, **rows are identified by their unique row key**, which do not have a data type and are instead stored and **treated as a byte array**.

Row keys are **similar to the concept of primary keys in relational databases**, in that they are automatically indexed; in HBase, table **rows are sorted by their row key** and because row keys are byte arrays

**Almost anything can serve as a row key from strings to binary representations of longs or even serialized data structures.**

**HBase stores its data as key/value pairs**, where all table lookups are performed via the table's row key, or unique identifier to the stored record data.

**Data within a row is grouped into column families, which consist of related columns.**

**Visually, you can picture an HBase table that holds census data for a given population.**

**Each row represents a person** and is accessed via a unique ID rowkey

**with column families for**

**1) personal data which contains columns for name and address, and**

**2)demographic info which contains columns for birthdate and gender.**

This example is shown in Figure

## PERSON TABLE

row key	personal_data	demographic	...		
<b>PersonID</b> 1 2 3 ... 500,000,000	<b>Name</b> H. Houdini D. Copper Merlin ... F. Cadillac	<b>Address</b> Budapest, Hungary New Jersey, USA Stonehenge, England ... Nevada, USA	<b>BirthDate</b> 1926-10-31 1956-09-16 1136-12-03 ... 1964-01-07	<b>Gender</b> M M F ... M	...

Figure 6-1. Census data as an HBase schema

**Storing data in columns** rather than rows has particular benefits for data warehouses and analytical databases where **aggregates are computed** over large sets of data with potentially sparse values, **where not all columns values are present.**

Although **column families are very flexible**, in practice a column family is **not entirely schema-less**.

**Column families are actually defined up front before we can begin inserting data into a particular row and column**, because they impact the physical arrangement of data stored in HBase.

However, the actual columns that make up a row can be determined and created on an as-needed basis.

**In fact, each row can have a different set of columns.**

Figure shows an example HBase table with two rows where **first row key utilizes three column families** and the **second row key utilizes just one column**

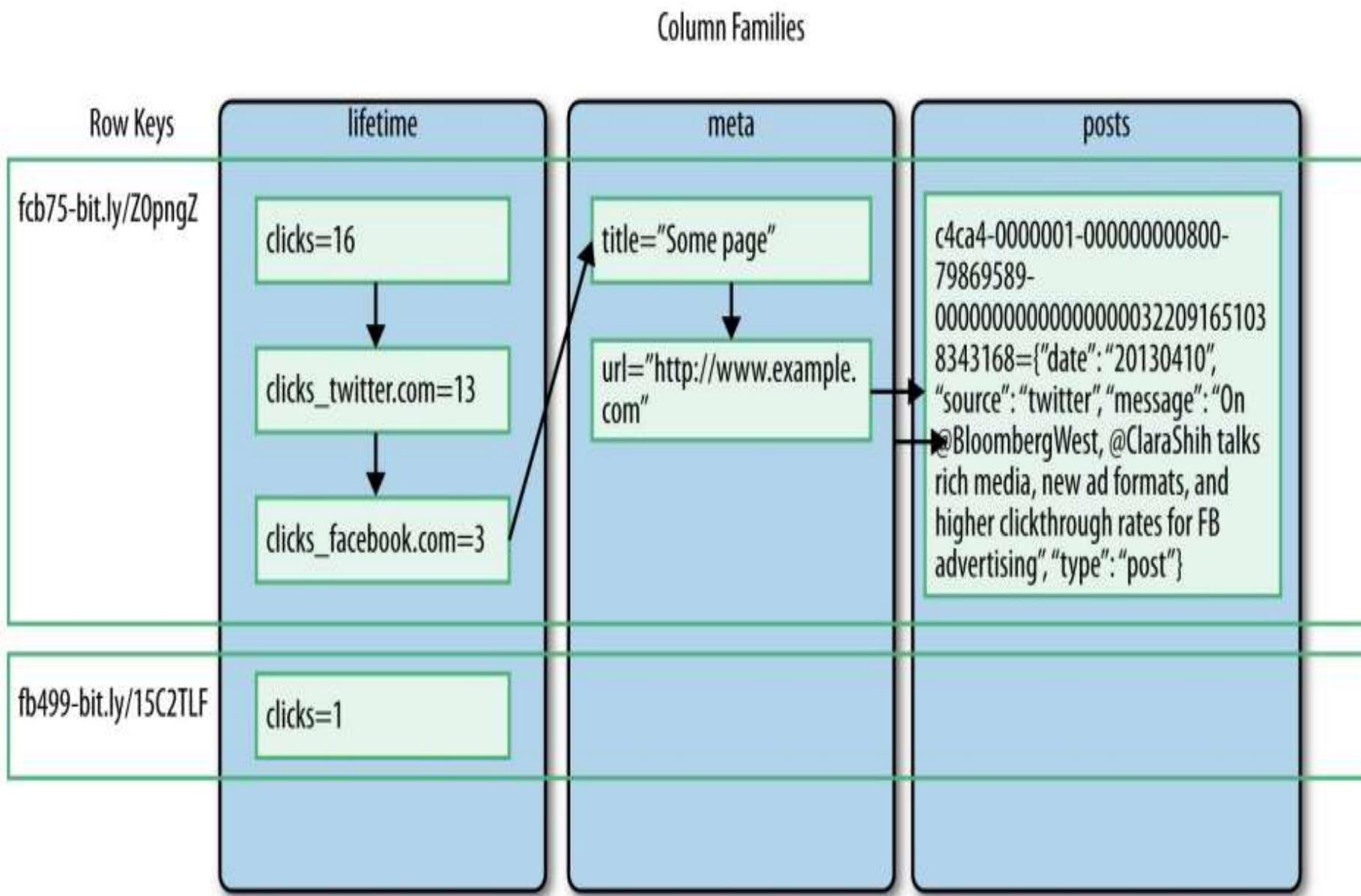


Figure 6-2. Social media events with sparse columns

# Sample code to create Hbase table:

```
create      'table_name',      'column_family1',
'column_family2', ...
```

Let's create a sample HBase table named users with two column families: personal\_info and account\_info.

```
create 'users', 'personal_info', 'account_info'
```

```
list
```

```
Users
```

**describe 'users'**

Table users is ENABLED  
users

## COLUMN FAMILIES DESCRIPTION

{**NAME => personal\_info**, VERSIONS => 1, TTL =>  
N/A, MIN\_VERSION => 0, MAX\_VERSION => 1,  
BLOCKCACHE => true, BLOOMFILTER => NONE,  
COMPRESSION => NONE}

{**NAME => account\_info**, VERSIONS => 1, TTL =>  
N/A, MIN\_VERSION => 0, MAX\_VERSION => 1,  
BLOCKCACHE => true, BLOOMFILTER => NONE,  
COMPRESSION => NONE}

Compression algorithms: Gzip, snappy, LZ4 etc.

**Another interesting feature of HBase and BigTable-based column-oriented databases is that the table cells, or the intersection of row and column coordinates, are versioned by timestamp, stored as a long integer representing milliseconds since January 1, 1970 UTC.**

HBase is thus also described as being a **multidimensional map** where time provides the third dimension, as shown in Figure 6-3.

**The time dimension is indexed in decreasing order, so that when reading from an HBase store, the most recent values are found first.**

The contents of a cell can be **referenced by a {rowkey, column, timestamp} tuple, or we can scan for a range of cell values by time range.**

The diagram illustrates HBase timestamp versioning using a 3D grid representation. The vertical axis represents the timestamp levels, labeled t1, t2, and t3 from top to bottom. The horizontal axis represents the columns, labeled key, cf1:colA, cf1:colB, cf2:colC, and cf2:colD. The depth axis represents the rows, labeled 1, 2, and 3 from top to bottom.

key	cf1:colA	cf1:colB	cf2:colC	cf2:colD
1	ca1-t3		cc1-t2	cd1-t3
2	ca2-t3	cb2-t3		cd2-t3
3	ca3-t3		cc3-t3	

Figure 6-3. HBase timestamp versioning

Now that we've covered the key features of HBase schema design, we'll learn how to design and query a simple HBase table for a hypothetical real-time linksharing application

## Real-Time Analytics with HBase

HBase schemas can be created or updated with the HBase Shell

or

with the Java API, using the HBaseAdmin interface class.

Additionally, HBase supports a number of other clients that can be used to support non-Java programming languages, including a REST API interface, Thrift, and Avro.

However, in a real-world setting, you would write your application using the native Java API or supported client libraries.

## Generating a schema

When designing schemas in HBase, it's important to think in terms of the **column-family structure** of the data model and how it affects data access patterns.

While schema definition for traditional relational databases is primarily driven by the accurate representation of the entities and relationships and performance considerations like joins.

**Successful HBase schema definition tends to be driven by the intended use cases of the application.**

Furthermore, because **HBase doesn't support joins** and provides only a single indexed rowkey, we must be careful to **ensure that the schema can fully support all use cases**.

The **good news is that because HBase allows dynamic column definition at runtime**, we have quite a bit of flexibility even after table creation to modify and scale our schema

## Namespaces, tables, and column families

First, we need to declare the table name, and at least one column-family name at the time of table definition.

We can also declare our own optional namespace (supported as of Apache HBase v0.96.0) to serve as a logical grouping of tables, analogous to a database in relational database systems.

If no namespace is declared, HBase will use the default namespace:

**hbase> create 'linkshare', 'link'**

0 row(s) in 1.5740 seconds

We just created a single table called linkshare in the default namespace with one column-family, named link.

To alter the table after creation, such as changing or adding column families, we need to first disable the table so that clients will not be able to access the table during the alter operation:

**hbase> disable 'linkshare'**

0 row(s) in 1.1340 seconds

**hbase> alter 'linkshare', 'statistics'**

Updating all regions with the new schema...

1/1 regions updated.

Done. 0 row(s) in 1.1630 seconds

We can then re-enable the table using the enable command:

**hbase> enable 'linkshare'**

0 row(s) in 1.1930 seconds

Use the `describe` command to verify that the table contains the two expected column families with the default configurations:

**hbase> describe 'linkshare'**

Table linkshare is ENABLED

COLUMN FAMILIES DESCRIPTION

{NAME => 'link', .....}

{NAME => 'statistics', .....

We've created a single HBase table (linkshare) with two column families (link and statistics), **but our table does not yet contain any rows.**

Before we insert row data, **we need to determine how to design our row key**

## **Row keys**

Good row key design affects how we query the table.

**Row Key Sorted Order:** HBase automatically stores data in lexicographically sorted order based on the row key.

This allows efficient range scans, meaning that scanning through rows with sequentially similar row keys can be done quickly and with minimal overhead.

In HBase, **row key design plays a crucial role in optimizing performance and**

**ensuring that the data is distributed evenly across the RegionServers to avoid bottlenecks such as RegionServer hotspotting.**

Thus, in addition to enabling our data access use cases, we also need to be mindful to account for row key distribution across regions.

For the current example, let's assume that we will use the unique reversed link URL for the row key.

## Inserting data with put

Now our table is ready to start storing data.

In our linkshare application, we want to store descriptive data about the link, such as its title, while maintaining a frequency counter that tracks the number of times the link has been shared.

We can insert, or put, a value in a cell at the specified table/row/column and optionally timestamp coordinates.

To put a cell value into table linkshare at row with row key org.hbase.www under column-family link and column title marked with the current timestamp, do:

```
hbase> put 'linkshare', 'org.hbase.www', 'link:title',  
'Apache HBase'
```

```
hbase> put 'linkshare', 'org.hadoop.www', 'link:title',  
'Apache Hadoop'
```

```
hbase> put 'linkshare', 'com.oreilly.www', 'link:title',  
'O'Reilly.com'
```

HBase provides a **more efficient way to handle counters via the incr (increment) command.**

To increment the 'share' counter in the row identified by 'org.hbase.www' under the column family statistics, column share, by 1:

**hbase> incr 'linkshare', 'org.hbase.www',  
'statistics:share', 1**

Similarly, to increment the 'like' counter for the same row and column family:

**hbase> incr 'linkshare', 'org.hbase.www',  
'statistics:like', 1**

To access a counter's current value any time using the get\_counter command, specifying the table name, row key, and column:

```
hbase> incr 'linkshare', 'org.hbase.www',  
'statistics:share', 1 (COUNTER VALUE is now 2)
```

```
hbase> get_counter 'linkshare', 'org.hbase.www',  
'statistics:share', 'dummy'
```

COUNTER VALUE = 2

HBase provides two general methods to retrieve data from a table:

- (1)the **get command** performs lookups by row key to retrieve attributes for a specific row, and
- (2)the **scan command**, which takes a set of filter specifications and iterates over multiple rows based on the indicated specifications.

### **(1)Get row or cell values**

In its simplest form, the **get command accepts the table name followed by the row key, and returns the most recent version timestamp and cell value for all columns in the row:**

hbase> get 'linkshare', 'org.hbase.www'

hbase> get 'linkshare', 'org.hbase.www'

COLUMN	CELL
link:title	timestamp=1422145743298, value=Apache HBase
statistics:like	timestamp=1422153344211, value=\x00\x00\x00\x00\x00\x00\x00\x1F
statistics:share	timestamp=1422153337498, value=\x00\x00\x00\x00\x00\x00\x00\x02

3 row(s) in 0.0310 seconds

There is also a shortcut to specify column parameters in a get by just appending the comma-delimited column names after the row key:

```
hbase> get 'linkshare', 'org.hbase.www', 'link:title'
```

```
hbase> get 'linkshare', 'org.hbase.www', 'link:title',  
'statistics:share'
```

To specify a time range of values we are interested in, we pass in a TIMERANGE parameter with start and end timestamps in milliseconds:

```
hbase> get 'linkshare', 'org.hbase.www', {TIMERANGE  
=> [1399887705673, 1400133976734]}
```

This command retrieves the two most recent versions of the share value from the statistics column family for the row with the key org.hbase.www in the linkshare table.

```
hbase> get 'linkshare', 'org.hbase.www', {COLUMN =>  
'statistics:share', VERSIONS => 2}
```

## (2) Scan rows

A scan operation is akin to database cursors or iterators, and takes advantage of the underlying sequentially sorted storage mechanism, **iterating through row data to match against the scanner specifications.**

With scan, we can scan an entire HBase table or specify a range of rows to scan.

Using scan is **similar** to using the get command; it also accepts **COLUMN**, **TIMESTAMP**, **TIMERANGE**, and **FILTER** parameters.

However, instead of specifying a single row key, you can specify an optional **STARTROW** and/or **STOPROW** parameter, which can be used to limit the scan to a specific range of rows.

If neither **STARTROW** nor **STOPROW** are provided, the scan operation will scan through the entire table.

You can, in fact, call scan with the table name to display all the contents of a table:

**hbase> scan 'linkshare'**

ROW	COLUMN+CELL
com.oreilly.www	column=link:title, timestamp=1422153270279, value=O'Reilly.com
org.hadoop.www	column=link:title, timestamp=1422153262507, value=Apache Hadoop
org.hbase.www	column=link:title, timestamp=1422145743298, value=Apache HBase
org.hbase.www	column=statistics:like, timestamp=1422153344211, value=\x00\x00\x00\x00\x00\x00\x00\x1F
org.hbase.www	column=statistics:share, timestamp=1422153337498, value=\x00\x00\x00\x00\x00\x00\x00\x02

3 row(s) in 0.0290 seconds

Keep in mind that the rows in HBase are stored in lexicographical order.

Let's retrieve the link:title column but limit our scan to the rows starting with row key org.hbase.www:

```
hbase> scan 'linkshare', {COLUMNS => ['link:title'],
STARTROW => 'org.hbase.www'}
```

ROW	COLUMN+CELL
org.hbase.www	column=link:title, timestamp=1453184861236, value=Apache HBase

1 row(s) in 0.0250 seconds

```
hbase> scan 'linkshare', {COLUMNS =>  
['link:title'], STARTROW => 'org'}
```

ROW	COLUMN+CELL
org.hadoop.www	column=link:title, timestamp=1422153262507, value=Apache Hadoop
org.hbase.www	column=link:title, timestamp=1422145743298, value=Apache HBase

2 row(s) in 0.0210 seconds

## Filters

HBase provides a **number of filter classes** that can be applied to further filter the row data returned from a get or scan operation.

HBase offers several filter classes that can be **applied to Get or Scan operations, allowing you to filter data on the server side before it is returned to the client.**

This server-side filtering **reduces the amount of data transferred over the network**, improving efficiency and performance, particularly when working with large datasets.

## **Using filters in HBase scans helps:**

**Reduce Network Traffic:** Filters allow HBase to process the filtering logic server-side, so only the filtered rows are sent back to the client.

**Improve Performance:** By eliminating rows that are not relevant before they are sent over the network, filters can greatly **reduce the time it takes to scan large datasets.**

**Minimize Data Processing on the Client:** Instead of receiving all data and filtering it on the client, you can filter results directly at the source.

# **Some of HBase's available filters include:**

**RowFilter** Used for data filtering based on row key values

**ColumnRangeFilter** Allows efficient **intra-row scanning**, can be used to get a slice of the columns of a **very wide row** (i.e., you have a million columns in a row but you only want to look at columns bbbb-bbdd)

**SingleColumnValueFilter** Filters the rows based on a single column value

# **RegexStringComparator**

**Used to test if a given regular expression matches a cell value in the column**

The HBase Java API provides a **Filter interface** and **abstract FilterBase class** plus a number of specialized **Filter subclasses**.

**Custom filters** can also be created by **subclassing the FilterBase abstract class** and **implementing the key abstract methods**.

**HBase filters are best applied by utilizing the HBase API within a Java program** as they generally require importing several dependent filter and comparator classes, but we can demonstrate a simple example of a filter in the shell.

To begin, we need to import the necessary classes, including the `org.apache.hadoop.hbase.util.Bytes` to convert our column family, column, and values into bytes, and the filter and comparator classes:

`hbase> import org.apache.hadoop.hbase.util.Bytes`

`hbase> import  
org.apache.hadoop.hbase.filter.SingleColumnValueFilter`

`hbase> import  
org.apache.hadoop.hbase.filter.BinaryComparator`

`hbase> import  
org.apache.hadoop.hbase.filter.CompareFilter`

Next, we'll create a filter that limits the results to rows where the **statistics:like** counter column value is greater than or equal to 10:

hbase> likeFilter =

SingleColumnValueFilter.new(Bytes.toBytes('statistics'),

Bytes.toBytes('like'),

CompareFilter::CompareOp.valueOf('GREATER\_OR\_EQ

UAL'),

BinaryComparator.new(Bytes.toBytes(10)))

## **Bytes.toBytes('statistics'):**

Converts the string 'statistics' into a byte array. This represents the **column family** in HBase where the like column is located.

## **Bytes.toBytes('like'):**

Converts the string 'like' into a byte array. This represents the **column qualifier** (i.e., the column name) inside the statistics column family.

## **CompareFilter::CompareOp.valueOf('GREATER\_OR\_EQUAL'):**

Specifies the **comparison operator** as GREATER\_OR\_EQUAL. This means the filter will return rows where the value in the like column is **greater than or equal to** the specified value (in this case, 10).

## **BinaryComparator.new(Bytes.toBytes(10)):**

This is the **comparator** that compares the binary representation of the value in the like column with the binary representation of the value 10.

Bytes.toBytes(10) converts the integer 10 into a byte array for comparison.

## **Interpretation of the Filter:**

The filter is applied to the **column family** 'statistics' and the **column qualifier** 'like'.

The filter checks if the value stored in the 'like' column is **greater than or equal to 10**.

If the value is greater than or equal to 10, the row is included in the result set; otherwise, the row is filtered out.

This filter will return rows where the value in the statistics:like column is 10 or more.

And because we don't have a value for this column for every row, we need to set a flag that tells this filter to skip any rows without a value in this column:

**hbase> likeFilter.setFilterIfMissing(true)**

At this point, we can run our scan operation with the filter we configured:

**hbase> scan 'linkshare', { FILTER => likeFilter }**

ROW COLUMN+CELL  
org.hbase.www column=link:title, timestamp=1422145743298,  
value=Apache HBase

org.hbase.www column=statistics:like, timestamp=1422153344211,  
value=\x00\x00\x00\x00\x00\x00\x00\x1F  
org.hbase.www column=statistics:share, timestamp=1422153337498,  
value=\x00\x00\x00\x00\x00\x00\x00\x02

1 row(s) in 0.0470 seconds

This should return all rows that contain a column value  
for statistics:like that is greater than or equal to 10;

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 4

TOPIC: Data Ingestion: Relational Data with Sqoop

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

## Data Ingestion: Relational Data with Sqoop

One of **Hadoop's greatest strengths** is that it's inherently **schema-less** and

It **can work with any type or format of data** regardless of structure (or lack of structure) from any source.

However, **in cases where the input data is already structured** because it resides in a relational database, it would be **convenient to leverage this known schema to import the data into Hadoop in a more efficient manner** than uploading CSVs to HDFS and parsing them manually.

## **Data Ingestion: Relational Data with Sqoop**

**Sqoop is designed to transfer data between relational database management systems (RDBMS) and Hadoop.**

**It automates most of the data transformation process**, relying on the RDBMS to provide the schema description for the data to be imported.

Sqoop can be a **very useful link in the analytics pipeline** for data infrastructures that involve **relational databases as a primary or intermediary data store**

## Importing Relational Data with Swoop

**Swoop (SQL-to-Hadoop)** is a relational database import and export tool created by Cloudera, and is now an Apache top-level project.

**Swoop** is designed to transfer data between a relational database like MySQL or Oracle, into a Hadoop data store, including HDFS, Hive, and HBase.

It automates most of the data transfer process by reading the **schema information directly from the RDBMS**. Swoop then uses **MapReduce** to import and export the data to and from Hadoop.

**Sqoop gives us the flexibility to maintain our data in its production state while copying it into Hadoop to make it available for further analysis without modifying the production database.**

We'll walk through a few ways to use Sqoop to import data from a MySQL database into various Hadoop data stores, including HDFS, Hive, and HBase.

We assume that you have installed the latest stable version of Sqoop that is compatible with your version of Hadoop, and

that Hadoop is configured in pseudo-distributed mode with all HDFS and YARN processes running

## **Importing from MySQL to HDFS**

When importing data from relational databases like MySQL, **Sqoop reads the source database to gather the necessary metadata** for the data being imported.

**Sqoop then submits a map-only Hadoop job to transfer the actual table** data based on the metadata that was captured in the previous step.

This **job produces a set of serialized files**, which may be delimited text files, binary format (e.g., Avro), or SequenceFiles containing a copy of the imported table or datasets.

**By default, the files are saved as comma-separated files to a directory on HDFS with a name that corresponds to the source table name.**

We'll use these defaults to export data from MySQL to HDFS.

Assuming that **you have set up MySQL**, let's go ahead and **create a sample database with some tables and data**.

We'll start by creating a **database called energydata** and a table called **average\_price\_by\_state**:

**~\$ mysql -uroot -p**

**mysql> CREATE DATABASE energydata;**

Query OK, 1 row affected (0.00 sec)

**mysql> GRANT ALL PRIVILEGES ON energydata.\* TO '%@'localhost';**

Query OK, 0 rows affected (0.00 sec)

**mysql> GRANT ALL PRIVILEGES ON energydata.\* TO "@localhost";**

Query OK, 0 rows affected (0.00 sec) mysql> USE energydata;

```
mysql> CREATE TABLE average_price_by_state( year INT NOT  
NULL, state VARCHAR(5) NOT NULL, sector VARCHAR(255),  
residential DECIMAL(10,2), commercial DECIMAL(10,2),  
industrial DECIMAL(10,2), transportation DECIMAL(10,2), other  
DECIMAL(10,2), total DECIMAL(10,2) );
```

**Query OK, 0 rows affected (0.02 sec)**

```
mysql> quit;
```

The data that we load into the average\_price\_by\_state table is provided by the [US Energy Information Administration](#) and includes the annual data from 1990–2012 on the average energy price per kilowatt hour (KwH) by state and provider type.

You can find the CSV named **avgprice\_kwh\_state.csv** within the **GitHub repo's /data directory**.

Download this CSV and load it into the MySQL table we just created:

```
~$ mysql -h localhost -u root -p energydata --local-infile=1
```

```
mysql> LOAD DATA LOCAL INFILE '/home/hadoop/hadoop-fundamentals/data/avgprice_kwh_state.csv'  
INTO TABLE average_price_by_state  
FIELDS TERMINATED BY ','  
LINES TERMINATED BY '\n'  
IGNORE 1 LINES;
```

Query OK, 3272 rows affected, 6 warnings (0.03 sec)  
Records: 3272 Deleted: 0 Skipped: 0 Warnings: 6

```
mysql> quit;
```

Before we proceed to run the sqoop import command, verify that HDFS and YARN are started with the jps command:

```
~$ sudo su hadoop  
hadoop@ubuntu:~$ jps  
  
4051 NodeManager  
31134 Jps  
3523 DataNode  
3709 SecondaryNameNode  
3375 NameNode  
3921 ResourceManager
```

At this point, we can import the data in table `average_price_by_state` into HDFS by using the `import` command.

We can specify the source database's connection string, username, and tablename with the `--connect` option, `--username` option, and `--table` option, respectively.

We'll set the optional `-m` flag to 1 to indicate that this job should use a single map task:

```
/srv/sqoop$ sqoop import --connect  
jdbc:mysql://localhost:3306/energydata --username root --  
table average_price_by_state -m 1
```

15/01/20 22:47:35 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6  
15/01/20 22:47:35 INFO manager.MySQLManager: Preparing to use a MySQL  
streaming resultset.

15/01/20 22:47:35 INFO tool.CodeGenTool: Beginning code generation

15/01/20 22:47:36 INFO manager.SqlManager: Executing SQL statement:  
SELECT t.\* FROM `average\_price\_by\_state` AS t LIMIT 1

(output truncated)

15/01/25 22:47:53 INFO mapreduce.ImportJobBase: Transferred 200.4287 KB in  
15.3718 seconds (13.0387 KB/sec)

15/01/25 22:47:53 INFO mapreduce.ImportJobBase: Retrieved 3272 records.

In this particular example, we needed to specify that the import command should use a **single map task**, as our table does not contain a primary key, which is required to split and merge multiple map tasks.

Because we specified that the import task use 1 map task (**-m 1**), we should expect a single file in HDFS:

```
/srv/sqoop$ hadoop fs -head average_price_by_state/part-m-00000 | head
```

| head- to have a quick preview of just first 10 lines alone

```
2012,AK,Total Electric Industry,17.88,14.93,16.82,0.00,null,16.33
2012,AL,Total Electric Industry,11.40,10.63,6.22,0.00,null,9.18
2012,AR,Total Electric Industry,9.30,7.71,5.77,11.23,null,7.62
2012,AZ,Total Electric Industry,11.29,9.53,6.53,0.00,null,9.81
2012,CA,Total Electric Industry,15.34,13.41,10.49,7.17,null,13.53
2012,CO,Total Electric Industry,11.46,9.39,6.95,9.69,null,9.39
2012,CT,Total Electric Industry,17.34,14.65,12.67,9.69,null,15.54
2012,DC,Total Electric Industry,12.28,12.02,5.46,9.01,null,11.85
2012,DE,Total Electric Industry,13.58,10.13,8.36,0.00,null,11.06
2012,FL,Total Electric Industry,11.42,9.66,8.04,8.45,null,10.44
```

We have now successfully imported data from MySQL to HDFS! From here, we can now run any further MapReduce processing on the imported data, or load the data into another Hadoop data source such as Hive, HBase, or HCatalog.

## Importing from MySQL to Hive

Given that our data is already structured in a relational schema (**MySQL**, in this case), it **makes a lot of sense to import that data into a similar schema within Hive, especially if we intend to run relational queries on the data.**

**Sqoop provides a couple ways to do this, either exporting to HDFS first and then loading the data into Hive using the LOAD DATA HQL command in the Hive shell,**

**or by using Sqoop to directly create the tables and load the relational database data into the corresponding tables in Hive**

**Sqoop can generate a Hive table and load data based on the defined schema and table contents from a source database, using the import command.**

**However, because Sqoop still actually utilizes MapReduce to implement the data load operation, we must first delete any preexisting data directory with the same output name before running the import tool:**

```
/srv/sqoop$ hadoop fs -rm -r  
/user/hadoop/average_price_by_state
```

We can then **run Sqoop's import command**, passing it the JDBC connection string to the database, the table name, field delimiter, line terminator, and null string value:

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost:3306/energydata  
--username root --table average_price_by_state  
--hive-import --fields-terminated-by ','  
--lines-terminated-by '\n' --null-string 'null' -m 1
```

(output truncated)

```
.5/01/20 00:14:37 INFO hive.HiveImport: Table default.average_price_by_state stats:  
[numFiles=2, numRows=0, totalSize=205239, rawDataSize=0]  
.5/01/20 00:14:37 INFO hive.HiveImport: OK  
.5/01/20 00:14:37 INFO hive.HiveImport: Time taken: 0.435 seconds  
.5/01/20 00:14:37 INFO hive.HiveImport: Hive import complete.  
.5/01/20 00:14:37 INFO hive.HiveImport: Export directory is empty, removing it.
```

In local mode, **Hive will create a metastore\_db directory** within the file system location from which it was run;

in the previous example, the metastore\_db will be created **under the SQOOP\_HOME (/srv/sqoop)**.

**Open the Hive shell and verify that the table average\_price\_by\_state was created:**

```
/srv/sqoop$ hive
```

```
hive> DESC average_price_by_state;
```

```
OK
```

year	int
state	string
sector	string
residential	double
commercial	double
industrial	double
transportation	double
other	double
total	double

```
Time taken: 0.858 seconds, Fetched: 9 row(s)
```

You can also run a COUNT query to verify that 3,272 rows have been imported;

alternatively, because this dataset is relatively small, you can run a **SELECT \* FROM average\_price\_by\_state** to validate the data.

With our data and schema now imported into Hive, we can continue running any further analysis on the data via the Hive command-line interface or other Hive interface

## Importing from MySQL to Hbase

HBase is designed to handle large volumes of data for a large number of concurrent clients that need real-time access to row-level data.

While relational databases also handle this requirement well in most low-to-modestly high-scale data applications, if the storage requirements of an application start to demand a more scalable solution,

we may consider offloading some of the high-scale and heavy-load data components to a distributed database like Hbase.

**Sqoop's import tool** allows us to import data from a relational database to HBase.

As with Hive, **there are two approaches to importing this data**. We can **import to HDFS first and then use the HBase CLI or API to load the data into an HBase table**, or we can use the --hbase-table option to instruct Sqoop to directly import to a table in HBase.

In this example, the data that we want to offload to HBase is a **table of weblog stats where each record contains a primary key**.

Primary key is composed of the **pipe-delimited IP address and year**, and a **column for each month that contains the number of hits for that IP and year**.

You can find the CSV named weblogs.csv in the GitHub repo's /data directory.

Download this CSV and load it into a MySQL table:

```
~$ mysql -u root -p
```

```
mysql> CREATE DATABASE logdata;
```

```
mysql> GRANT ALL PRIVILEGES ON logdata.* TO '%'@'localhost';
```

```
mysql> GRANT ALL PRIVILEGES ON logdata.* TO '@localhost';
```

```
mysql> USE logdata;
```

```
mysql> CREATE TABLE weblogs (ipyear varchar(255) NOT NULL PRIMARY KEY,  
january int(11) DEFAULT NULL,  
february int(11) DEFAULT NULL,  
march int(11) DEFAULT NULL,  
april int(11) DEFAULT NULL,  
may int(11) DEFAULT NULL,  
june int(11) DEFAULT NULL,  
july int(11) DEFAULT NULL,  
august int(11) DEFAULT NULL,  
september int(11) DEFAULT NULL,  
october int(11) DEFAULT NULL,  
november int(11) DEFAULT NULL,  
december int(11) DEFAULT NULL);
```

```
mysql> quit;
```

```
~$ mysql -u root -p logdata --local-infile=1
```

```
mysql> LOAD DATA LOCAL INFILE '/home/hadoop/hadoop-fundamentals/data/weblogs.csv'  
      INTO TABLE weblogs FIELDS TERMINATED BY ','  
      LINES TERMINATED BY '\n' IGNORE 1 LINES;
```

Query OK, 27300 rows affected (0.20 sec)

Records: 27300 Deleted: 0 Skipped: 0 Warnings: 0

```
mysql> quit;
```

Again, we need to verify that Hadoop is running, as well as HBase daemons:

```
~$ cd $HBASE_HOME  
/srv/hbase$ bin/start-hbase.sh
```

We can then run Squeryl's `import` command, passing it the JDBC connection string to the database, the table name, HBase table name, column family name, and row key name:

```
sqoop import --connect jdbc:mysql://localhost:3306/logdata  
--table weblogs --hbase-table weblogs --column-family traffic  
--hbase-row-key ipyear --hbase-create-table -m 1
```

(output truncated)

```
15/01/20 00:33:01 INFO mapreduce.ImportJobBase: Transferred 0 bytes in  
19.0716 seconds (0 bytes/sec)
```

```
15/01/20 00:33:01 INFO mapreduce.ImportJobBase: Retrieved 27300 records.
```

Once the import MapReduce job has completed, you should see a console message indicating `INFO mapreduce.ImportJobBase: Retrieved 27300 records`. We can verify that the HBase table and rows have been imported successfully in the HBase shell with the `list` and `scan` commands:

```
/srv/sqoop$ cd $HBASE_HOME  
/srv/hbase$ bin/hbase shell
```

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

```
TABLE  
linkshare  
weblogs
```

2 row(s) in 1.2900 seconds

=> ["linkshare", "weblogs"]

hbase(main):002:0> scan 'weblogs', {'LIMIT' => 50}

(output truncated)

We have successfully used Sqoop to import relational data from MySQL to HDFS, Hive, and HBase, using the import tool, which **actually generates a Java class that encapsulates the schema of each row of the imported table.**

This class is **used during the import process by Sqoop itself**, but **can also be used in subsequent MapReduce processing of the data.**

Thus, in addition to automating import/export to and from Hadoop and relational databases,

Sqoop **allows you to quickly develop processing pipelines** across other Hadoop-compatible data sources

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 4

TOPIC: Analytics with Higher-level APIs - Pig

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

## **Analytics with Higher-Level APIs- Pig**

### **Pig**

It is an abstraction of MapReduce, allowing users to express their data processing and analysis operations in a higher-level language that then compiles into a MapReduce job.

Pig was developed at Yahoo as a tool for researchers and engineers to more easily write their data mining Hadoop scripts by representing them as data flows .

Pig is now a top-level Apache Project that includes two main platform components:

**1.Pig Latin**, a procedural scripting language used to express data flows.

**2.The Pig execution environment** to run Pig Latin programs, which can be run in local or MapReduce mode and includes the **Grunt command-line interface**

**Unlike Hive's HQL, which draws heavily from SQL's declarative style,**

**Pig Latin is procedural in nature** and designed to enable programmers to easily implement a series of data operations and transformations that are applied to datasets to form a data pipeline.

While Hive is great for use cases that translate well to SQL-based scripts, SQL can become unwieldy when multiple complex data transformations are required.

**Pig Latin is ideal for implementing these types of multistage data flows**, particularly in cases where we need to aggregate data from multiple sources and perform subsequent transformations at each stage of the data processing flow

**Pig Latin scripts start with data, apply transformations to the data until the script describes the desired results, and execute the entire data processing flow as an optimized MapReduce job.**

**Additionally, Pig supports the ability to integrate custom code with user-defined functions (UDFs) that can be written in Java, Python, or JavaScript, among other supported languages.**

**Pig thus enables us to perform near arbitrary transformations and ad hoc analysis on our big data using comparatively simple constructs.**

**It is important to remember the earlier point that Pig, like Hive, ultimately compiles into MapReduce and cannot transcend the limitations of Hadoop's batch-processing approach.**

However, Pig does provide us with powerful tools to easily and succinctly write complex data processing flows, with the fine-grained controls that we need to build real business applications on Hadoop.

We'll review some of the basic components of Pig and implement both native Pig Latin operators and custom-defined functions to perform some simple sentiment analysis on Twitter data.

We assume that you have installed Pig to run on Hadoop in pseudo-distributed mode.

## Pig Latin

Now that we have Pig and the Grunt shell set up, let's examine a sample Pig script and explore some of the commands and expressions that Pig Latin provides.

The following script loads Twitter tweets with the hashtag #unitedairlines over the course of a single week.

The data file, `united_airlines_tweets.tsv`, provides the tweet ID, permalink, date posted, tweet text, and Twitter username.

The script loads a dictionary, `dictionary.tsv`, of known “positive” and “negative” words along with sentiment scores (1 and -1, respectively) associated to each word.

The script then performs a series of Pig transformations to generate a sentiment score and classification, either POSITIVE or NEGATIVE, for each computed tweet:

```
grunt> tweets = LOAD 'united_airlines_tweets.tsv' USING  
PigStorage('\t')
```

```
AS (id_str:chararray, tweet_url:chararray,  
created_at:chararray, text:chararray, lang:chararray,  
retweet_count:int, favorite_count:int,  
screen_name:chararray);
```

```
grunt> dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t')  
AS (word:chararray, score:int);
```

```
grunt> english_tweets = FILTER tweets BY lang == 'en';
```

```
grunt> tokenized = FOREACH english_tweets GENERATE id_str,  
FLATTEN( TOKENIZE(text) ) AS word;
```

```
grunt> clean_tokens = FOREACH tokenized GENERATE id_str,  
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;
```

```
grunt> token_sentiment = JOIN clean_tokens BY word,  
dictionary BY word;
```

```
grunt> sentiment_group = GROUP token_sentiment BY id_str;
```

```
grunt> sentiment_score = FOREACH sentiment_group  
GENERATE group AS id, SUM(token_sentiment.score) AS final;
```

```
grunt> classified = FOREACH sentiment_score GENERATE id, (final >= 0)? 'POSITIVE' : 'NEGATIVE' ) AS classification, final AS score;
```

```
grunt> final = ORDER classified BY score DESC;
```

```
grunt> STORE final INTO 'sentiment_analysis';
```

Let's break down this script at each step of the data processing flow.

## Relations and tuples

The first two lines in the script loads data from the file system into relations called tweets and dictionary:

```
tweets = LOAD 'united_airlines_tweets.tsv' USING  
PigStorage('\t') AS (id_str:chararray, tweet_url:chararray,  
created_at:chararray, text:chararray, lang:chararray,  
retweet_count:int, favorite_count:int,  
screen_name:chararray);
```

```
dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t') AS  
(word:chararray, score:int);
```

In Pig, a **relation** is conceptually similar to a table in a relational database, but instead of an ordered collection of rows, a relation consists of an **unordered set of tuples**.

Tuples are an ordered set of fields.

It is important to note that although a **relation declaration is on the left side of an assignment**, much like a variable in a typical programming language, relations are not variables.

We used the **LOAD operator** to specify the filename of the file (either on the local file system or HDFS) to load into the tweets and dictionary relations.

We also use the **USING** clause with the PigStorage load function to specify that the file is tab-delimited.

Although not required, we also defined a schema for each relation using the AS clause and specifying column aliases for each field, along with the corresponding data type.

If a schema is not defined, we can still reference the fields for each tuple in our relation by using Pig's positional columns (\$0 for the first field, \$1 for the second, etc.).

This may be preferable if we are loading data with many columns, but are only interested in referencing a few of them

## Filtering

The next line performs a **SIMPLE FILTER** data transformation on the tweets relation to filter out any tuples that are not in English:

**english\_tweets = FILTER tweets BY lang == 'en';**

The FILTER operator selects tuples from a relation **based on some condition**, and is commonly used to select the data that you want;

or, conversely, to filter out (remove) the data you don't want.

Because the “lang” field is typed as a **chararray**, the Pig equivalent of the Java String data type, we used the == comparison operator to retain values that equal en for English.

The result is stored into a new relation called english\_tweets.

## Projection

Now that we've filtered the data to retain only English tweets

We need to **split the tweet text into word tokens**, which we can match against our dictionary, and

perform some **additional data cleanup on the words to remove hashtags, preceded by #, and user handle tags, preceded by @:**

```
tokenized = FOREACH english_tweets GENERATE id_str,  
FLATTEN( TOKENIZE(text) ) AS word;
```

```
clean_tokens = FOREACH tokenized GENERATE id_str,  
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;
```

Pig provides the **FOREACH...GENERATE** operation to work with **columns of data** in relations or collections and apply a set of expressions to every tuple in the collection.

The **GENERATE** clause contains the values and/or evaluation expression that will derive a new collection of tuples to pass onto the next step of the pipeline.

In our example, we project the `id_str` key from the `english_tweets` relation, and use the **TOKENIZE** function to split the **text** field into word tokens (splitting on whitespace).

The **FLATTEN** function extracts the resulting collection of tuples into a single collection.

The collection of tuples we generate is actually a special data type in Pig, called a **bag**, and represents an unordered collection of tuples.

In our FOREACH command, the result produces a new relation called **tokenized** where the first field is the stock\_tweet ID (**id\_str**) and the second field is a bag composed of single-word tuples.

We then perform another projection based on the tokenized relation to project the **id\_str** and lowercased word without any leading hashtag or handle tag.

We've performed quite a few transformations on our data, so it would be a good time to verify that our relations are well structured.

We can use the **ILLUSTRATE** operator at any time to view the schemas of each relation generated based on a concise sample dataset (output truncated due to size):

```
grunt> ILLUSTRATE clean_tokens;
-----
| tweets | id_str:chararray | tweet_url:chararray |
-----
|      | 474415416874250240 | https://.../474415416874250240 |
```

The **ILLUSTRATE** command is helpful to use periodically as we design our Pig flows to help us understand what our queries are doing and validate each checkpoint in the pipeline.

## Grouping and joining

Now that we've tokenized the selected tweets and cleaned the word tokens, we would like to **JOIN the resulting tokens against the dictionary, matching against the word if found:**

**token\_sentiment = JOIN clean\_tokens BY word, dictionary BY word;**

Pig provides the JOIN command to perform a join on two or more relations based on a common field value.

Both inner joins and outer joins are enabled, **although inner joins are used by default.**

In our example, we perform an inner join between the **clean\_tokens** relation and **dictionary** relation based on the **word** field, which will generate a new relation called **token\_sentiment** that contains the fields from both relations:

```
-----  
| token_sentiment | clean_tokens::id_str:chararray |  
clean_tokens::word:chararray |  
dictionary::word:chararray | dictionary::score:int |  
-----  
|               | 473233757961723904 | delayedflight | delayedflight | -1 |  
-----
```

Now we need to GROUP those rows by the Tweet ID, id\_str, so we can later compute an aggregated SUM of the score for each tweet:

**sentiment\_group = GROUP token\_sentiment BY id\_str;**

The GROUP operator groups together tuples that have the same group key (id\_str).

We can now perform the final aggregation of our data, by computing the sum score for each tweet, grouped by ID:

```
sentiment_score = FOREACH sentiment_group GENERATE  
group AS id, SUM(token_sentiment.score) AS final;
```

And then classify each tweet as POSITIVE or NEGATIVE based on the score:

```
classified = FOREACH sentiment_score GENERATE id, ( (final >=  
0)? 'POSITIVE' : 'NEGATIVE' ) AS classification, final AS score;
```

Finally, let's sort the results by score in descending order:

```
final = ORDER classified BY score DESC;
```

We've now defined all the operations and projections needed for our sentiment analysis.

In the next section, we'll save this data to a file on HDFS where we can later view and analyze the results.

## Storing and outputting data

Now that we've applied all the necessary transformations on our data, we would like to write out the results somewhere.

For this purpose, Pig provides the **STORE** statement, which takes a relation and writes the results into the specified location.

By default, the STORE command will write data to HDFS in tabdelimited files using **PigStorage**.

In our example, we dump the results of the final relation into our Hadoop user directory (`/user/hadoop/`) in a folder called **sentiment\_analysis**:

**STORE final INTO 'sentiment\_analysis';**

The contents of that directory will include one or more part files:

```
$ hadoop fs -ls sentiment_analysis
Found 2 items
-rw-r--r-- 1 hadoop supergroup          0 2015-02-19 00:10
sentiment_analysis/_SUCCESS
-rw-r--r-- 1 hadoop supergroup 7492 2015-02-19 00:10
sentiment_analysis/part-r-00000
```

In local mode, only one part file is created, but in MapReduce mode the number of part files depends on the parallelism of the last job before the store.

When working with smaller datasets, it's convenient to quickly output the results from the grunt shell to the screen rather than having to store it.

The DUMP command takes the name of a relation and prints the contents to the console:

**grunt> DUMP sentiment\_analysis;**

The DUMP command is convenient for quickly testing and verifying the output of your Pig script, but generally for large dataset outputs, you will STORE the results to the file system for later analysis.

## Data Types

We covered some of the nested data structures available in Pig, including **fields, tuples, and bags**.

Pig also provides a **map structure**, which contains a set of key/value pairs.

The **key should always be of type chararray**, but the values do not have to be of the same data type.

We saw some of the native scalar types that Pig supports when we defined the schema for the stock data

*Table 8-1. Pig scalar types*

Category	Type	Description	Example
Numeric	int	32-bit signed integer	12
	long	64-bit signed integer	34L
	float	32-bit floating-point number	2.18F
	double	64-bit floating-point number	3e-17
Text	chararray	String or array of characters	hello world
Binary	bytearray	Blob or array of bytes	N/A

## Relational Operators

Pig provides data manipulation commands via the relational operators in Pig Latin.

We used several of these to load, filter, group, project, and store data earlier in our example.

*Table 8-2. Pig relational operators*

Category	Operator	Description
Loading and storing	LOAD	Loads data from the file system or other storage source
	STORE	Saves a relation to the file system or other storage
	DUMP	Prints a relation to the console

## Filtering and projection

### FILTER

Selects tuples from a relation based on some condition

### DISTINCT

Removes duplicate tuples in a relation

### FOREACH... GENERATE

Generates data transformations based on columns of data.

### MAPREDUCE

Executes native MapReduce jobs inside a Pig script

### STREAM

Sends data to an external script or program

### SAMPLE

Selects a random sample of data based on the specified sample size

Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data from two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross-product of two or more relations
Sorting	ORDER	Sorts the relation by one or more fields
	LIMIT	Limits the number of tuples returned from a relation
Combining and splitting	UNION	Computes the union of two or more relations
	SPLIT	Partitions a relation into two or more relations

## User-Defined Functions

One of Pig's most powerful features lies in its ability to let users combine Pig's native relational operators with their own custom processing.

Pig provides extensive support for such **user-defined functions (UDFs)**, and currently provides integration libraries for six languages:

**Java, Jython, Python, JavaScript, Ruby, and Groovy.**

However, **Java is still the most extensively supported language for writing Pig UDFs**, and generally **more efficient**, as it is the same language as Pig and can thus **integrate with Pig interfaces such as the Algebraic Interface and the Accumulator Interface**.

Let's demonstrate a simple UDF for the script we wrote earlier. In this scenario, we would like to write a custom eval UDF that will allow us to convert the score classification evaluation into a function, so that instead of:

```
classified = FOREACH sentiment_score GENERATE id,
( (final >= 0)? 'POSITIVE' : 'NEGATIVE' )
AS classification, final AS score;
```

We can write something like:

```
classified = FOREACH sentiment_score GENERATE id,
classify(final) AS classification, final AS score;
```

In Java, we need to extend Pig's EvalFunc class and implement the exec() method, which takes a tuple and will return a String:

```
package com.statistics.pig;

import java.io.IOException;

import org.apache.pig.EvalFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.Tuple;

public class Classify extends EvalFunc {

    @Override
    public String exec(Tuple input) throws IOException {
        if (args == null || args.size() == 0) {
            return false;
        }
        try {
            Object object = args.get(0);
            if (object == null) {
                return false;
            }
            int i = (Integer) object;
            if (i >= 0) {
                return new String("POSITIVE");
            } else {
                return new String("NEGATIVE");
            }
        } catch (ExecException e) {
            throw new IOException(e);
        }
    }
}
```

To use this function, we need to compile it, package it into a JAR file, and then register the JAR with Pig by using the REGISTER operator:

```
grunt> REGISTER statistics-pig.jar;
```

We can then invoke the function in a command:

```
grunt> classified = FOREACH sentiment_score  
GENERATE id, com.statistics.pig.Classify(final) AS  
classification, final AS score;
```

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 4

TOPIC: Spark's Higher Level API

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

# Spark

**Apache Spark is an open-source, distributed computing system designed for big data processing.**

**It is known for its speed, ease of use, and general-purpose capabilities, particularly in large-scale data analytics tasks.**

**Spark extends the Hadoop ecosystem, but unlike Hadoop's MapReduce, it performs operations in memory, making it faster for certain types of workloads.**

## **Key Features**

**Fast In-Memory Processing:** Performs computations in memory, reducing disk I/O and speeding up data processing.

**Easy-to-Use APIs:** Supports high-level APIs in Python, Scala, Java, and R, with an interactive shell.

**Unified Analytics Engine:** Handles batch processing, real-time streaming, machine learning (MLlib), and graph computation (GraphX).

**Fault Tolerance:** Resilient Distributed Dataset(RDD) ensure data recovery in case of failures.

**Scalable:** Works efficiently on **small to massive clusters**, integrating with HDFS, S3(Amazon's Simple Storage Service), and other storage systems.

**SQL and Structured Data:** Supports SQL queries and structured data processing via Spark SQL and DataFrames.

# Components of Spark:

- **Spark Core:** The foundation that provides basic functionality like **task scheduling, memory management, fault recovery, and data processing using RDDs** (Resilient Distributed Datasets).
- **Spark SQL:** Enables processing of structured data through SQL queries or DataFrames (tables with named columns).
- **Spark Streaming:** For real-time data processing using micro-batches.
- **Mlib:** A library that provides scalable machine learning algorithms.
- **GraphX:** A library for graph computation and analysis.

# Spark's Core Concepts

- **RDD (Resilient Distributed Dataset):** This is the **fundamental data structure** in Spark. RDDs are **immutable, fault-tolerant distributed collections of objects.**
- They allow operations to be performed **in parallel** across a cluster.
  - **Transformations:** Operations like `map()`, `filter()`, and `groupBy()` that are applied to RDDs.
  - They are **lazy**, meaning they don't execute immediately but are instead recorded to be executed when an action is triggered.

## **Spark's Core Concepts continued..**

- **Actions:** Operations like `count()`, `collect()`, and `saveAsTextFile()` that trigger the execution of transformations on an RDD.
- **DataFrames and Datasets:** Spark later introduced these high-level APIs for structured data. They provide optimization and are easier to work with compared to RDDs.
- **DataFrame:** A distributed collection of data organized into named columns, similar to a table in a relational database.
- **Dataset:** A strongly-typed version of DataFrame, which allows you to work with data using domain-specific objects.

Spark, provides **two major programming advantages over the MapReduce-centric Hadoop stack:**

**Built-in expressive APIs** in standard, general-purpose languages like Scala, Java, Python, and R

**A unified programming interface** that includes several built-in higher-level libraries to support a broad range of data processing tasks, including complex interactive analysis, structured querying, stream processing, and machine learning.

## Using Spark's RDD Vs DataFrame

Consider the operation shown in Figure 8-1, which attempts to compute the average age of professors grouped by department.

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	42	B Jones
Chem	61	M Kennedy

RDD API

```
pdata.map(lambda x: (x.dept, [x.age, 1]))\n    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]])\n    .map(lambda x: [x[0], x[1][0] / x[1][1]])\n    .collect()
```

Figure 8-1. Aggregation with Spark's RDD API

In practice, it's much more natural to manipulate structured, tabular data like this using the lingua franca(a bridge b/w different tools) of relational data: SQL

Fortunately, Spark provides an integrated module that allows us to express the preceding aggregation into the simple one-liner shown in Figure 8-2.

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	42	B Jones
Chem	61	M Kennedy

DataFrame API

`data.groupBy("dept").avg("age")`

Figure 8-2. Aggregation with Spark's DataFrames API

# Spark SQL

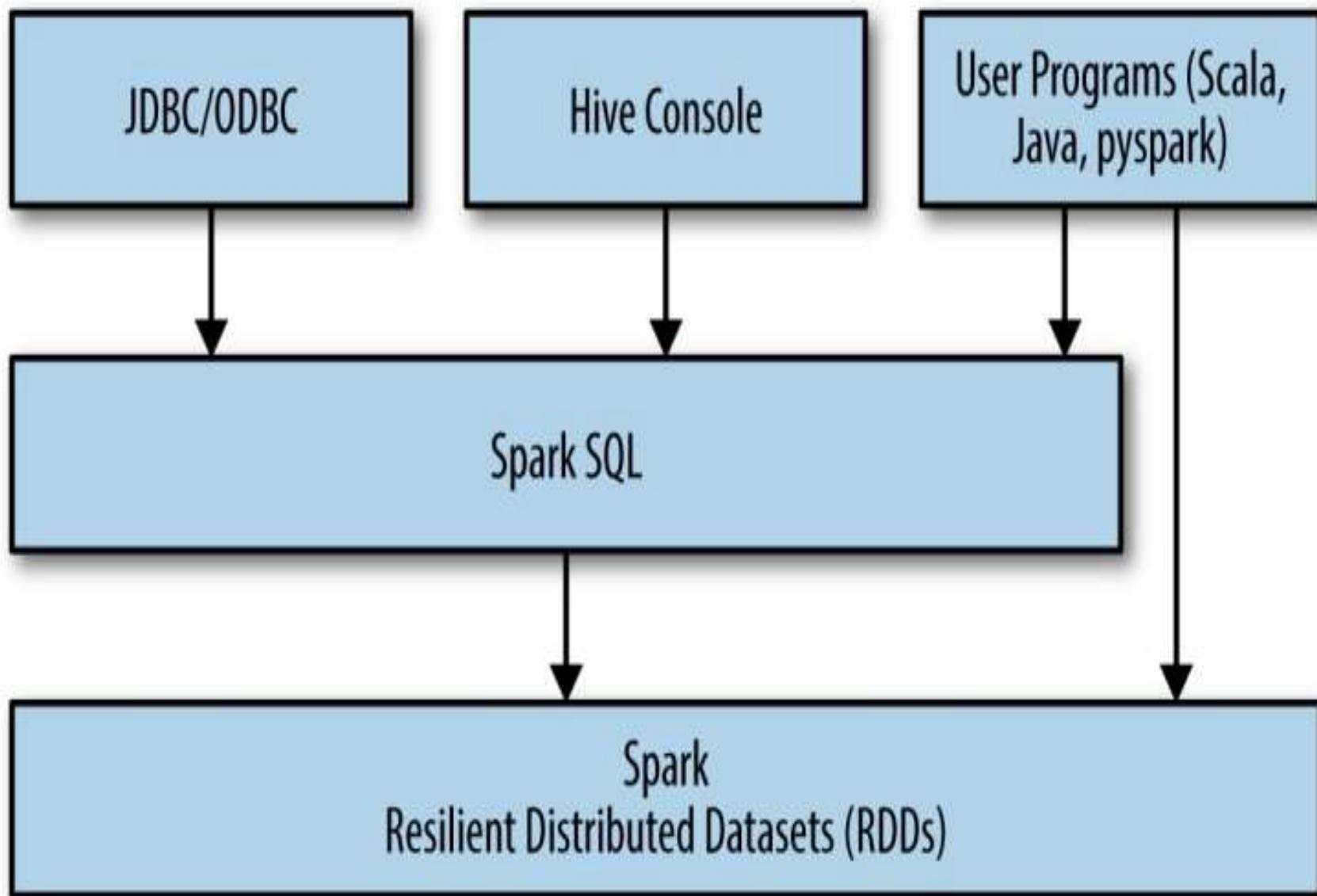
Spark SQL is a module in Apache Spark that provides a **relational interface to work with structured data** using familiar SQL-based operations in Spark.

It can be accessed through :

- JDBC/ODBC connectors,
- a built-in **interactive Hive console**, or
- via its **built-in APIs**.

The last method of access is the **most interesting and powerful aspect of Spark SQL**;

because **Spark SQL actually runs as a library on top of Spark's Core engine and APIs**, we can access the Spark SQL API using the same programming interface that we use for Spark's RDD APIs, as shown in Figure 8-3.



*Figure 8-3. Spark SQL interface*

This allows us to seamlessly combine and leverage the benefits of **relational queries** with the **flexibility of Spark's procedural processing** and the power of **Python's analytic libraries**, all in one programming environment.

Let's **write a simple program** that uses the Spark SQL API to **load JSON(Javascript Object Notation)** data and query it.

You can **enter these commands directly in a running pyspark shell** or in a **Jupyter notebook** that is using a pyspark kernel;

in either case, **ensure that you have a running SparkContext, which we'll assume is referenced by the variable sc**

To begin, we'll need to import the `SQLContext` class from the `pyspark.sql` package.

The `SQLContext` class is the entry point into the Spark SQL API and is created by wrapping an active `SparkContext`:

```
from pyspark.sql  
import SQLContext sqlContext = SQLContext(sc)
```

In this example, we'll load a JSON-formatted dataset from **SF Open Data** that lists publicly available **off-street parking in San Francisco** as of September 2011

With the file properly formatted, we can easily load its contents by calling `sqlContext.read.json` and passing it the path to the file:

```
parking =  
sqlContext.read.json('../data/sf_parking/sf_parking_clean.json')
```

Alternatively, we can pass the `sqlContext` a path to a directory and it will load all files found within into the `parking` object.

Spark SQL automatically infers the schema of a JSON dataset, which we can visualize in a nice tree-format with the `printSchema` method:

```
parking.printSchema()
```

```
root
|-- address: string (nullable = true)
|-- garorlot: string (nullable = true)
|-- landusetyp: string (nullable = true)
|-- location_1: struct (nullable = true)
|   |-- latitude: string (nullable = true)
|   |-- longitude: string (nullable = true)
|   |-- needs_recoding: boolean (nullable = true)
|-- mccap: string (nullable = true)
|-- owner: string (nullable = true)
|-- primetype: string (nullable = true)
|-- regcap: string (nullable = true)
|-- secondtype: string (nullable = true)
|-- valetcap: string (nullable = true)
```

We can also view a sample of the first row of data:

```
parking.first()
```

```
Row(address=u'2110 Market St', garorlot=u'L', landusetyp=u'restaurant',
location_1=Row(latitude=u'37.767378', longitude=u'-122.429344',
needs_recoding=False), mccap=u'0', owner=u'Private', primetype=u'PPA',
regcap=u'13', secondtype=u' ', valetcap=u'0')
```

In order to run a SQL statement against our dataset, we must first

**parking.registerTempTable("parking")**

This allows us to run additional table and SQL methods, including show, which will display the first 20 rows of data in a tabular format:

**parking.show()**

...output truncated...

To execute a SQL statement on the parking table, we use the sql method, passing it the full query.

Let's run an aggregation, grouping the parking by primary and secondary types and getting the count as well as average number of spaces for general parking spaces available.

We'll store it in aggr\_by\_type and call show() to view the full results

```
aggr_by_type = sqlContext.sql("SELECT primetype, secondtype,  
count(1) AS count, round(avg(regcap), 0) AS avg_spaces " +  
"FROM parking " + "GROUP BY primetype, secondtype " +  
"HAVING trim(primetype) != " " + "ORDER BY count DESC")
```

```
aggr_by_type.show()
```

In addition to JSON, Spark SQL supports several other data sources, including files (such as text, parquet, or CSV, which can be parsed with Databricks's CSVreader utility, etc.) from the local file system, HDFS, or S3, JDBC sources like MySQL, and Hive.

Additionally, Spark can even be used as the underlying execution engine in Hive, simply by setting `hive.execution.engine=spark` in your active Hive session.

However, the Spark SQL module is much more than just a SQL interface, and the power of Spark SQL comes down to its underlying data abstraction, the DataFrame.

# DataFrames

DataFrames are the **underlying data abstraction in Spark SQL**.

A **DataFrame** in Apache Spark is a **distributed collection of data organized into named columns**, similar to a table in a relational database or a **data frame** in R or Python's pandas library.

They provide a way to **work with structured and semi-structured data**.

Spark's DataFrames are interoperable with native Pandas (using **pyspark**) and R data frames (using **SparkR**).

The key difference between a Spark DataFrame and a dataframe in Pandas or R is that a **Spark DataFrame is a distributed collection** that actually wraps an RDD; you can think of it as an RDD of row objects

Additionally, **DataFrame** operations entail many optimizations(Filtering the required rows, projecting the required columns , choosing a join strategy)in addition to compile the query plan into executable code, but substantially **improve the performance** and memory-footprint over comparable handcoded RDD operations.

In fact, in a **benchmark test** that compared the runtimes between DataFrames code **that aggregated 10 million integer pairs** against equivalent RDD code,

**DataFrames were not only found to be up to 4–5x faster** for these workloads, but they also **close the performance gap between Python and JVM implementations**, as shown in Figure 8-4

## Not Just Less Code: Faster Implementations

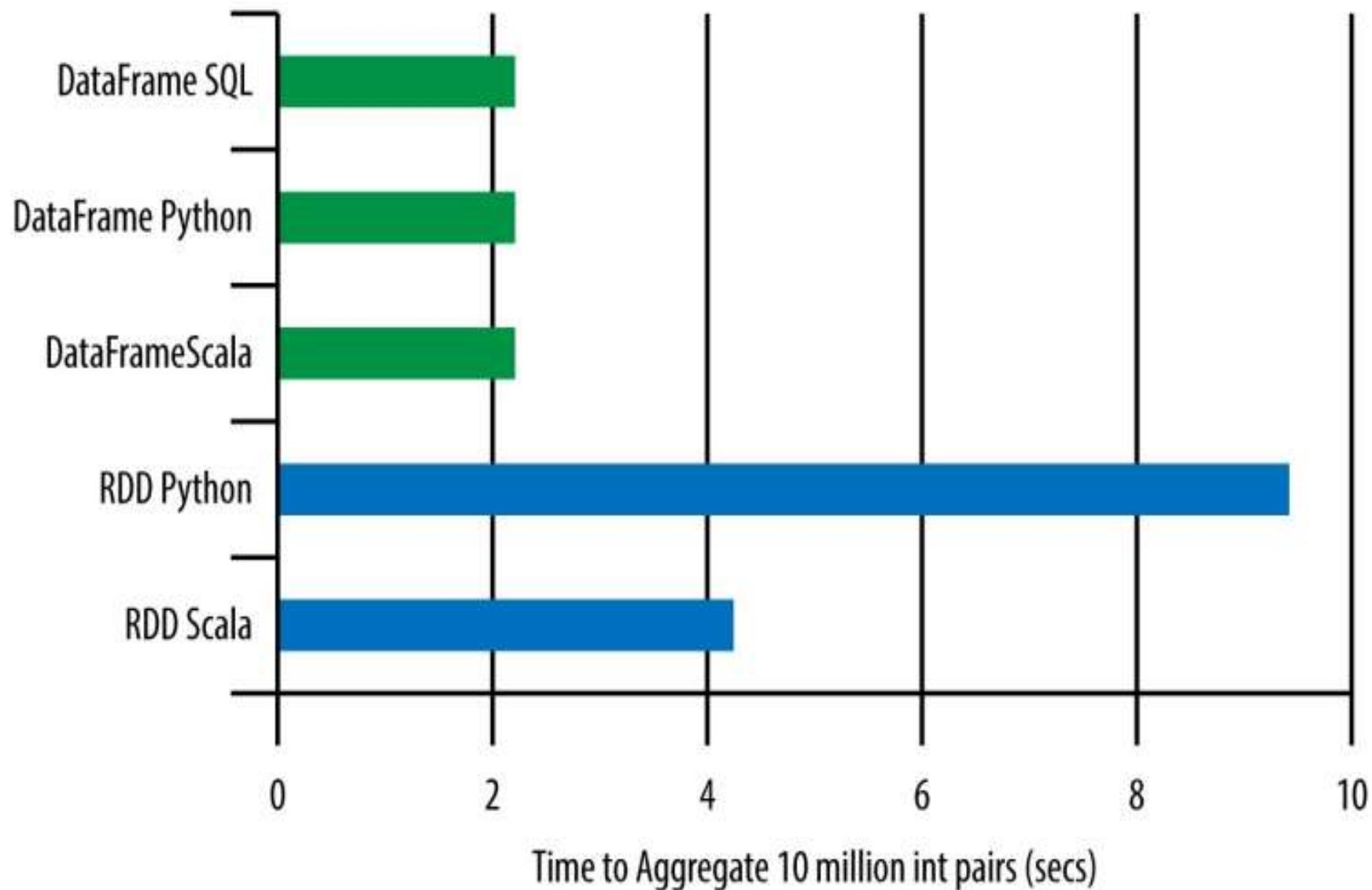


Figure 8-4. DataFrames optimization

**These benefits of the `DataFrames` API coupled with the performance optimizations provided by its computational engine was the reason to make `DataFrames` the main interface for all of Spark's modules, including**

Spark SQL,  
RDDs, MLlib, and  
GraphX.

In this way, the `DataFrames` API provides a unified engine across all of Spark's data sources, workloads, and environments, as shown in Figure 8-5.

Scala

Java

Python

R

DataFrames API

Spark  
SQL

Spark  
Streaming

MLlib

GraphX

RDD API

Spark Core

*Figure 8-5. DataFrames as Spark's unified interface*

**When we loaded the SF parking data in the last example using Spark SQL's read interface, we actually created a DataFrame called parking.**

**While in that example we registered the DataFrame as a temporary table to execute raw SQL queries, there are a multitude of relational operators and window functions that can be called on the parking DataFrame itself.**

**In fact, we can rewrite the SQL query in the previous example by chaining several simple DataFrame operations:**

```
from pyspark.sql import functions as F
```

```
aggr_by_type = (parking
    .select("primetype", "secondtype", "regcap")
    .where("trim(primetype) != ''")
    .groupBy("primetype", "secondtype")
    .agg(
        F.count("*").alias("count"),
        F.round(F.avg("regcap"), 0).alias("avg_spaces")
    )
    .sort("count", ascending=False)
)
```

The advantage of this approach over raw SQL is that we can easily iterate on a complex query by **successively chaining and testing operations**.

Additionally, we have **access to a rich collection of built-in functions from the DataFrames API**, including the **count, round, and avg aggregation functions** that we used previously.

The **pyspark.sql.functions module also contains several mathematical and statistical utilities that include functions for:**

- Random data generation
- Summary and descriptive statistics
- Sample covariance and correlation
- Cross tabulation (a.k.a. contingency table)
- Frequency computation
- Mathematical functions

Let's use one such function to compute some descriptive summary statistics to get a better sense of the distribution and frequency of the parking availability data.

The function `describe` returns a DataFrame containing the count of non-null entries, mean, standard deviation, and minimum and maximum values for each numerical column specified:

```
parking.describe("regcap", "valetcap", "mccap").show()
```

summary	regcap	valetcap	mccap
count	1000	1000	1000
mean	137.294	3.297	0.184
stddev	361.05120902655824	22.624824279398823	1.9015151221485882
min	0	0	0
max	998	96	8

Perhaps we want to determine what the **joint frequency distribution is between the parking owner and the parking's primary type**, or “primetype”.

This is commonly done in statistics by computing a **contingency table** or **crosstabulation** that displays the co-occurrence frequencies between two variables in a matrix format.

We can **easily compute this for a Spark DataFrame by using the crosstab method from the stat interface**:

```
parking.stat.crosstab("owner", "primetype").show()
```

```
parking.stat.crosstab("owner", "primetype").show()
```

owner_primetype	PPA	PHO	CPO	CGO	
Port of SF	7	7	0	4	0
SFPD	0	3	0	6	0
SFMTA	42	14	0	0	0
GG Bridge Authority	2	0	0	0	0
SFSU	2	6	0	0	0
SFRA	2	0	0	0	0

..output truncated..

## Data wrangling DataFrames

While many of the operations and functions in Spark's **DataFrames API** should translate well to Pandas and R users,

there are some important differences due to the immutable and distributed nature of **DataFrames** that Pandas/R programmers should be aware of.

Spark infers data types when loading data, but it defaults to treating ambiguous data as strings.

In Pandas, for example, you can specify data types when reading in data, allowing for more control over how columns are interpreted.

In Pandas, we could easily cast the values in this column by selecting the columns and using astype to cast the values:

`parking['regcap'].astype(int)`

However, because DataFrames are actually just a wrapper for RDDs, which are immutable collections, we need to perform a few steps in order to convert this column into an int type.

In Spark's Dataframes it is a bit complicated.

This workaround involves

**creating a new column based off the existing column,  
casting its values to the correct type, and  
finally dropping the old column.**

In order to retain the column name, we'll first use the `withColumnRenamed` method to rename the existing column to "regcap\_old",

then use the `withColumn` method to add the new column "regcap", which will contain the values of the cast values from `regcap_old`:

```
parking = parking.withColumnRenamed('regcap', 'regcap_old')
```

```
parking = parking.withColumn('regcap',
```

```
    parking['regcap_old'].cast('int'))
```

```
parking = parking.drop('regcap_old')
```

```
parking.printSchema()
```

We'll also want to do this for other numerical columns, so in the spirit of DRY ("don't repeat yourself"), let's define a utility function that will perform this conversion for any arbitrary column and data type:

```
def convert_column(df, col, new_type):
    old_col = '%s_old' % col
    df = df.withColumnRenamed(col, old_col)
    df = df.withColumn(col, df[old_col].cast(new_type))
    df.drop(old_col)
    return df

parking = convert_column(parking, 'valetcap', 'int')
parking = convert_column(parking, 'mccap', 'int')
parking.printSchema()
```

Unfortunately, this function doesn't work with "latitude" and "longitude," because they're actually fields in the "location\_1" struct.

However, we can do even better and **define another function that will take a "location\_1" struct type and use Google's Geocoding API to perform a lookup on the latitude and longitude to return the neighborhood name.**

We'll use the requests library to make request:

```
import requests
```

```
def to_neighborhood(location):
```

```
    name = 'N/A'
```

```
    lat = location.latitude
```

```
    long = location.longitude
```

```
    api_key=google api key
```

```
    # Make the API request
```

```
    r = requests.get(
```

```
f'https://maps.googleapis.com/maps/api/geocode/json?la  
tlng={lat},{long}&key={api_key}'
```

```
)
```

```
if r.status_code == 200:
```

```
    content = r.json() # Parse the JSON response
```

```
# results is a list of matching places
places = content['results']
neighborhoods = [p['formatted_address'] for p in places if
'neighborhood' in p['types']]

if neighborhoods:
    # Addresses are formatted as Japantown, San Francisco, CA
    # so split on comma and just return neighborhood name
    name = neighborhoods[0].split(',')[0]

return name
```

**The `to_neighborhood` function accepts a location struct and returns a string type, but how can we use this function in the context of a column expression?**

The `pyspark.sql.functions` module **provides the `udf` function** to register a user-defined function (UDF).

We declare an inline UDF by passing UDF a callable Python function and the Spark SQL data type that corresponds to the return type;

in this case, we are returning a string so we will use the `StringType` data type from `pyspark.sql.types`. Once registered, we can use the UDF to reformat the “`location_1`” column with a `withColumn` expression:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

location_to_neighborhood=udf(to_neighborhood, StringType())

sfmta_parking = parking.filter(parking.owner == 'SFMTA') \
    .select("location_1", "primetyp", "landusetyp",
    "garorlot", "regcap", "valetcap", "mccap") \
    .withColumn("location_1",
    location_to_neighborhood("location_1")) \
    .sort("regcap", ascending=False)

sfmta_parking.show()
```

location_1	primetype	landusetyp	garorlot	regcap	valetcap	mccap
South of Market	PPA	G	2585	0	47	
N/A	PPA	G	1865	0	0	
Financial District	PPA	G	1095	0	0	
Union Square	PPA	G	985	0	0	

.. output truncated ..

**As you can see, the process of defining and registering a UDF in Spark's DataFrame API is much easier than in Pig or Hive.**

**Once registered, UDFs can be used by other programs on the same Spark cluster and even by BI tools that are connected to Spark SQL via JDBC/ODBC interface.**

**This makes the udf function easily the most powerful function provided by the DataFrames API, as it exposes endless possibilities for applying advanced computations or operations to SQL users.**

# INT 404R01 BIG DATA ANALYTICS

B.Tech CSE 'A'  
Year/Sem: IV/VII

Unit 4

TOPIC: Machine Learning – Scalable ML with Spark

Handled by,

Dr.M.Devi Sri Nandhini

AP III/School of Computing

# Machine Learning

Machine learning computations aim to **derive predictive models from current and historical data.**

The inherent premise is that a **learned algorithm will improve with more training or experience**, and in particular, machine learning algorithms **can achieve extremely effective results for very narrow domains using models trained from large datasets.**

As a result, computations of scale are implicated in most machine learning algorithms.

For this reason, **machine learning computations are well suited to a distributed computing paradigm, like Spark, in order to leverage large training sets to produce meaningful results.**

# Machine Learning

We'll learn about the  
**built-in Spark machine learning library,**

**Spark MLlib**, which consists of many common learning algorithms and utilities, including **classification, regression, clustering, collaborative filtering, dimensionality reduction**.

**It also contains a new “ML-pipeline” framework, spark.ml, which provides a uniform set of high-level APIs that help users create and tune practical machine learning pipelines.**

# Scalable Machine Learning with Spark

Spark is an in-memory distributed computing engine that can run on a Hadoop cluster.

But additionally, the **Spark platform ships with several built-in components** that utilize Spark's processing engine to enable other types of analytical workloads, which benefit from Spark's computational optimizations.

We'll take a **closer look at Spark's built-in machine learning library, MLlib, which includes a suite of common statistical and machine learning algorithms** and utilities, all of which are designed to scale out across a cluster.

Some of you may already be familiar with programming libraries for data mining and machine learning, such as Python's Weka or Scikit-Learn.

These **libraries work well for small to medium-sized datasets** that can be processed on a **single machine**, but **for large datasets that require distributed storage and the power of parallel processing**, we not only need a computational engine that can process a distributed dataset but we also need algorithms that are designed for parallel platforms.

**Spark MLlib only contains parallel algorithms**, in which **operations can be applied in parallel across nodes using Spark's RDD operations.**

Fortunately, there are a number of machine learning techniques and algorithms that are well suited to parallelization.

But it's important to remember that **when using Spark MLlib**, as with the Spark API, **we need to be mindful of creating data (as RDDs) and operating on data in a distributed, parallelizable manner**—

for example, calling `parallelize()` on a small primitive dataset like a Python dictionary or list, so that it can be made available to all nodes in the cluster.

While Spark MLlib includes a number of statistical and machine learning techniques, including sampling, correlation calculation, hypothesis testing, and more, we will specifically focus on MLlib's machine learning algorithms.

This class of algorithms attempts to make predictions or decisions based on training data, often maximizing a mathematical objective about how the algorithm should behave.

Spark MLlib learning algorithms focus on three key areas of machine learning, often referred to as the three Cs of machine learning:

**Collaborative filtering** Also known as **recommender engines**, which produce recommendations based on past behavior, preferences, or similarities to known entities/users

**Classification** Also known as **supervised learning**, which learns from a supervised training set and assigns a category to unclassified items based on that training set

**Clustering** Also known as **unsupervised learning**, which groups data into clusters based on similar characteristics

In general, the **implementation** of these algorithms begins with **defining and extracting a set of features from the data** as numerical representations of features.

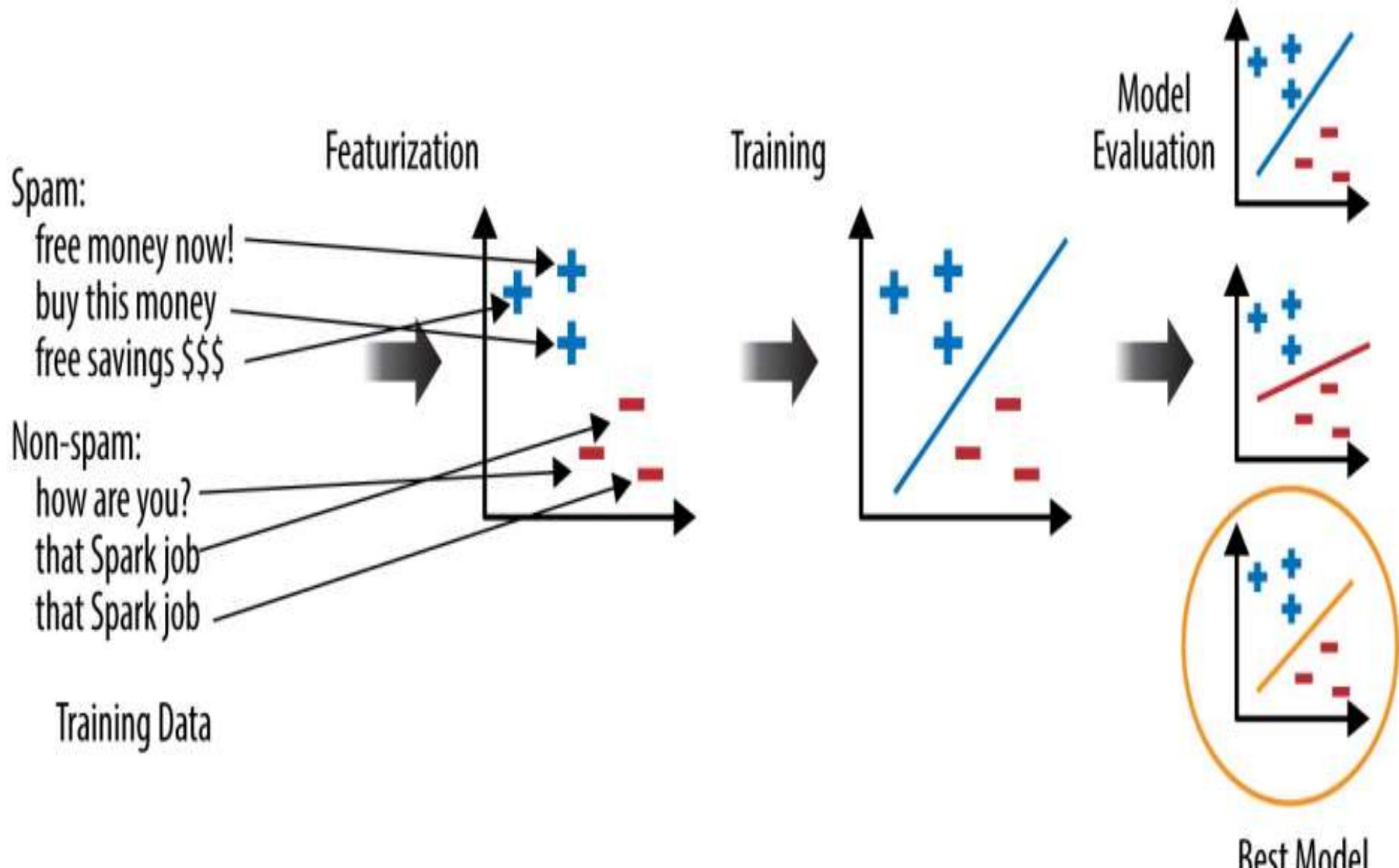
For example, if we were to design a **recommendation system** that **suggests products with similar attributes** (price, color, brand, etc.), we could define **feature vectors** consisting of **weighted number values for each product attribute**.

Or, in the case of **extracting features from unstructured text** (i.e., filtering emails based on **spam detection**), we might represent **each word as a vector of its term frequency-inverse document frequency (TF-IDF)** per classification category (i.e., spam, not spam).

Once we have extracted the feature vectors from the data, we can feed them as training data into a machine learning algorithm that will return a trained model representing the predictions.

When training supervised learning models, we'll generally hold back a segment of the training data as "test data," apply the model to the test data, and quantify the accuracy of the model by comparing the test data predictions to the actual results.

This allows us to assess the accuracy of the model and tune its precision. The high-level stages of the machine learning pipeline are shown in Figure 9-1



*Figure 9-1. The machine learning pipeline*

# **Collaborative Filtering**

Collaborative filtering, or recommendation systems, are perhaps **most commonly recognized in the ecommerce space**, where companies like **Amazon and Netflix** mine user-behavior data such as views, ratings, clicks, and purchases to generate and **suggest other product recommendations**.

Broadly, there are two types of collaborative filtering algorithms:

## **User-based recommenders**

Finds users that are similar to the target user, and uses their collaborative ratings to make recommendations for the target user

## **Item-based recommenders**

Finds and recommends items that are similar or related to items associated with the target user

**MLlib's collaborative filtering library** focuses on **user-based** recommendations, using an implementation of the **Alternating Least Squares (ALS) algorithm**.

MLlib's collaborative filtering approach represents a **user's preferences as a user-item association matrix**, where each dot-product of user and item is a value derived from the preference-score (or rating) multiplied by a weighted factor.

This allows us to accept users' explicit (i.e., positive ratings, purchases) and implicit (i.e., views, clicks) feedback and incorporate them into our model as a combination of binary preferences and confidence values.

The model then tries to find latent factors that can be used to predict the expected preference for an item.

## User-based recommender: An example

Let's use **MLlib's ALS algorithm** to generate recommendations or potential matches for an online dating service.

We'll generate the recommendations for a given user based on a **dataset consisting of profile ratings from an existing dating site**.

In the GitHub repo under the **data/mllib/dating** directory, you'll find two CSVs containing datasets:

user-ratings data containing 1+ million anonymous ratings (**ratings.dat**) for 168,791 user profiles (**gender.dat**).

The data is available from **Occam's Lab**.

MLlib's collaborative filtering library focuses on user-based recommendations, using an implementation of the Alternating Least Squares (ALS) algorithm.

The ratings data is formatted in the following format: **UserID**, **ProfileID**, **Rating**.

UserID is the **user who provided the rating**, ProfileID is the **user who has been rated**, and Rating is a **score on a 1–10 scale where 10 is the highest rating**.

UserIDs range between 1 and 135,359 and ProfileIDs range between 1 and 220,970 (not every profile has been rated).

Only users who have provided atleast 20 ratings were included. Users who provided constant ratings were excluded.

User gender information is in the following format:

**UserID, Gender.**

Gender is denoted by a “M” for male and “F” for female and “U” for unknown.

The full program to the working dating recommender can be found in the GitHub repository under:

**hadoop-  
fundamentals/mllib/collaborative\_filtering/als/matchmaker.py**

This program **can be run on Spark using the spark-submit command and passing it two arguments:**

the **UserID** of the user who we should generate matches for and

the **gender preference (M or F)** for partner matches.

We recommend piping this output to a file, too:

```
$ $SPARK_HOME/bin/spark-submit \
~/hadoop-fundamentals/mllib/collaborative_filtering/als/
matchmaker.py 1 M \
> ~/matchmaking_recs.txt
```

We'll examine each major step of the program.

First, we need to **configure our SparkContext with the name of the application** and

**set the amount of memory to use per executor to 2 GB**, as the ALS algorithm on this amount of data will require a lot of memory:

```
# Configure Spark
conf = SparkConf().setMaster("local") \
    .setAppName("Dating Recommender") \
    .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

```
.setMaster("local") :
```

- This sets the master URL for the Spark application. The value "local" means that Spark will run locally on one node, utilizing the resources on your local machine. It's useful for development and testing, where you don't need a distributed cluster.

```
.setAppName("Dating Recommender") :
```

- This sets the name of the application to "Dating Recommender". This is a user-defined name that will appear in the Spark UI and logs, helping to identify the job.

```
.set("spark.executor.memory", "2g") :
```

- This sets the amount of memory allocated for each executor in the Spark cluster. Here, each executor is assigned 2 GB of memory. Executors are the distributed agents responsible for running tasks on different parts of the dataset.

```
sc = SparkContext(conf=conf) :
```

- This creates a `SparkContext` object `sc`, which is the entry point to using Spark functionality. The `sparkContext` allows you to interact with the Spark cluster, manage RDDs (Resilient Distributed Datasets), and perform distributed computations.

This function, `parse_rating`, is designed to parse a line of text representing a rating, typically from a CSV file, and return the parsed information in a specific format.

Next, we'll read the argument for the User ID as well as the user's gender preference for matches, and call a custom-defined `parse_rating` method against each record in the ratings file:

```
def parse_rating(line, sep=','):
    """
    Parses a rating line
    Returns: tuple of (random integer, (user_id, profile_id, rating))
    """
    fields = line.strip().split(sep)
```

```
user_id = int(fields[0])      # convert user_id to int
profile_id = int(fields[1])    # convert profile_id to int
rating = float(fields[2])     # convert rated_id to int
return random.randint(1, 10), (user_id, profile_id, rating)
```

Given a rating row, the `parse_rating` method returns a tuple where the first item is a random integer and the second item is another tuple of (`user_id`, `profile_id`, `rating`):

- If the input line is "123,456,4.5", the output of the function might look something like this:
- python
- Copy code
- (7, (123, 456, 4.5))
- Here, 7 is a random integer between 1 and 10, and (123, 456, 4.5) is the tuple of parsed values (user ID, profile ID, rating).

```
matchseeker = int(sys.argv[1])
gender_filter = sys.argv[2]

# Create ratings RDD of (randint, (user_id, profile_id, rating))
ratings = sc.textFile(
    "/home/hadoop/hadoop-fundamentals/data/dating/ratings.dat")\
    .map(parse_rating)
```

- **.map(parse\_rating):**

The .map() function applies the parse\_rating function to each line of the file. This transforms each line into a tuple consisting of:

- A random integer (for partitioning or sampling),
- A tuple containing user\_id, profile\_id, and rating from the ratings file.

Next, we read the user profile data from gender.dat by mapping the customdefined parse\_user method to each row of the file:

```
def parse_user(line, sep=','):
    """
    Parses a user line
    Returns: tuple of (user_id, gender)
    """

    fields = line.strip().split(sep)
    user_id = int(fields[0]) # convert user_id to int
    gender = fields[1]
    return user_id, gender
```

Given a user row, the parse\_user method returns a tuple of (user\_id, gender).

Once we've generated the RDD of user tuples, we'll call collect() to convert the RDD to a list:

```
# Create users RDD  
users = dict(sc.textFile(  
    "/home/hadoop/hadoop-fundamentals/data/dating/gender.dat")\\  
    .map(parse_user).collect())
```

### .collect():

The .collect() method takes the elements from the RDD and brings them into a Python list. It gathers all the tuples from the distributed environment back into the local environment.

### dict():

The dict() function converts the list of tuples returned by .collect() into a dictionary. In this dictionary:

The user\_id will be the **key**.

The gender will be the **value**.

For example, if gender.dat contains:

123,male

456,female

789,male

The result of this would be a dictionary like:

{123: 'male', 456: 'female', 789: 'male'}

Now let's split our ratings data into a training set that we'll use to train the model, and a validation set that we'll use to evaluate our model.

We'll try to reserve 60% of the data for training and 40% for validation, by filtering on the random integer key we added to each tuple.

We'll increase the parallelism of these RDDs by setting the number of partitions to 4 (or whatever number of cores your machine supports), and caching the result:

```
# Create the training (60%) and validation (40%) set, based on last digit  
# of timestamp  
num_partitions = 4  
training = ratings.filter(lambda x: x[0] < 6) \  
    .values() \  
    .repartition(num_partitions) \  
    .cache()  
  
validation = ratings.filter(lambda x: x[0] >= 6) \  
    .values() \  
    .repartition(num_partitions) \  
    .cache()  
  
num_training = training.count()  
num_validation = validation.count()  
  
print "Training: %d and validation: %d\n" % (num_training, num_validation)
```

ALS provides us with the following training parameters that we can set and adjust to tune the model:

rank

Size of feature vectors to use, where size is determined by the number of latent factors; larger ranks can lead to better models, but are more expensive to compute (default: 10)

num\_iterations

Number of iterations to run (default: 10)

lambda

Regularization parameter (default: 0.01)

## alpha

A constant used for computing confidence in implicit ALS (default: 1.0)

Because we are just capturing explicit ratings here, we will ignore alpha and use the default value.

For the other parameters, we'll use a rank of 8, and set the number of iterations to 8, and a lambda of 0.1.

These initial training parameters are somewhat arbitrary, considering we don't know enough about the data to determine the number of latent factors or appropriate regularization value;

however, we can start with this combination and later evaluate the results against other models with different combinations of training parameters to determine the best-fitting model:

```
# rank is the number of latent factors in the model  
# num_iterations is the number of iterations to run.  
# lambda specifies the regularization parameter in ALS  
rank = 8  
num_iterations = 8  
lambda = 0.1
```

We can now create the model using the `ALS.train()` method, which accepts the training RDD of ratings tuples and our training parameters:

```
# Train model with training data and configured rank and iterations
model = ALS.train(training, rank, num_iterations, lambda)

# evaluate the trained model on the validation set
print "The model was trained with rank = %d, lambda = %.1f, and %d iterations.
\n" % \
    (rank, lambda, num_iterations)
```

Once the model is created, we'll use the root mean squared error (RMSE) to compute the error of each model.

The RMSE is the square root of the average value of  $(\text{actual rating} - \text{predicted rating})^2$  for all users that have an actual rating.

$$RMS = \left( \frac{1}{n} \sum_{i=1}^n (\text{model}_i - \text{observed}_i)^2 \right)^{\frac{1}{2}}$$

In our recommender program, we can implement the RMSE computation accordingly:

```
def compute_rmse(model, data, n):
    """
    Compute Root Mean Squared Error (RMSE), or square root of the average value
    of (actual rating - predicted rating)^2
    """
    predictions = model.predictAll(data.map(lambda x: (x[0], x[1])))
    predictions_ratings = predictions.map(lambda x: ((x[0], x[1]), x[2])) \
        .join(data.map(lambda x: ((x[0], x[1]), x[2]))) \
        .values()
    return sqrt(predictions_ratings.map(lambda x: (x[0] - x[1]) ** 2). \
    reduce(add) / float(n))
```

The RMSE indicates the absolute fit of the model to the data (how close the observed data points are to the model's predicted values) and has the useful property of being in the same units as the rating value.

Lower values of RMSE indicate better fit, but because it's relative to the rating value, we should evaluate it on a scale of 1–10.

Depending on the result, we may decide to tune our model by adjusting the training parameters or by providing more or better training data:

```
# Print RMSE of model
validation_rmse = compute_rmse(model, validation, num_validation)

print "The model was trained with rank=%d, lambda=%.1f, and %d iterations." % \
(rank, lambda, num_iterations)
print "Its RMSE on the validation set is %f.\n" % validation_rmse
```

Assuming that we are fine with our model's fit as indicated by the RMSE value, we can now apply it to generate recommendations for the given user.

We'll first generate a set of eligible users by filtering on the given user's preferred gender.

This will form our recommendation candidates, RDD:

```
# Filter on preferred gender
partners = sc.parallelize([u[0] for u in filter(lambda u: u[1] ==
gender_filter, users.items())])
```

Now we'll use the model's predictAll() method, passing it a pair RDD with key=user\_id where the user\_id is the given matchseeker.

This allows the model to generate its recommendations. We'll collect the results into a list, and sort them by reverse rating value, taking the top 10 recommended users:

```
# run predictions with trained model
predictions = model.predictAll(partners.map(lambda x: (matchseeker, x))). \
.collect()

# sort the recommendations
recommendations = sorted(predictions, key=lambda x: x[2], reverse=True)[:10]
```

Finally, we'll print the full list of recommendations and stop the SparkContext:

```
print "Eligible partners recommended for User ID: %d" % matchseeker
for i in xrange(len(recommendations)):
    print ("%2d: %s" % (i + 1, recommendations[i][1])).encode('ascii', 'ignore')

# clean up
sc.stop()
```

If you submitted this job to Spark using this command to save the output to a results file, you should see output similar to the following:

```
$ cat matchmaking_recs.txt
```

Training: 542953 and validation: 542279

The model was trained with rank = 8, lambda = 0.1, and 8 iterations.

Its RMSE on the validation set is 3.580347.

Eligible partners recommended for User ID: 1

1: 100939	
2: 70020	8: 51378
3: 109013	9: 8849
4: 54998	10: 118595
5: 132170	
6: 3843	
7: 170778	

Computing the RMSE gives us a useful metric to evaluate the performance of our model, but with collaborative filtering models, as with most ML-algorithms, the model will perform better with more data and iterations.

For ALS, it's recommended to try a combination of rank, iterations, and regularization (lambda) parameters and compare their respective RMSEs to find the best fit. :

# Classification

Classification attempts to categorize data, usually text or documents, based on supervised training methods that utilize **annotated training sets** to discover patterns that will allow the machine learner to quickly label new records.

For example, a simple classification algorithm might keep track of the features and words associated with a category, as well as the number of times those words are seen for a given category.

Once the machine learner has extracted the features from the training data, it can generate a feature vector and apply a statistical model to build a predictive model, which can then be applied to new data.

MLlib provides several algorithms for **binary** and **multiclass classification**, as well as algorithms for regression analysis.

In **binary classification**, we want to classify entities into one of two distinct categories or labels (e.g., determining whether or not emails are spam).

In **multiclass classification**, we want to classify entities into one of more than two categories (e.g., determining what category a news article appears to most belong to).

The **goal of regression analysis** algorithms is to estimate the relationships and dependencies between a dependent variable (e.g., physical activity level) and one or more independent variables (e.g., risk of heart disease) as a continuous function.

In each of these types of algorithms, the **MLlib implementation involves applying the algorithm on a set of labeled examples.**

These are represented as **LabeledPoint** objects, which include a **numerical value (for binary classification)** or **feature vector (for multiclass)** along with the category label.

The training data of already categorized **LabeledPoints** are **used to train the model**, which can then be used to predict the category for new entities.

In this section, we'll create a simple binary classifier by applying a logistic regression procedure using stochastic gradient descent, also known as `LogisticRegressionWithSGD`.

## Logistic regression classification:

An example In this example, we'll build a simple spam classifier that we'll train with email data that we've categorized as spam and not spam (or ham).

Our spam classifier will utilize two MLlib algorithms, **HashingTF**, which we'll use to extract the feature vectors as term frequency vectors from the training text, and **LogisticRegressionWithSGD**, which implements a logistic regression using stochastic gradient descent.

The training data, spam.txt and ham.txt, can be found within the GitHub repo's /data directory as spam\_classifier.zip. This data is a subset of the SpamAssassin public corpus.

The full spam classifier program can be found under the mllib/classification directory; you can run the program using the command:

```
$ $SPARK_HOME/bin/spark-submit \
/home/hadoop/hadoop-fundamentals/mllib/classification/spam_classifier.py \
/home/hadoop/hadoop-fundamentals/data/spam_classifier/spam.txt \
/home/hadoop/hadoop-fundamentals/data/spam_classifier/ham.txt
```

We'll examine each of the major steps.

We first configure our SparkContext, again setting the application name and increasing the executor memory to 2 GB:

```
# Configure Spark
conf = SparkConf().setMaster("local") \
    .setAppName("Spam Classifier") \
    .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

Next, we'll read the command-line arguments to get the paths to the spam and ham training data files. We'll read those in to create the spam and ham RDDs:

```
spam_file = sys.argv[1]
ham_file = sys.argv[2]

spam = sc.textFile(spam_file)
ham = sc.textFile(ham_file)
```

Now we'll instantiate the HashingTF object, setting the number of features to extract at 10,000:

```
tf = HashingTF(numFeatures=10000)
```

We'll apply HashingTF's `transform()` method to our spam and ham data, first splitting the contents into word tokens. This will extract the term frequency vectors from the spam and ham RDDs, and project them as new RDDs of feature vectors:

```
spam_features = spam.map(lambda email: tf.transform(email. \
    split(" ")))  
ham_features = ham.map(lambda email: tf.transform(email. \
    split(" ")))
```

We'll now convert each feature vector in our RDDs into a `LabeledPoint`. This is a binary classifier, so we'll represent spam as 1 and ham as a 0. The second value in the `LabeledPoint` object will consist of the feature. We'll take the union of these RDDs as our training dataset, and cache it because logistic regression is an iterative algorithm:

```
positive_examples = spam_features.map(lambda features: LabeledPoint(1, features))
negative_examples = ham_features.map(lambda features: LabeledPoint(0, features))
training = positive_examples.union(negative_examples)
training.cache()
```

Now we'll run the logistic regression using the SGD algorithm and our training data:

```
model = LogisticRegressionWithSGD.train(training)
```

We can now create test data, consisting of text content that should be classified as positive for spam, as well as content that should be classified as negative (ham).

We'll use our trained model to predict whether the test data is considered spam or ham.

Recall from our discussion of LabeledPoints that 1 is considered spam, and 0 is considered ham:

```
# Create test data and test the model
positive_test = tf.transform("Guaranteed to Lose 20 lbs in 10 days
Try FREE!".split(" "))
negative_test = tf.transform("Hi, Mom, I'm learning all about Hadoop
and Spark!".split(" "))

print "Prediction for positive test example: %g" % model.predict(positive_test)
print "Prediction for negative test example: %g" % model.predict(negative_test)
```

From here, we can evaluate the accuracy of our classifier model by comparing the predicted results against a holdout of categorized data, or apply the model on an unlabeled set of data.

MLlib's classification algorithms are optimized for large sets of supervised training data, so in general, more data will yield better results than small, but higher-precision data.

However, it's still important to consider the data and apply the most appropriate algorithm and evaluation methods.

For the full list of supported classification algorithms and evaluation metrics, refer to the official Spark MLlib documentation and pay particular attention to what APIs (Scala, Java, Python) are supported for each. :

# Clustering

Unlike collaborative filtering and classification algorithms, **clustering utilizes unsupervised learning techniques to build a model.**

Clustering algorithms attempt to **organize a collection of data into groups of similar items.**

Examples of clustering might include finding groups of customers with similar characteristics or interests, or grouping animals/plants into common species.

The goal of clustering is to **partition data into a number of clusters such that the data within each cluster is more similar to each other than to data in other clusters.**

Spark MLlib offers a handful of popular clustering models,  
but perhaps the simplest and **most popular clustering algorithm included is k-means.**

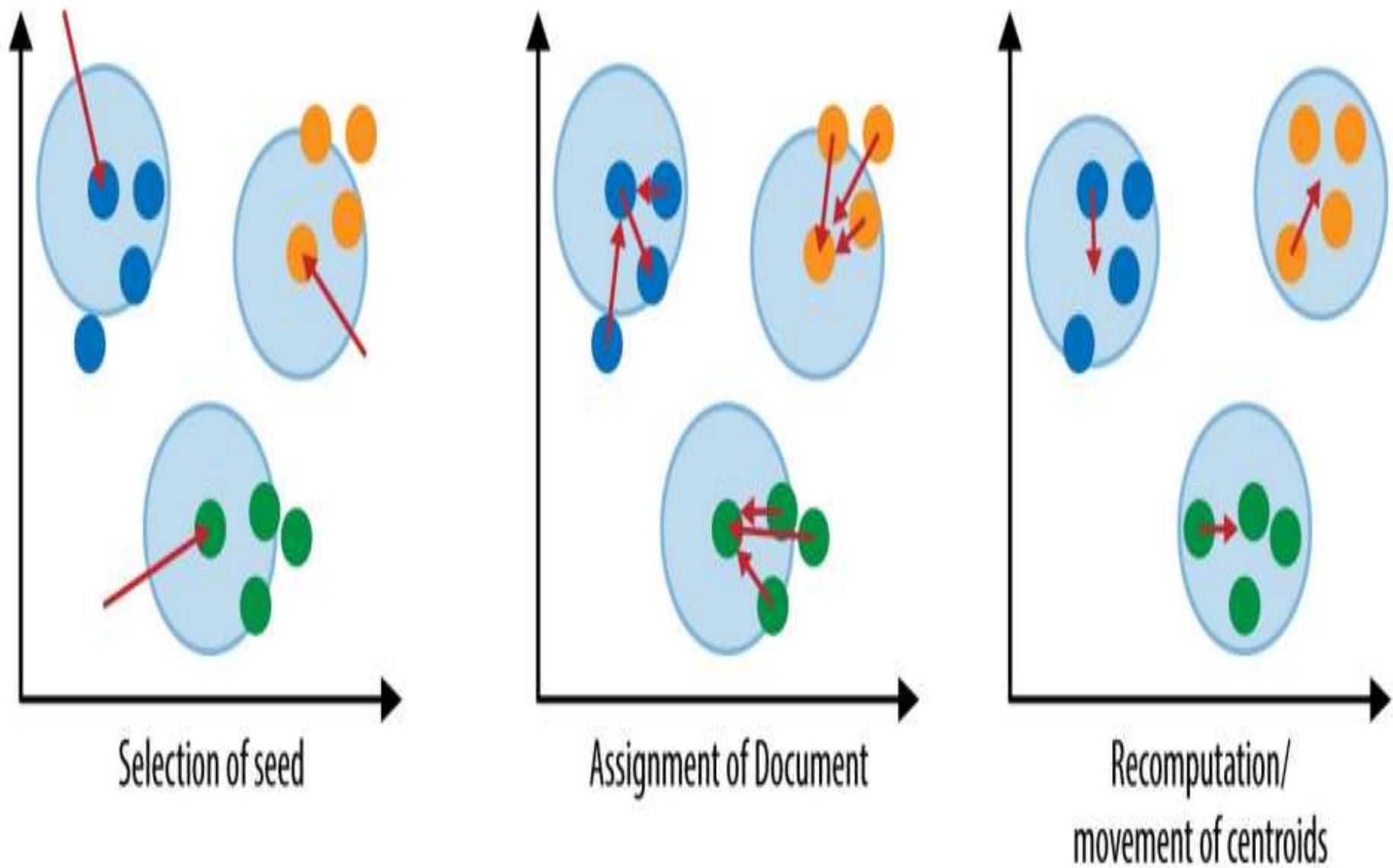
The kmeans algorithm requires that **we represent all objects as a set of numerical features**, and that we specify the target number of clusters (k clusters) we want up front.

MLlib's implementation of k-means clustering starts again with **vectorizing the dataset**, and representing **each object within a feature vector in n-dimensional space**, where n is the number of all features used to describe the objects to be clustered.

The algorithm randomly chooses k points in that vector space, which serve as the initial centers, or centroids, of the clusters

The algorithm then **assigns each object to the centroid that it's closest to**, recalculating the centroid point using the average of the coordinates of all the points in the cluster and reassigning objects to their closest cluster, as necessary.

The process of assigning objects and re-computing centers is repeated until the process converges, as shown in Figure 9-2.



*Figure 9-2. Computational stages of k-means clustering algorithm*

The most important issue in clustering is to determine how to quantify the similarity of the objects being clustered.

The weighting method may be derived from TF-IDF (term frequency-inverse document frequency), which is particularly useful for text-documents,

or it may be determined by a function of other custom properties in our data (i.e., segmenting customers based on total purchase amount in dollars) based on average engagement as measured by some computed metric.

**For MLlib's k-means clustering input, we need to express whatever weighting method we use as a feature vector.**

For example, if we determined that we want to cluster all customers by three features:

- total purchase amount,
- average purchase frequency, and
- average per/purchase amount,

then a sample of our customers might be represented as shown in Table 9-1

*Table 9-1. Customer feature vectorization*

Name	Total purchase amount (in \$)	Average # purchases per-month	Average per-purchase amount	Feature vector
Jane	825	5	115	[825,5,115]
Bob	201	1	45	[201,1,45]
Emma	649	2	65	[649,2,65]

With multiple features, we must be mindful of dimension values that are expressed in different units, or are not normalized with respect to each other.

If we applied a simple distance-based metric to determine similarity between these vectors, total purchase amounts would dominate the results.

Weighting the different dimensions solves this problem. k-means clustering:

An example In this example, we'll apply the k-means clustering algorithm to determine which areas in the United States have been most hit by earthquakes so far this year.

This information can be found within the GitHub repo's /data directory, as **earthquakes.csv**. The **columns for this CSV file are as follows:**

- time
- latitude
- longitude
- depth
- magnitude
- magnitudeType
- nst
- gap
- dmin
- rms
- net
- id
- updated
- place

We'll extract the latitude and longitude from these records, and use that as the input for training our model.

In this iteration, we'll attempt **to generate 6 clusters**. The full program can be run using the command:

```
$ $SPARK_HOME/bin/spark-submit \
/home/hadoop/hadoop-fundamentals/mllib/clustering/earthquakes_clustering.py \
/home/hadoop/hadoop-fundamentals/data/earthquakes.csv \
6 > clusters.txt
```

First, we'll configure Spark and create our SparkContext:

```
# Configure Spark
conf = SparkConf().setMaster("local") \
    .setAppName("Earthquake Clustering") \
    .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

Next, we'll create our training RDD from the earthquakes file, parsing the latitude and longitude from each line and converting it into a NumPy array:

```
# Create training RDD of (lat, long) vectors
earthquakes_file = sys.argv[1]
training = sc.textFile(earthquakes_file).map(parse_vector)
```

We'll set  $k$ -clusters based on the second argument passed—in this case, 6:

```
k = int(sys.argv[2])
```

Now we can call `KMeans.train()` and pass it our training set and  $k$  (set to 6). This will generate the model and allow us to access the cluster centers:

```
# train model based on training data and k-clusters
model = KMeans.train(training, k)

print "Earthquake cluster centers: " + str(model.clusterCenters)
sc.stop()
```

If you inspect the output, *clusters.txt*, you should see output similar to the following:

```
Earthquake cluster centers: [array([ 38.63343185, -119.22434212]),  
 array([ 13.9684592 , 142.97677391]),  
 array([ 61.00245376, -152.27632577]),  
 array([ 35.74366346, 27.33590769]),  
 array([ 10.8458037, -158.656725 ]),  
 array([ 23.48432962, -82.3864285 ])]
```

From here, we can plot the resulting output against the training data to perform an “eyeball” evaluation of the results, and tune the number of clusters ( $k$ ) and number of iterations to adjust the cluster centers.

For a more precise evaluation metric, we could also compute the “Within Set Sum of Squared Errors”, which measures the compactness of cluster points around each center point:

```
def error(point):
    center = model.centers[model.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

WSSSE = training.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE))
```