BIG DATA ANALYTICS

MODULE 1

INTRODUCTION TO BIG DATA AND HADOOP

Motivation:

To provide the students with the knowledge of

Learning Objectives:

- Analyze the growing level of data
- Demonstrate the characteristics and types of big data

Syllabus:

Module	Syllabus	Duration
1	Introduction to Big Data, Big Data characteristics, types of Big Data, Traditional vs. Big Data business approach, Case Study of Big Data Solutions. Concept of Hadoop, Core Hadoop Components; Hadoop Ecosystem	06

Learning Outcomes:

After studying this chapter, students able to:

- Analyze why traditional data storage cannot be used to store big data
- Learn the availability of modern technologies developed to handle big data
- Implement big data solution application for different case studies

Defining Big Data - Big data typically refers to the following types of data:

- Traditional enterprise data – includes customer information from CRM systems, transactional ERP data, web store transactions, and general ledger data.
- Machine-generated /sensor data – includes Call Detail Records ("CDR"), weblogs, smart meters, manufacturing sensors, equipment logs (often referred to as digital exhaust), trading systems data.
- Social data – includes customer feedback streams, micro-blogging sites like Twitter, social media platforms like Facebook

BIG DATA CHARACTERISTICS

The McKinsey Global Institute estimates that data volume is growing 40% per year, and will grow 44x between 2009 and 2020. But while it's often the most visible parameter, volume of data is not the only characteristic that matters. In fact, there are four key characteristics that define big data:

- **Volume.** Machine-generated data is produced in much larger quantities than non-traditional data. For instance, a single jet engine can generate 10TB of data in 30 minutes. With more than 25,000 airline flights per day, the daily volume of just this single data source runs into the Petabytes. Smart meters and heavy industrial equipment like oil refineries and drilling rigs generate similar data volumes, compounding the problem.
- **Velocity**. Social media data streams – while not as massive as machine-generated data – produce a large influx of opinions and relationships valuable to customer relationship management. Even at 140 characters per tweet, the high velocity (or frequency) of Twitter data ensures large volumes (over 8 TB per day).
- **Variety.** Traditional data formats tend to be relatively well defined by a data schema and change slowly. In contrast, non-traditional data formats exhibit a dizzying rate of change. As new services are added, new sensors deployed, or new marketing campaigns executed, new data types are needed to capture the resultant information.
- **Value.** The economic value of different data varies significantly. Typically there is good information hidden amongst a larger body of non-traditional data; the challenge is identifying what is valuable and then transforming and extracting that data for analysis.

To make the most of big data, enterprises must evolve their IT infrastructures to handle these new high-volume, high-velocity, high-variety sources of data and integrate them with the pre-existing enterprise data to be analyzed.

TYPES OF BIG DATA

Big data can be grouped into two broad categories: **structured and unstructured**.

Structured Data

1. Created

Created data is just that; data businesses purposely create, generally for market research. This may consist of customer surveys or focus groups. It also includes more modern methods of research, such as creating a loyalty program that collects consumer information or asking users to create an account and login while they are shopping online.

2. Provoked
A Forbes Article defined provoked data as, "Giving people the opportunity to express their views." Every time a customer rates a restaurant, an employee, a purchasing experience or a product they are creating provoked data. Rating sites, such as Yelp, also generate this type of data.

3. Transacted
Transactional data is also fairly self-explanatory. Businesses collect data on every transaction completed, whether the purchase is completed through an online shopping cart or in-store at the cash register. Businesses also collect data on the steps that lead to a purchase online. For example, a customer may click on a banner ad that leads them to the product pages which then spurs a purchase.
As explained by the Forbes article, "Transacted data is a powerful way to understand exactly what was bought, where it was bought, and when. Matching this type of data with other information, such as weather, can yield even more insights. (We know that people buy more Pop-Tarts at Walmart when a storm is predicted.)"

4. Compiled
Compiled data is giant databases of data collected on every U.S. household. Companies like Acxiom collect information on things like credit scores, location, demographics, purchases and registered cars that marketing companies can then access for supplemental consumer data.

5. Experimental
Experimental data is created when businesses experiment with different marketing pieces and messages to see which are most effective with consumers. You can also look at experimental data as a combination of created and transactional data.

Unstructured Data
People in the business world are generally very familiar with the types of structured data mentioned above. However, unstructured is a little less familiar not because there's less of it, but before technologies like NoSQL and Hadoop came along, harnessing unstructured data wasn't possible. In fact, most data being created today is unstructured. Unstructured data, as the name suggests, lacks structure. It can't be gathered based on clicks, purchases or a barcode, so what is it exactly?

6. Captured
Captured data is created passively due to a person's behavior. Every time someone enters a search term on Google that is data that can be captured for future benefit. The GPS info on our smartphones is another example of passive data that can be captured with big data technologies.

7. User-generated
User-generated data consists of all of the data individuals are putting on the Internet every day. From tweets, to Facebook posts, to comments on news stories, to videos put up on YouTube, individuals are creating a huge amount of data that businesses can use to better target consumers and get feedback on products.

Big data is made up of many different types of data. The seven listed above comprise types of external data included in the big data spectrum.

TRADITIONAL VS. BIG DATA BUSINESS APPROACH

The traditional business intelligence (BI) is shaped like a pyramid (Dyche, 2007): from the standard report at the bottom to the multidimensional report, the segmentation/predictive modeling, and finally to knowledge discovery, which is at the top of the pyramid. Going from collecting a standard report to knowledge discovery, data maturity of the organization increases and there are fewer assumptions needed. This is similar to the capability maturity model in software development (Paulk, Curtis, et al. 1993).

The BI pyramid defines a sequence of efforts from simple to increasingly complex, as in crawl, walk, and run. Most organizations are somewhere in the middle in "maturity" level; they never go beyond the stage of multidimensional reporting or simple analysis. These companies may just have built a data collection infrastructure, or may not have the required analytic talents, or may not be ready due to organizational and cultural reasons to achieve a higher level on the pyramid. They never had a detailed analysis of the data; no predictive modeling was ever done. Again and again in our years of experience, we found data issues that are subtle enough to look normal without a detailed analysis. For example, a data warehouse may take many data feeds from different departments or regions, and only one of them has problems. The numbers are not missing, but they are not accurate or not correct.
If a company adopts a stepwise approach according to the traditional BI pyramid, business rules used to produce standard reporting will need to be decided beforehand. Before big data technology is available, because of the high cost of storage and computing power to process, most data are not collected or discarded. Only data deemed to be the most important are kept. Since analytic tools are built on databases, there is usually no easy way to analyze data in raw format. Therefore, assumptions have to be made about the data before we can look at them. We have to make decisions on data structure before loading raw data into a database. This can be a source of problems. Once the designs are implemented, they are difficult to change. Without the benefit of a thorough analysis, an initial design may hinder the optimal extraction of information and knowledge. This may not be optimal.

Big Data offers major improvements over its predecessor in analytics, traditional business intelligence (BI). Take the fact that BI has always been top-down, putting data in the hands of executives and managers who are looking to track their businesses on the big-picture level. Big Data, on the other hand, is bottom-up. When properly harnessed, it empowers business end-users in the trenches, enabling them to carry out in-depth analysis to inform real-time decision-making.

While the scope of traditional BI is limited to structured data that can be stuffed into columns and rows on a data warehouse, the fact is that over 90% of today’s data is unstructured. BI could never have anticipated the multitude of images, MP3 files, videos and social media snippets that companies would contend with in the Big Data era, but that’s the reality of business today. Traditional BI is stuck in a rut, left behind by forward-looking businesses that are desperate to tame and gain competitive advantage from unstructured business data floating around within and beyond their enterprise.Traditional Business Intelligence (BI) systems provide various levels and kinds of analyses on structured data but they are not designed to handle unstructured data.

For these systems Big Data brings big problems because the data that flows in may be either structured or unstructured. That makes them hugely limited when it comes to delivering Big Data benefits.

The way forward is a complete rethink of the way we use BI - in terms of how the data is ingested, stored and analyzed.

CASE STUDY OF BIG DATA SOLUTIONS

The Internet of Trains
Analysing sensor data helps Siemens keep operators on track by reducing train failures

“We are heading towards next-generation maintenance”, says Gerhard Kress, Director of Mobility Data Services at Siemens. “It is a whole new business model. Instead of selling our customers a train, we sell them its performance over a certain period of time.” And with guarantee periods of up to 20 years, this business model is as attractive to Siemens’customers as it is risky for the company itself. From reactive to predictive maintenanceTrain operators the world over are expected to work miracles, i.e. never to be late. So, with acute service and availability targets to meet, an efficient maintenance program is important. And data-enabled functionality is a must for Siemens. Reactive maintenance (after an incident) and routine, preventive maintenance with its visual inspections and scheduled exchange of components, are no longer enough. We’ve moved on to more cost-effective, condition-based, predictive maintenance.The actual condition of components is measured via the transfer and remote monitoring of diagnostic sensor data; data which is also used to analyse patterns and trends. This helps predict when a component is likely to fail, so it can be repaired before anything untoward happens.To ensure the commercial sustainability of this approach, Siemens needs to use and re-use existing data, creating a kind of ‘Internet of Trains’. Towards this end, they’re analysing sensor data in near real time, which means they can react very quickly, ensuring that customer transport services aren’t interrupted. “It is really difficult to define every issue before it impacts operations using only data from the trains”, Kress explains. However, recent success stories prove that everything is possible. Predicting failures in time
For instance, Spanish train operator RENFE uses Siemens’ high-speed train, Velaro E, key components of which are continually monitored by Siemens. A train developing abnormal patterns is dispatched for an

What is Hadoop?

Hadoop is a platform that provides both distributed storage and computational capabilities. Hadoop was first conceived to fix a scalability issue that existed in Nutch, an open source crawler and search engine. At the time Google had published papers that described its novel distributed filesystem, the Google File System (GFS), and Map-Reduce, a computational framework for parallel processing. The successful implementation of these papers’ concepts in Nutch resulted in its split into two separate projects, the second of which became Hadoop, a first-class Apache project.

Hadoop, as shown in figure 1, is a distributed master-slave architecture that consists of the Hadoop Distributed File System (HDFS) for storage and Map-Reduce for computational capabilities. Traits intrinsic to Hadoop are data partitioning and parallel computation of large datasets. Its storage and computational capabilities scale with the addition of hosts to a Hadoop cluster, and can reach volume sizes in the petabytes on clusters with thousands of hosts.

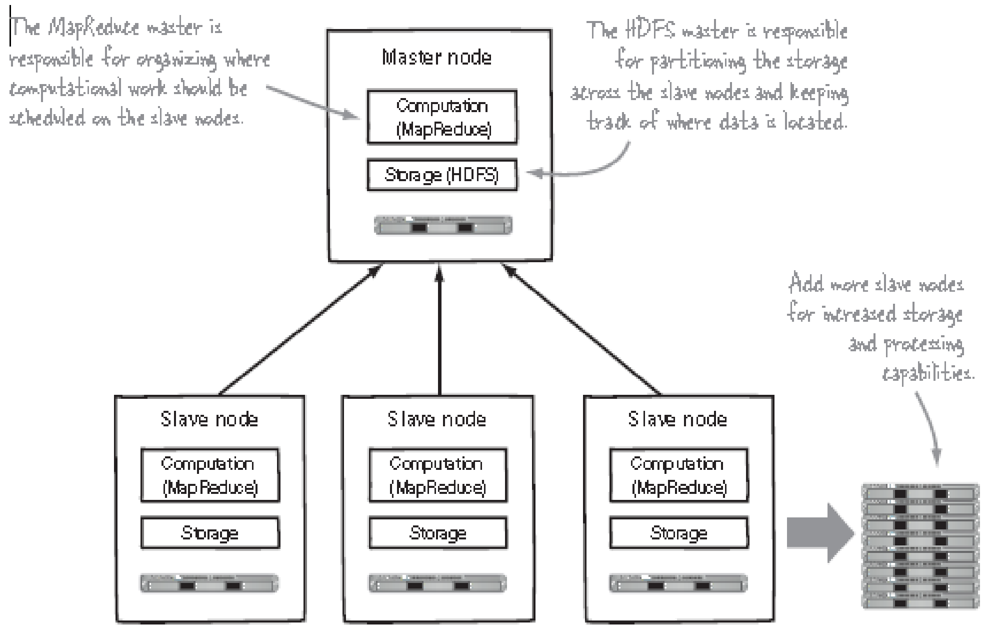

The MapReduce master is responsible for organizing where computational work should be scheduled on the slave nodes.
Master node
Computation (MapReduce)
Storage (HDFS)
The HDFS master is responsible for partitioning the storage across the slave nodes and keeping track of where data is located.
Add more slave nodes for increased storage and processing capabilities.
Slave node
Computation (MapReduce)
Storage
Slave node
Computation (MapReduce)
Storage
Slave node
Computation (MapReduce)
Storage

Fig- 1 High-level Hadoop architecture

Core Hadoop Components
HDFS
HDFS is the storage component of Hadoop. It's a distributed filesystem that's modeled after the Google File System (GFS) paper.4 HDFS is optimized for high throughput and works best when reading and writing large files (gigabytes and larger). To support this throughput HDFS leverages unusually large (for a filesystem) block sizes and data locality optimizations to reduce network input/output (I/O). Scalability and availability are also key traits of HDFS, achieved in part due to data replication and fault tolerance. HDFS replicates files for a configured number of times, is tolerant of both software and hardware failure, and automatically re-replicates data blocks on nodes that have failed. Figure 2 shows a logical representation of the components in HDFS: the Name- Node and the DataNode. It also shows an application that's using the Hadoop filesystem library to access HDFS. Now that you have a bit of HDFS knowledge, it's time to look at MapReduce, Hadoop's computation engine.

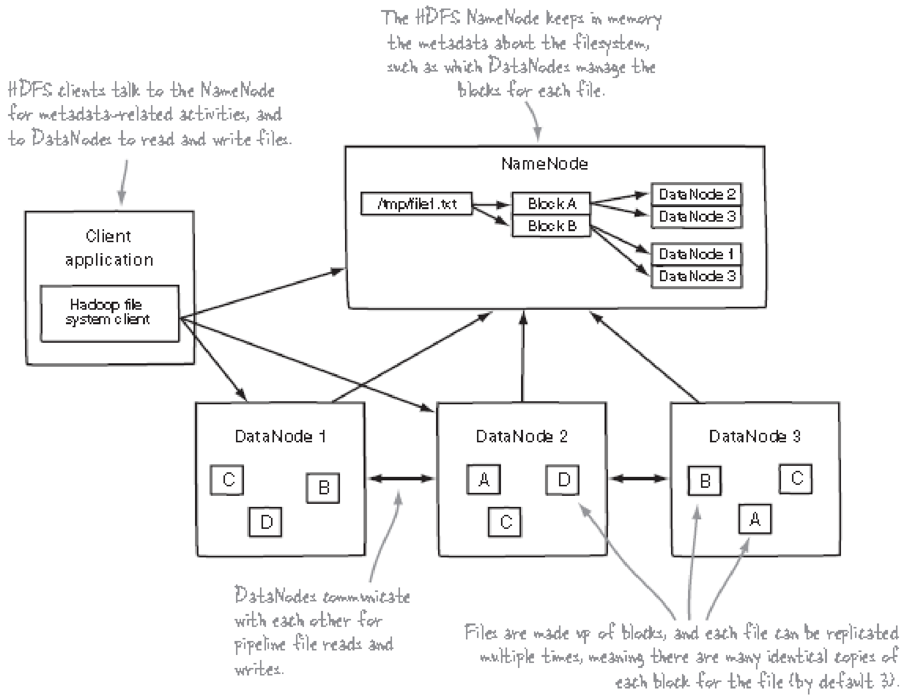

The HDFS NameNode keeps in memory the metadata about the filesystem, such as which DataNodes manage the blocks for each file.
HDFS clients talk to the NameNode for metadata-related activities, and to DataNodes to read and write files.
NameNode
/tmp/file1.txt
Block A
Block B
DataNode 2
DataNode 3
DataNode 1
DataNode 3
Client application
Hadoop file system client
DataNode 1
DataNode 2
DataNode 3
C
B
D
A
D
C
B
C
A
DataNodes communicate with each other for pipeline file reads and writes.
Files are made up of blocks, and each file can be replicated multiple times, meaning there are many identical copies of each block for the file (by default 3).

Fig- 2 HDFS architecture shows an HDFS client communicating with the master NameNode and slave DataNodes.

MAPREDUCE

MapReduce is a batch-based, distributed computing framework modeled after Google's paper on MapReduce. It allows you to parallelize work over a large amount of raw data, such as combining web logs with relational data from an OLTP database to model how users interact with your website. This type of work, which could take days or longer using conventional serial programming techniques, can be reduced down to minutes using MapReduce on a Hadoop cluster.

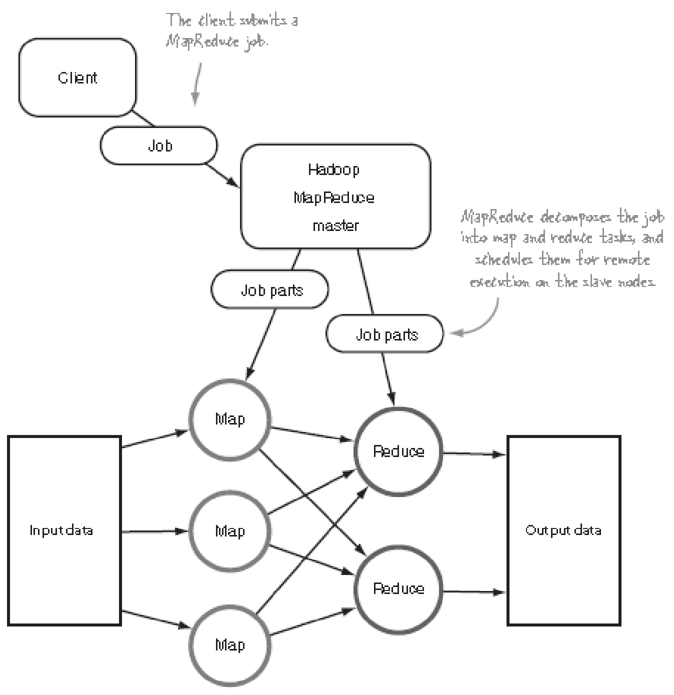
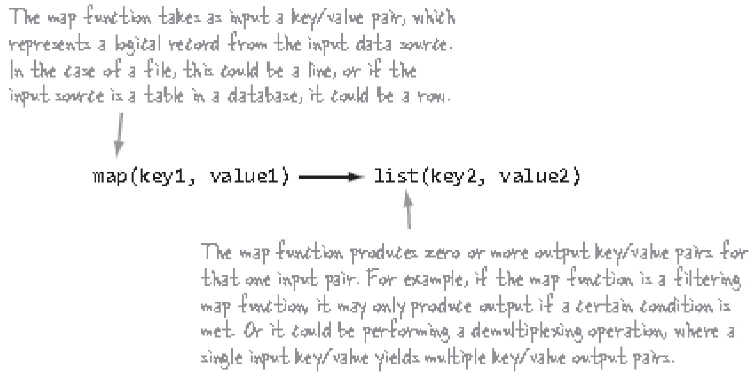

The client submits a MapReduce job.
Client
Job
Hadoop MapReduce master
MapReduce decomposes the job into map and reduce tasks, and schedules them for remote execution on the slave nodes.
Job parts
Job parts
Input data
Map
Map
Map
Reduce
Reduce
Output data

Fig-3 A client submitting a job to MapReduce

The MapReduce model simplifies parallel processing by abstracting away the complexities involved in working with distributed systems, such as computational parallelization, work distribution, and dealing with unreliable hardware and software. With this abstraction, MapReduce allows the programmer to focus on addressing business needs, rather than getting tangled up in distributed system complications. MapReduce decomposes work submitted by a client into small parallelized map and reduce workers, as shown in figure 3. The map and reduce constructs used in MapReduce are borrowed from those found in the Lisp functional programming language, and use a shared-nothing model6 to remove any parallel execution interdependencies that could add unwanted synchronization points or state sharing.

The role of the programmer is to define map and reduce functions, where the map function outputs key/value tuples, which are processed by reduce functions to produce the final output. Figure 4 shows a pseudo-code definition of a map function with regards to its input and output.


The map function takes as input a key/value pair, which represents a logical record from the input data source. In the case of a file, this could be a line, or if the input source is a table in a database, it could be a row.

```
map(key1, value1) ——▶ list(key2, value2)
```

The map function produces zero or more output key/value pairs for that one input pair. For example, if the map function is a filtering map function, it may only produce output if a certain condition is met. Or it could be performing a demultiplexing operation, where a single input key/value yields multiple key/value output pairs.

Fig 4 A logical view of the map function

The power of MapReduce occurs in between the map output and the reduce input, in the shuffle and sort phases, as shown in figure 5.

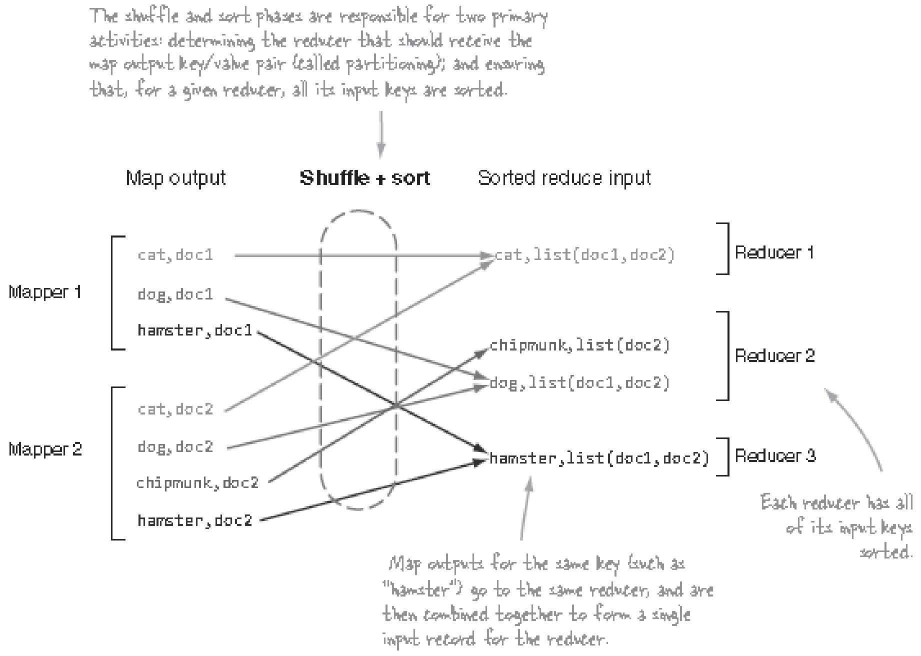

The shuffle and sort phases are responsible for two primary activities: determining the reducer that should receive the map output key/value pair (called partitioning); and ensuring that, for a given reducer, all its input keys are sorted.
Map output
Shuffle + sort
Sorted reduce input
Mapper 1
cat,doc1
dog,doc1
hamster,doc1
Mapper 2
cat,doc2
dog,doc2
chipmunk,doc2
hamster,doc2
cat,list(doc1,doc2)
Reducer 1
chipmunk,list(doc2)
dog,list(doc1,doc2)
Reducer 2
hamster,list(doc1,doc2)
Reducer 3
Each reducer has all of its input keys sorted.
Map outputs for the same key (such as "hamster") go to the same reducer, and are then combined together to form a single input record for the reducer.

Fig 5 MapReduce's shuffle and sort

Figure 6 shows a pseudo-code definition of a reduce function. Hadoop's MapReduce architecture is similar to the master-slave model in HDFS.

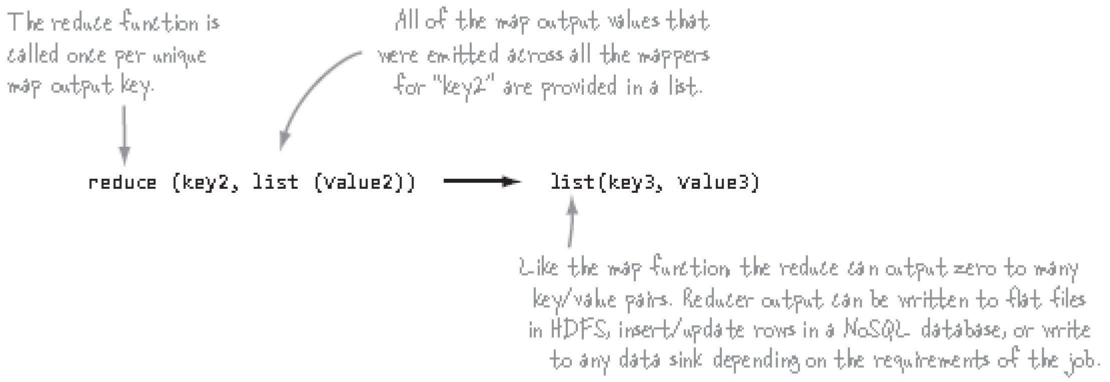

The reduce function is called once per unique map output key.
All of the map output values that were emitted across all the mappers for "key2" are provided in a list.
reduce (key2, list (value2)) → list(key3, value3)
Like the map function the reduce can output zero to many key/value pairs. Reducer output can be written to flat files in HDFS, insert/update rows in a NoSQL database, or write to any data sink depending on the requirements of the job.

Fig 6 A logical view of the reduce function

The main components of MapReduce are illustrated in its logical architecture, as shown in figure 7.

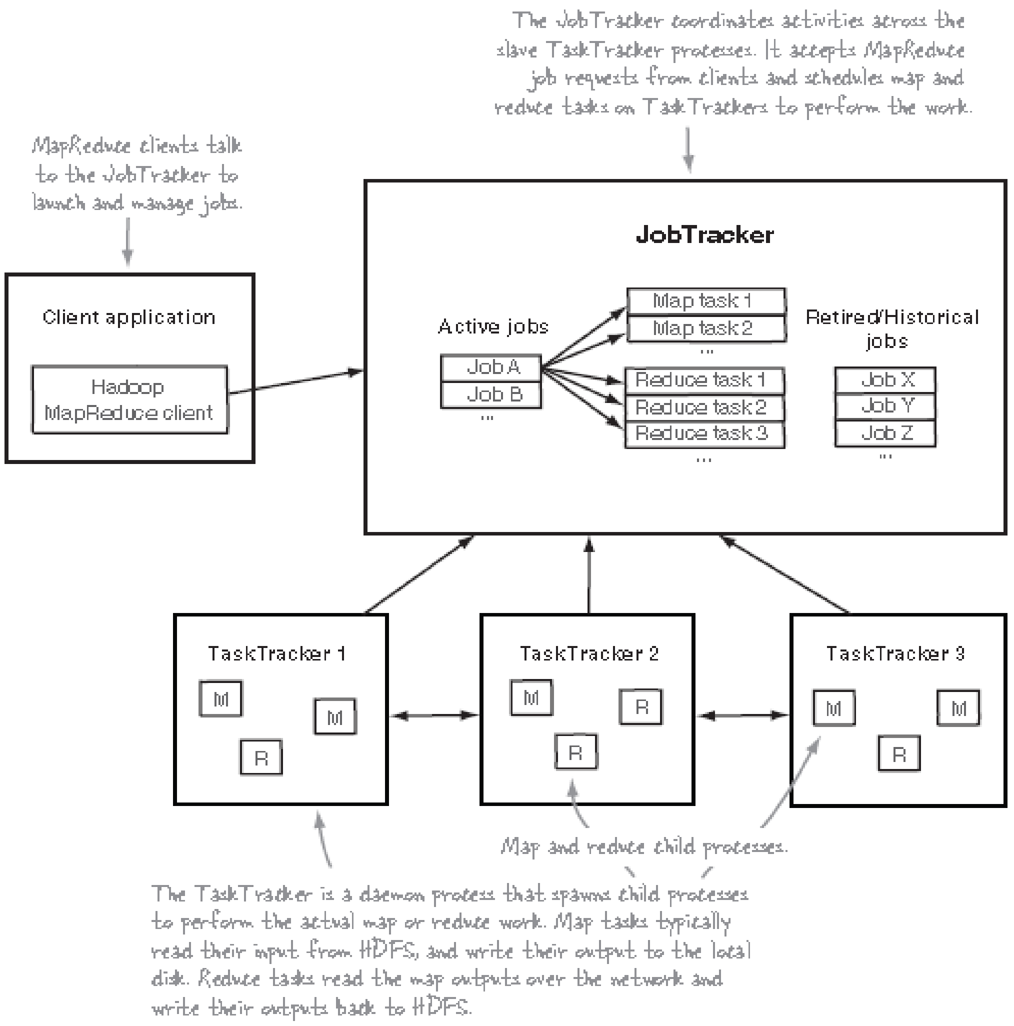

The JobTracker coordinates activities across the slave TaskTracker processes. It accepts MapReduce job requests from clients and schedules map and reduce tasks on TaskTrackers to perform the work.
MapReduce clients talk to the JobTracker to launch and manage jobs.
JobTracker
Client application
Hadoop MapReduce client
Active jobs
Job A
Job B
Map task 1
Map task 2
Reduce task 1
Reduce task 2
Reduce task 3
Retired/Historical jobs
Job X
Job Y
Job Z
TaskTracker 1
TaskTracker 2
TaskTracker 3
M
R
Map and reduce child processes.
The TaskTracker is a daemon process that spawns child processes to perform the actual map or reduce work. Map tasks typically read their input from HDFS, and write their output to the local disk. Reduce tasks read the map outputs over the network and write their outputs back to HDFS.

Fig 7 MapReduce logical architecture

Hadoop Ecosystem

The Hadoop ecosystem is diverse and grows by the day. It’s impossible to keep track of all of the various projects that interact with Hadoop in some form.

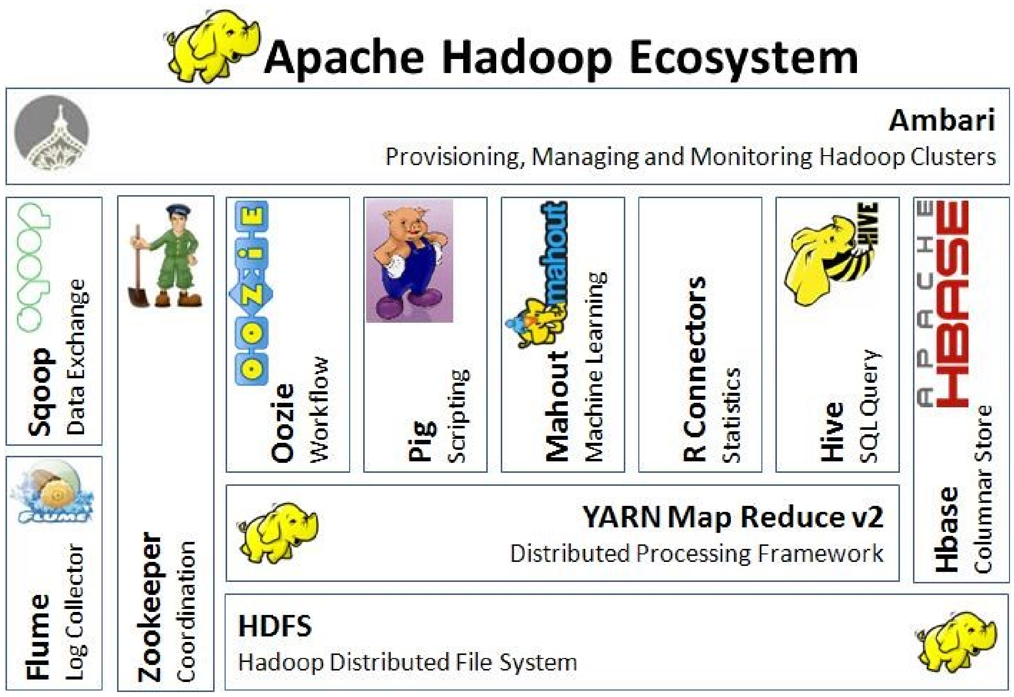


Fig 8 Hadoop Ecosystem

1. Storage

HDFS

The primary distributed file system used by Hadoop applications which runs on large clusters of commodity machines. HDFS clusters consist of a NameNode that manages the file system metadata and DataNodes that store the actual data.

Architecture:

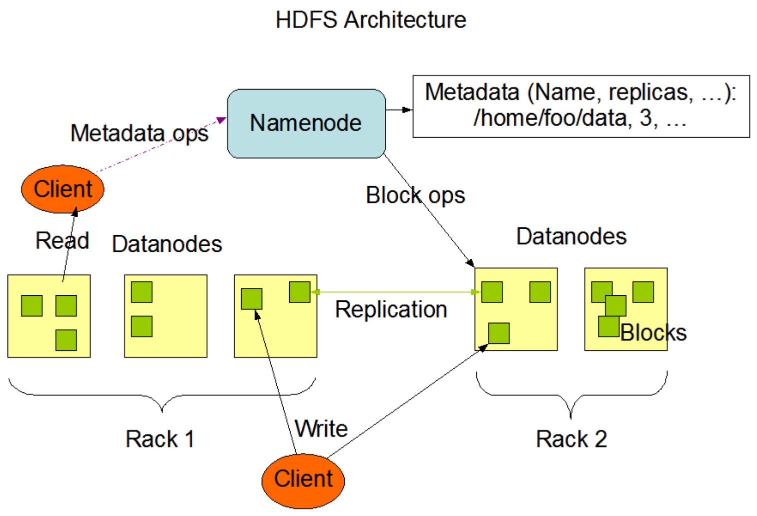


Fig 9 HDFS Architecture

Uses:

- Storage of large imported files from applications outside of the Hadoop ecosystem
- Staging of imported files to be processed by Hadoop applications

HBase
A distributed, column-oriented database. HBase uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads).

Architecture:

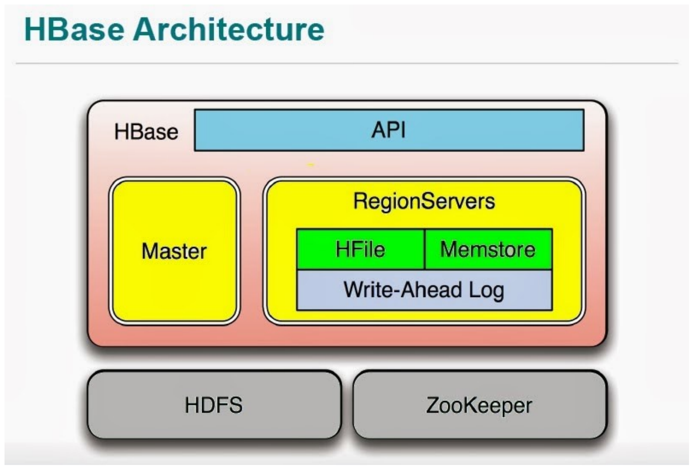


Fig 10 HBase Architecture

Uses:

- Storage of large data volumes (billions of rows) atop clusters of commodity hardware
- Bulk storage of logs, documents, real-time activity feeds and raw imported data
- Consistent performance of reads/writes to data used by Hadoop applications
- Data Store than can be aggregated or processed using MapReduce functionality
- Data platform for Analytics and Machine Learning

HCatalog
A table and storage management layer for Hadoop that enables Hadoop applications
(Pig, MapReduce, and Hive) to read and write data to a tabular form as opposed to files. It also provides REST APIs so that external systems can access these tables' metadata.

Architecture:

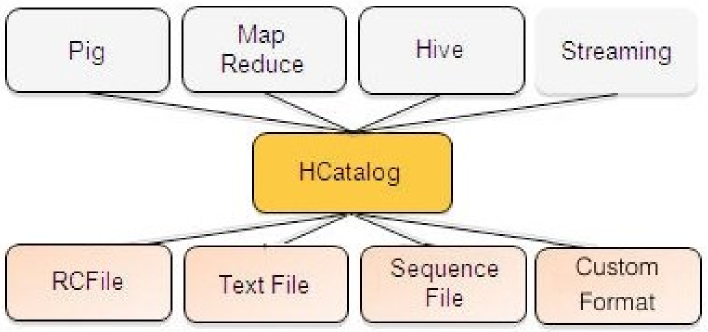


Fig 11 HCatalogAchitecture

Uses:

- Centralized location of storage for data used by Hadoop applications
- Reusable data store for sequenced and iterated Hadoop processes (ex: ETL)
- Storage of data in a relational abstraction

2. Processing

MapReduce

A distributed data processing model and execution environment that runs on large clusters of commodity machines. It uses the MapReducealgorithm which breaks down all operations into Map or Reduce functions.

Architecture:

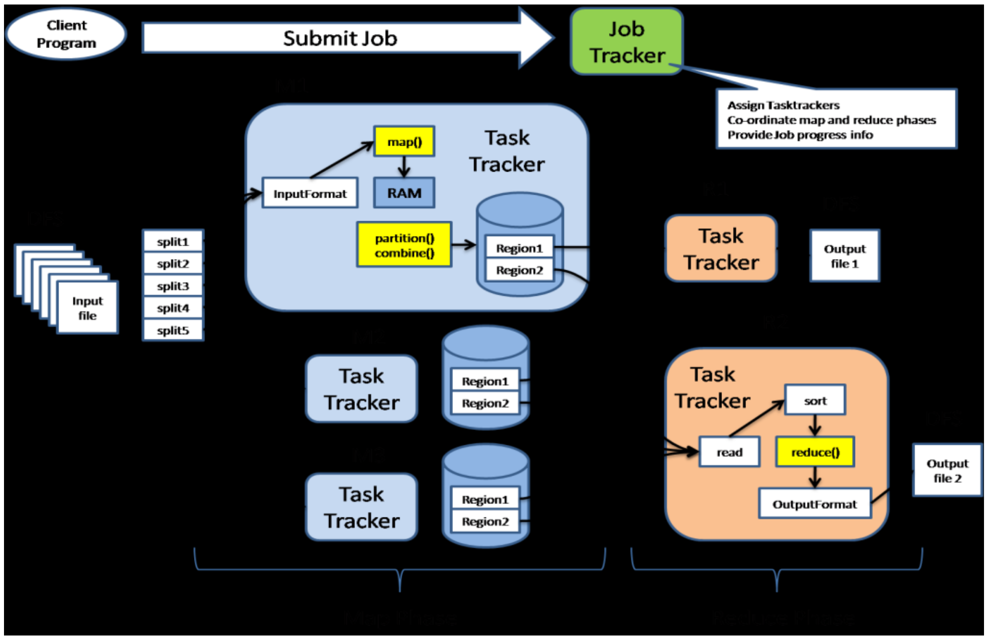


Fig 12 MapReduce Architecture

Uses:

- Aggregation (Counting, Sorting, Filtering, Stitching) on large and desperate data sets
- Scalable parallelism of Map or Reduce tasks
- Distributed task execution
- Machine learning

Pig

A scripting SQL based language and execution environment for creating complex MapReduce transformations. Functions are written in Pig Latin (the language) and translated into executable MapReduce jobs. Pig also allows the user to create extended functions (UDFs) using java.

Architecture:

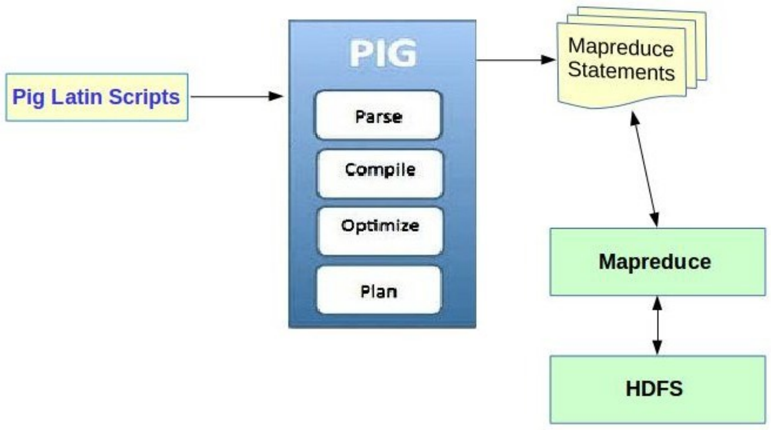


Fig 13 Pig Architecture

Uses:

- Scripting environment to execute ETL tasks/procedures on raw data in HDFS
- SQL based language for creating and running complex MapReduce functions
- Data processing, stitching, schematizing on large and desperate data sets

3. Querying

HIVE

A distributed data warehouse built on top of HDFS to manage and organize large amountsof data. Hive provides a query language based on SQL semantics (HiveQL) which is translated by the runtime engine to MapReduce jobs for querying the data.

Architecture:

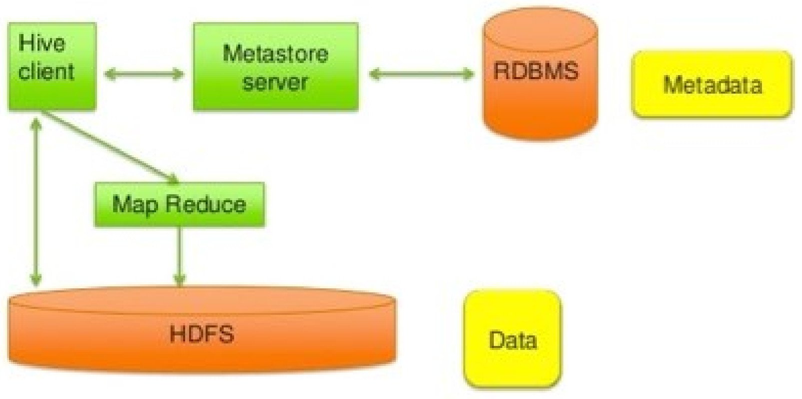


Fig 14 Hive architecture

Uses:

- Schematized data store for housing large amounts of raw data
- SQL-like Environment to execute analysis and querying tasks on raw data in HDFS
- Integration with outside RDBMS applications

4. External integration

Flume
A distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data into HDFS. Flume's transports large quantities of event data using a steaming data flow architecture that is fault tolerant and failover recovery ready.

Architecture:

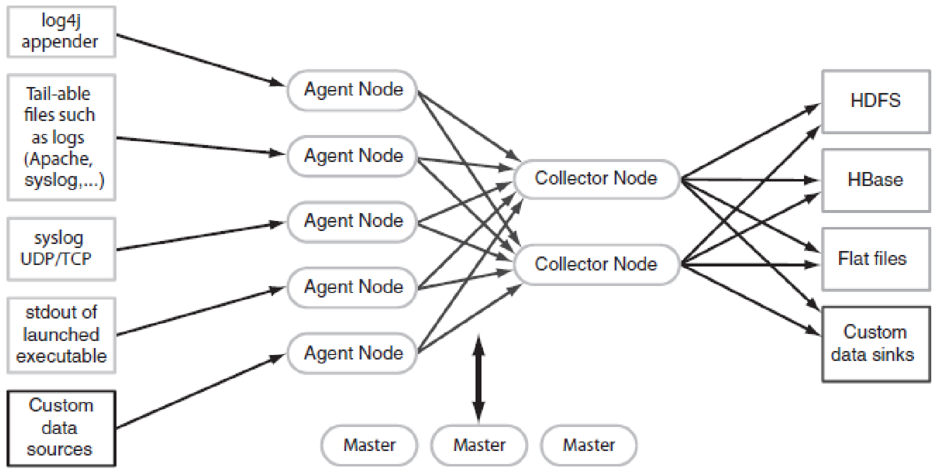


Fig 15 Flume Architecture

Uses:

- Transportation of large amounts of event data (network traffic, logs, email messages)
- Stream data from multiple sources into HDFS
- Guaranteed and reliable real-time data streaming to Hadoop applications

Physical Architecture

The physical architecture lays out where you install and execute various components. Figure 8 shows an example of a Hadoop physical architecture involving Hadoop and its ecosystem, and how they would be distributed across physical hosts. ZooKeeper requires an odd-numbered quorum, so the recommended practice is to have at least three of them in any reasonably sized cluster. For Hadoop physical architecture to include CPU, RAM, disk, and network, because they all have an impact on the throughput and performance of your cluster.

The term commodity hardware is often used to describe Hadoop hardware requirements. It's true that Hadoop can run on any old servers you can dig up, but you still want your cluster to perform well, and you don't want to swamp your operations department with diagnosing and fixing hardware issues. Therefore, commodity refers to mid-level rack servers with dual sockets, as much error-correcting RAM as is affordable, and SATA drives optimized for RAID storage. Using RAID, however, is strongly discouraged on the DataNodes, because HDFS already has replication and error-checking built-in; but on the NameNode it's strongly recommended for additional reliability. From a network topology perspective with regards to switches and firewalls, all of the master and slave nodes must be able to open connections to each other. For small clusters, all the hosts would run 1 GB network cards connected to a single, good-quality switch. For larger clusters look at 10 GB top-of-rack switches that have at least multiple 1 GB uplinks to dual-central switches. Client nodes also need to be able to talk to all of the master and slave nodes, but if necessary that access can be from behind a firewall that permits connection establishment only from the client side.

HADOOP LIMITATIONS

Common areas identified as weaknesses across HDFS and MapReduce include availability and security. All of their master processes are single points of failure, although should note that there's active work on High Availability versions in the community. Security is another area that has its wrinkles, and again another area that's receiving focus.

HIGH AVAILABILITY

Until the Hadoop 2.x release, HDFS and MapReduce employed single-master models, resulting in single points of failure.11 The Hadoop 2.x version will eventually bring both NameNode and JobTracker High Availability (HA) support. The 2.x NameNode HA design requires shared storage for NameNode metadata, which may require expensive HA storage. It supports a single standby NameNode, preferably on a separate rack.

SECURITY

Hadoop does offer a security model, but by default it's disabled. With the security model disabled, the only security feature that exists in Hadoop is HDFS file and directory- level ownership and permissions. But it's easy for malicious users to subvert and assume other users' identities. By default, all other Hadoop services are wide open, allowing any user to perform any kind of operation, such as killing another user's MapReduce jobs. Hadoop can be configured to run with Kerberos, a network authentication protocol, which requires Hadoop daemons to authenticate clients, both user and other Hadoop components. Kerberos can be integrated with an organization's existing Active Directory, and therefore offers a single sign-on experience for users. Finally, and most important for the government sector, there's no storage or wire-level encryption in Hadoop. Overall, configuring Hadoop to be secure has a high pain point due to its complexity.Let's examine the limitations of some of the individual systems.

HDFS
The weakness of HDFS is mainly around its lack of High Availability, its inefficient handling of small files, and its lack of transparent compression. HDFS isn't designed to work well with random reads over small files due to its optimization for sustained throughput. The community is waiting for append support for files, a feature that's nearing production readiness.

MAPREDUCE
MapReduce is a batch-based architecture, which means it doesn't lend itself to use cases that need real-time data access. Tasks that require global synchronization or sharing of mutable data aren't a good fit for MapReduce, because it's a shared nothing architecture, which can pose challenges for some algorithms.

ECOSYSTEM VERSION COMPATIBILITIES
There also can be version-dependency challenges to running Hadoop. For example, HBase only works with a version of Hadoop that's not verified as production ready, due to its HDFS sync requirements (sync is a mechanism that ensures that all writes to a stream have been written to disk across all replicas). Hadoop versions 0.20.205 and newer, including 1.x and 2.x, include sync support, which will work with HBase. Other challenges with Hive and Hadoop also exist, where Hive may need to be recompiled to work with versions of Hadoop other than the one it was built against. Pig has had compatibility issues, too. For example, the Pig 0.8 release didn't work with Hadoop 0.20.203, requiring manual intervention to make them work together. This is one of the advantages to using a Hadoop distribution other than Apache, as these compatibility problems have been fixed.
One development worth tracking is the creation of BigTop (http://incubator.apache.org/projects/bigtop.html), currently an Apache incubator project, which is a contribution from Cloudera to open source its automated build and compliance system. It includes all of the major Hadoop ecosystem components and runs a number of integration tests to ensure they all work in conjunction with each other.

References:

Anand Rajaraman and Jeff Ullman "Mining of Massive Datasets", Cambridge University Press

Objective Question (minimum 10-15) with answers.

1. As companies move past the experimental phase with Hadoop, many cite the need for additional capabilities, including:

   a) Improved data storage and information retrieval

   b) Improved extract, transform and load features for data integration

c) Improved data warehousing functionality

d) Improved security, workload management and SQL support

Answer:d

Explanation:Adding security to Hadoop is challenging because all the interactions do not follow the classic client- server pattern.

2. Point out the correct statement :

   a) Hadoop do need specialized hardware to process the data

   b) Hadoop 2.0 allows live stream processing of real time data

   c) In Hadoop programming framework output files are divided in to lines or records

   d) None of the mentioned

Answer:b

Explanation:Hadoop batch processes data distributed over a number of computers ranging in 100s and 1000s.

3. According to analysts, for what can traditional IT systems provide a foundation when they're integrated with big data technologies like Hadoop ?

   a) Big data management and data mining

   b) Data warehousing and business intelligence

   c) Management of Hadoop clusters

   d) Collecting and storing unstructured data

Answer:a

Explanation:Data warehousing integrated with Hadoop would give better understanding of data.

4. Hadoop is a framework that works with a variety of related tools. Common cohorts include:

   a) MapReduce, Hive and HBase

   b) MapReduce, MySQL and Google Apps

   c) MapReduce, Hummer and Iguana

   d) MapReduce, Heron and Trumpet

Answer:a

Explanation: To use Hive with HBase you'll typically want to launch two clusters, one to run HBase and the other to run Hive.

5. Point out the wrong statement :

   a) Hardtop’s processing capabilities are huge and its real advantage lies in the ability to process terabytes & petabytes of data

   b) Hadoop uses a programming model called “MapReduce”, all the programs should confirms to this model in order to work on Hadoop platform

   c) The programming model, MapReduce, used by Hadoop is difficult to write and test

   d) All of the mentioned

Answer:c

Explanation:The programming model, MapReduce, used by Hadoop is simple to write and test.

Subjective Questions:

1) What is Big Data

Big data is defined as the voluminous amount of structured, unstructured or semi-structured data that has huge potential for mining but is so large that it cannot be processed using traditional database systems. Big data is characterized by its high velocity, volume and variety that requires cost effective and innovative methods for information processing to draw meaningful business insights. More than the volume of the data – it is the nature of the data that defines whether it is considered as Big Data or not.

2) What do the four V’s of Big Data denote?

a) Volume –Scale of data

b) Velocity –Analysis of streaming data

c) Variety – Different forms of data

d) Veracity –Uncertainty of data

3) How big data analysis helps businesses increase their revenue? Give example

Big data analysis is helping businesses differentiate themselves – for example Walmart the world’s largest retailer in 2014 in terms of revenue - is using big data analytics to increase its sales through better predictive analytics, providing customized recommendations and launching new products based on customer preferences and needs. Walmart observed a significant 10% to 15% increase in online sales for $1 billion in incremental revenue. There are many more companies like Facebook, Twitter, LinkedIn, Pandora, JPMorgan Chase, Bank of America, etc. using big data analytics to boost their revenue.

4)Name some companies that use Hadoop

Yahoo (One of the biggest user & more than 80% code contributor to Hadoop)

Facebook, Netflix, Amazon, Adobe, eBay, Hulu, Spotify, Rubikloud, Twitter

University Questions:

Q. 1 What is Big Data? What is Hadoop? How big data and Hadoop are Linked? May 17

Q.2 Expalin Hadoop Ecosystem with core components. Explain its physical architecture. State limitations of Hadoop. May 17

Q.3 What is Triangular Matrix? How it is used for main memory counting?

MODULE 2

HADOOP HDFS AND MAPREDUCE

Syllabus:

Module	Syllabus	Duration
2	**Distributed File Systems:** Physical Organization of Compute Nodes, Large-Scale File-System Organization**.** **MapReduce:** The Map Tasks, Grouping by Key, The Reduce Tasks, Combiners, Details of MapReduce Execution, Coping With Node Failures. **Algorithms Using MapReduce:** Matrix-Vector Multiplication by MapReduce, Relational-Algebra Operations, Computing Selections by MapReduce, Computing Projections by MapReduce, Union, Intersection, and Difference by MapReduce **Hadoop Limitations**	10

Introduction:
Modern data-mining applications, often called "big-data" analysis, require us to manage immense amounts of data quickly. In many of these applications, the data is extremely regular, and there is ample opportunity to exploit parallelism. Important examples are:

1. The ranking of Web pages by importance, which involves an iterated matrix-vector multiplication where the dimension is many billions.
2. Searches in "friends" networks at social-networking sites, which involve graphs with hundreds of millions of nodes and many billions of edges.

To deal with applications such as these, a new software stack has evolved. These programming systems are designed to get their parallelism not from a "supercomputer," but from "computing clusters" – large collections of commodity hardware, including conventional processors ("compute nodes") connected by Ethernet cables or inexpensive switches. The software stack begins with a new form of file system, called a "distributed file system," which features much larger units than the disk blocks in a conventional operating system. Distributed file systems also provide replication of data or redundancy to protect against the frequent media failures that occur when data is distributed over thousands of low-cost compute nodes. On top of these file systems, many different higher-level programming systems have been developed. Central to the new software stack is a programming system called MapReduce. Implementations of MapReduce enable many of the most common calculations on large-scale data to be performed on

computing
clusters efficiently and in a way that is tolerant of hardware failures during the computation.MapReduce systems are evolving and extending rapidly. Today, it is common for MapReduce programs to be created from still higher-level programmingsystems, often an implementation of SQL. Further, MapReduce turns out to be a useful, but simple, case of more general and powerful ideas. We include in this chapter a discussion of generalizations of MapReduce, first to systems that support acyclic workflows and then to systems that implement recursive algorithms.

Distributed File Systems:
Most computing is done on a single processor, with its main memory, cache, and local disk (a compute node). In the past, applications that called for parallel processing, such as large scientific calculations, were done on special-purpose parallel computers with many processors and specialized hardware. However,the prevalence of large-scale Web services has caused more and more computing to be done on installations with thousands of compute nodes operating more or less independently. In these installations, the compute nodes are commodity hardware, which greatly reduces the cost compared with special-purpose parallel machines.

These new computing facilities have given rise to a new generation of programming systems. These systems take advantage of the power of parallelism and at the same time avoid the reliability problems that arise when the computing hardware consists of thousands of independent components, any of which could fail at any time. In this section, we discuss both the characteristics of these computing installations and the specialized file systems that have been developed to take advantage of them.

Physical Organization ofCompute Nodes

The new parallel-computing architecture, sometimes called cluster computing, is organized as follows. Compute nodes are stored on racks, perhaps 8–64 on a rack. The nodes on a single rack are connected by a network, typically gigabit Ethernet. There can be many racks of compute nodes, and racks are connected by another level of network or a switch. The bandwidth of inter-rack communication is somewhat greater than the intrarack Ethernet, but given the number of pairs of nodes that might need to communicate between racks, this bandwidth may be essential. Figure 2.1 suggests the architecture of a largescale computing system. However, there may be many more racks and many more compute nodes per rack.

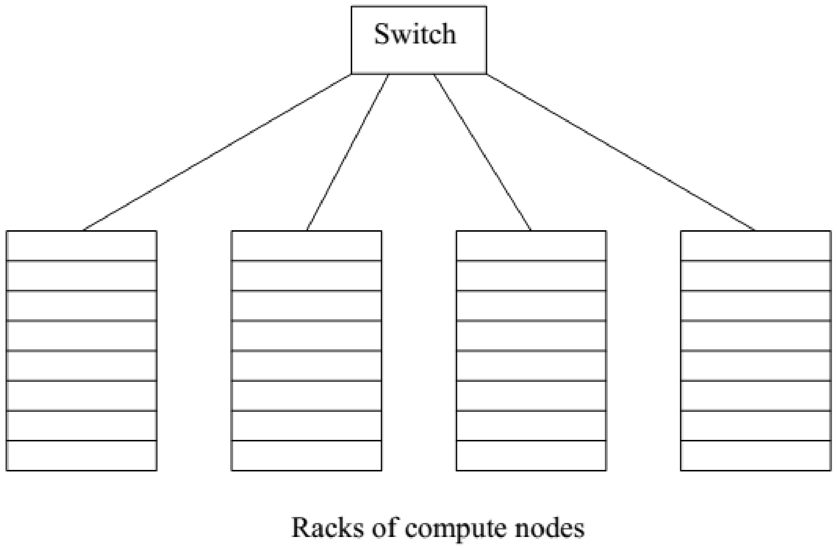


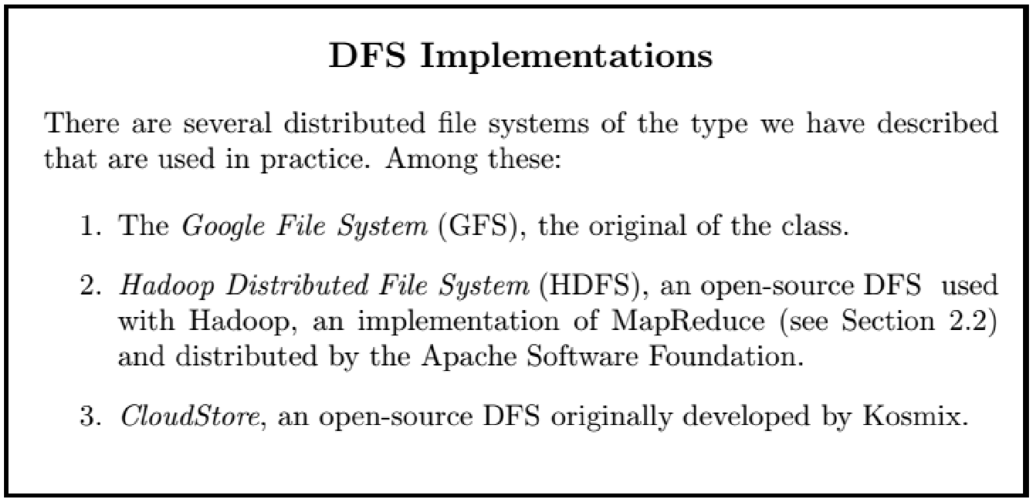
Fig - Compute nodes are organized into racks, and racks are interconnected by a switch

It is a fact of life that components fail, and the more components, such as compute nodes and interconnection networks, a system has, the more frequently something in the system will not be working at any given time. For systems such as Fig. the principal failure modes are the loss of a single node (e.g.,the disk at that node crashes) and the loss of an entire rack (e.g., the network connecting its nodes to each other and to the outside world fails). Some important calculations take minutes or even hours on thousands of compute nodes. If we had to abort and restart the computation every time one component failed, then the computation might never complete successfully. The solution to this problem takes two forms:

1. Files must be stored redundantly. If we did not duplicate the file at several compute nodes, then if one node failed, all its files would be unavailable until the node is replaced. If we did not back up the files at all, and the disk crashes, the files would be lost forever.
2. Computations must be divided into tasks, such that if any one task fails to execute to completion, it can be restarted without affecting other tasks. This strategy is followed by the MapReduce programming system

Large-Scale File-System Organization.

To exploit cluster computing, files must look and behave somewhat differently from the conventional file systems found on single computers. This new file system, often called a distributed file system or DFS (although this term has had other meanings in the past), is typically used as follows.



DFS Implementations

There are several distributed file systems of the type we have described that are used in practice. Among these:

1. The *Google File System* (GFS), the original of the class.
2. *Hadoop Distributed File System* (HDFS), an open-source DFS used with Hadoop, an implementation of MapReduce (see Section 2.2) and distributed by the Apache Software Foundation.
3. *CloudStore*, an open-source DFS originally developed by Kosmix.

- Files can be enormous, possibly a terabyte in size. If you have only small files, there is no point using a DFS for them.
- Files are rarely updated. Rather, they are read as data for some calculation, and possibly additional data is appended to files from time to time. For example, an airline reservation system would not be suitable for a DFS, even if the data were very large, because the data is changed so frequently.

Files are divided into chunks, which are typically 64 megabytes in size. Chunks are replicated, perhaps three times, at three different compute nodes. Moreover, the nodes holding copies of one chunk should be located on different racks, so we don't lose all copies due to a rack failure. Normally, both the chunk size and the degree of replication can be decided by the user.

To find the chunks of a file, there is another small file called the master node or name node for that file. The master node is itself replicated, and a directory for the file system as a whole knows where to find its copies. The directory itself can be replicated, and all participants using the DFS know where the directory copies are.

MapReduce:

MapReduce is a style of computing that has been implemented in several systems, including Google'sinternal implementation (simply called MapReduce) and the popular open-source implementation Hadoop which can be obtained, along with the HDFS file system from the Apache Foundation. You can use an implementation of MapReduce to manage many large-scale computations in a way that is tolerant of hardware faults. All you need to write are two functions, called Map and Reduce, while the system manages the parallel execution, coordination of tasks that execute Map or Reduce, and also deals with the possibility that one of these tasks will fail to execute. In brief, a MapReducecomputation executes as follows:
1. Some number of Map tasks each are given one or more chunks from a distributed file system. These

Map tasks turn the chunk into a sequence of key-value pairs. The way key value pairs are produced from the input data is determined by the code written by the user for the Map function.

2. The key-value pairs from each Map task are collected by a master controller and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.

3. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way. The manner of combination of values is determined by the codewritten by the user for the Reduce function.

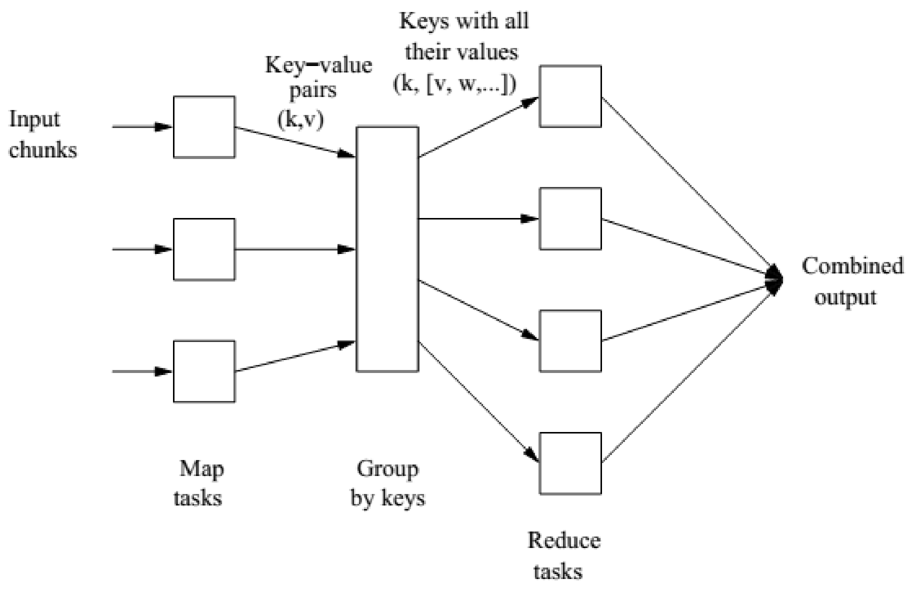


Fig-Schematic of a MapReduce computation

The Map Tasks

We view input files for a Map task as consisting of elements, which can be any type: a tuple or a document, for example. A chunk is a collection of elements, and no element is stored across two chunks. Technically, all inputsto Map tasks and outputs from Reduce tasks are of the key-value-pair form, but normally the keys of input elements are not relevant and we shall tend to ignore them. Insisting on this form for inputs and outputs is motivated by the desire to allow composition of several MapReduce processes. The Map function takes an input element as its argument and produces zero or more key-value pairs. The types of keys and values are each arbitrary. Further, keys are not "keys" in the usual sense; they do not have to be unique. Rather a Map task can produce several key-value pairs with the same key, even from the same element. Example 2.1 :We shall illustrate a MapReduce computation with what has become the standard example application: counting the number of occurrences for each word in a collection of documents. In this example, the input file is a repository of documents, and each document is an element. The Map function for this example uses keys that are of type String (the words) and values that are integers. The Map task reads a document and breaks it into its sequence of words w1, w2, . . . ,wn. It then emits a sequence of key-value pairs where the value is always 1. That is, the output of the Map task for this document is the sequence of key-value pairs:

(w1 , 1), (w2, 1), . . . , (wn, 1)

Note that a single Map task will typically process many documents – all the documents in one or more chunks. Thus, its output will be more than the sequence for the one document suggested above. Note also that if a word w appears m times among all the documents assigned to that process, then there will be m key-value pairs (w, 1) among its output. An option, which we discuss in Section 2.2.4, is to combine these m pairs into a single pair (w, m), but we can only do that because, as we shall see, the Reduce tasks apply an associative and commutative operation, addition, to the values.

Grouping by Key

As soon as the Map tasks have all completed successfully, the key-value pairs are grouped by key, and the values associated with each key are formed into a list of values. The grouping is performed by the system, regardless of what the Map and Reduce tasks do. The master controller process knows how many Reduce tasks there will be, say r such tasks. The user typically tells the MapReduce system what r should be. Then the master controller picks a hash function that applies to keys and produces a bucket number from 0 to r − 1. Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files. Each file is destined for one of the Reduce tasks.

To perform the grouping by key and distribution to the Reduce tasks, the master controller merges the files from each Map task that are destined for a particular Reduce task and feeds the merged file to that process as a sequence of key-list of-value pairs. That is, for each key k, the input to theReduce task that handles key k is a pair of the form (k, [v1, v2, . . . ,vn]), where (k, v1), (k, v2), . . . , (k, vn) are all the key-value pairs with key k coming from all the Map tasks.

The Reduce Task

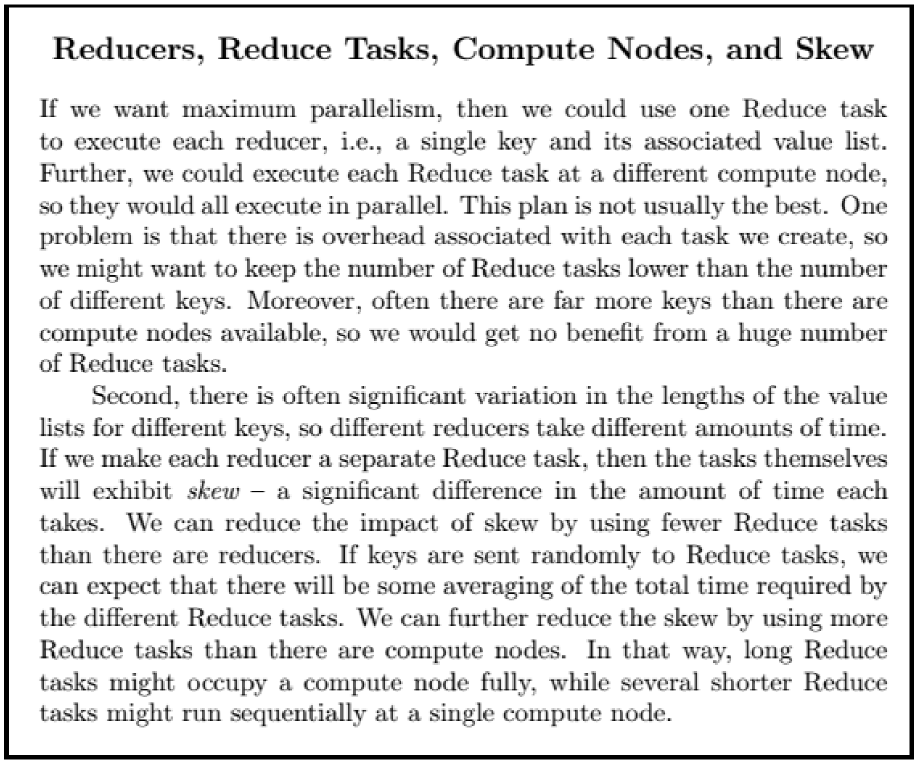
The Reduce function's argument is a pair consisting of a key and its list of associated values. The output of the Reduce function is a sequence of zero or more key-value pairs. These key-value pairs can be of a type different from those sent from Map tasks to Reduce tasks, but often they are the same type. We shall refer to the application of the Reduce function to a single key and its associated list of values as a reducer.

A Reduce task receives one or more keys and their associated value lists. That is, a Reduce task executes one or more reducers. The outputs from all the Reduce tasks are merged into a single file. Reducers may be partitioned among a smaller number of Reduce tasks is by hashing the keys and associating each Reduce task with one of the buckets of the hash function.

Example 2.2 : Let us continue with the word-count example of Example 2.1. The Reduce function simply adds up all the values. The output of a reducer consists of the word and the sum. Thus, the output of all the Reduce tasks is a sequence of (w, m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents.

Combiners

Sometimes, a Reduce function is associative and commutative. That is, the values to be combined can be combined in any order, with the same result. The addition performed in Example 2.2 is an example of an associative and commutative operation. It doesn't matter how we group a list of numbers v1, v2, . . . ,vn; the sum will be the same. When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks. For example, instead of the Map tasks in Example 2.1 producing many pairs (w, 1), (w, 1), . . ., we could apply the Reduce function within the Map task, before the output of the Map tasks is subject to grouping and aggregation. These key-value pairs would thus be replaced by one pair with key w and value equal to the sum of all the 1's in all those pairs. That is, the pairs with key w generated by a single Map task would be replaced by a pair (w, m), where m is the number of times that wappears among the documents handled by this Map task. Note that it is still necessary to do grouping and aggregation and to pass the result to the Reducetasks, since there will typically be one key value pair with key w coming from each of the Map tasks.



Reducers, Reduce Tasks, Compute Nodes, and Skew

If we want maximum parallelism, then we could use one Reduce task to execute each reducer, i.e., a single key and its associated value list. Further, we could execute each Reduce task at a different compute node, so they would all execute in parallel. This plan is not usually the best. One problem is that there is overhead associated with each task we create, so we might want to keep the number of Reduce tasks lower than the number of different keys. Moreover, often there are far more keys than there are compute nodes available, so we would get no benefit from a huge number of Reduce tasks.

Second, there is often significant variation in the lengths of the value lists for different keys, so different reducers take different amounts of time. If we make each reducer a separate Reduce task, then the tasks themselves will exhibit *skew* – a significant difference in the amount of time each takes. We can reduce the impact of skew by using fewer Reduce tasks than there are reducers. If keys are sent randomly to Reduce tasks, we can expect that there will be some averaging of the total time required by the different Reduce tasks. We can further reduce the skew by using more Reduce tasks than there are compute nodes. In that way, long Reduce tasks might occupy a compute node fully, while several shorter Reduce tasks might run sequentially at a single compute node.

Details of MapReduce Execution

Let us now consider in more detail how a program using MapReduce is executed. Figure 2.3 offers an outline of how processes, tasks, and files interact. Taking advantage of a library provided by a MapReduce system such as Hadoop, the user program forks a Master controller process and some number of Worker processes at different compute nodes. Normally, a Worker handles either Map tasks (a Map worker) or Reduce tasks (a Reduce worker), but not both. The Master has many responsibilities. One is to create some number of Map tasks and some number of Reduce tasks, these numbers being selected by the user program. These tasks will be assigned to Worker processes by the Master. It is reasonable to create one Map task for every chunk of the input file(s), but we may wish to create fewer Reduce tasks. The reason for limitingthe number of Reduce tasks is that it is necessary foreach Map task to create an intermediate file for each Reduce task, and if there are too many Reduce tasks the number of intermediate files explodes. The Master keeps track of the status of each Map and Reduce task (idle,executing at a particular Worker, or completed). A Worker process reports to the Master when it finishes a task, and a new task is scheduled by the Master for that Worker process.

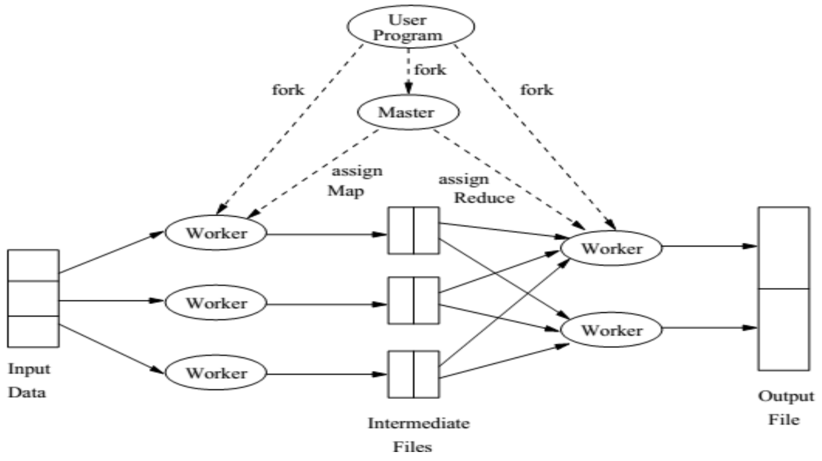


Fig- Overview of the execution of a MapReduce program

Each Map task is assigned one or more chunks of the input file(s) and executes on it the code written by the user. The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task. The Master is informed of the location and sizes of each of these files, and the Reduce task for which each is destined. When a Reduce task is assigned by the Master to a Worker process, that task is given all the files that form its input. The Reduce task executes code written by the user and writes its output to a file that is part of the surrounding distributed file system.

Coping With Node Failures:

The worst thing that can happen is that the compute node at which the Master is executing fails. In this case, the entire MapReduce job must be restarted. But only this one node can bring the entire process down; other failures will be managed by the Master, and the MapReduce job will complete eventually. Suppose the compute node at which a Map worker resides fails. This failure will be detected by the Master, because it periodically pings the Worker processes. All the Map tasks that were assigned to this Worker will have to be redone, even if they had completed. The reason for redoing completed Map tasks is that their output destined for the Reduce tasks resides at that compute node, and is now unavailable to the Reduce tasks. The Master sets the status of each of these Map tasks to idle and will schedule them on a Worker when one becomes available. The Master must also inform each Reduce task that the location of its input from that Map task has changed.

Dealing with a failure at the node of a Reduce worker is simpler. The Master simply sets the status of its currently executing Reduce tasks to idle. These will be rescheduled on another reduce worker later.

Algorithms Using MapReduce:

MapReduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. The entire distributed-file-system milieu makes sense only when files are very large and are rarely updated in place. Thus, we would not expectto use either a DFS or an implementation of MapReduce for managing online retail sales, even though a large on-line retailer such as Amazon.com uses thousands of compute nodes when processing requests over the Web. The reason is that the principal operations on Amazon data involve responding to searches for products, recording sales, and so on, processes that involve relatively little calculation and that change the database.2 On the other hand, Amazon might use MapReduce to perform certain analytic queries on large amounts of data, such as finding for each user those users whose buying patterns were most similar. The original purpose for which the Google implementation of MapReducewas created was to execute very large matrix-vector multiplications as areneeded in the calculation of PageRank .We shall see that matrix vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. Another important class of operations that can use MapReduce effectively are the relational-algebra operations. We shall examine the MapReduce execution of these operations as well.

Matrix-Vector Multiplication by MapReduce

Suppose we have an n × n matrix M, whose element in row i and column j will be denoted mij. Suppose we also have a vector v of length n, whose jth element is v j. Then the matrix vector product is the vector x of length n, whose ithelement xi is given by

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

If n = 100, we do not want to use a DFS or MapReduce for this calculation. But this sort of calculation is at the heart of the ranking of Web pages that goes on at search engines, and there, n is in the tens of billions.3 Let us first assume that n is large, but not so large that vector v cannot fit in main memory and thus be available to every Map task. The matrix M and the vector v each will be stored in a file of the DFS. We assume that the row-column coordinates of each matrix element will be discoverable, either from its position in the file, or because it is stored with explicit coordinates, as a triple (i, j, mij). We also assume the position of element vjin the vector v will be discoverable in the analogous way.

The Map Function: The Map function is written to apply to one element of M. However, if v is not already read into main memory at the compute node executing a Map task, then v is first read, in its entirety, and subsequently will be available to all applications of the Map function performed at this Map task. Each Map task will operate on a chunk of the matrix M. From each matrix element mij it produces the key-value pair (i, mijvj). Thus, all terms of the sum that make up the component xi of the matrix-vector product will get the same key, i.

The Reduce Function: The Reduce function simply sums all the values associated with a given key i. The result will be a pair (i, xi).

However, it is possible that the vector v is so large that it will not fit in its entirety in main memory. It is not required that v fit in main memory at a compute node, but if it does not then there will be a very large number ofdisk accesses as we move pieces of the vector into main memory to multiply components by elements of the matrix. Thus, as an alternative, we can divide the matrix into vertical stripes of equal width and divide the vector into an equalnumber 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 suggests what the partition looks like if the matrix and vector are each divided into five stripes.

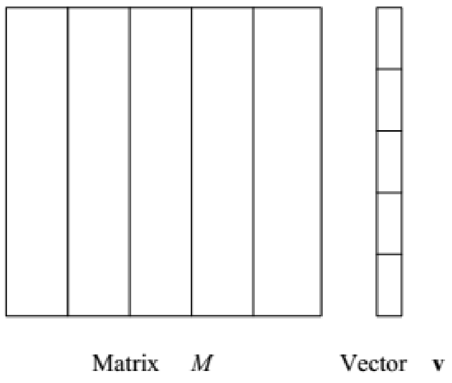


Fig- Division of a matrix and vector into five stripes

The ith stripe of the matrix multiplies only components from the ith 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.

We shall take up matrix-vector multiplication using MapReduce again in Section 5.2. There, because of the particular application (PageRank 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. We shall see there that the best strategy involves partitioning the matrix M into square blocks, rather than stripes.

Relational-Algebra Operations

There are a number of operations on large-scale data that are used in database queries.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. However, there are many operations on data that can be described easily in terms of the common database-query primitives, even if the queries themselvesare not executed within a database management system. Thus, 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, but to review, 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(A1 , A2, . . . , An) to say that the relation name is R and its attributes are A1, A2, . . . , An.

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4
...	...

Fig- Relation Links consists of the set of pairs of URL's, such that the first has one or more links to the second

Example: In Fig 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. 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: 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 σC(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 πS(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. There are also bag (multiset) versions of the operations in SQL, withsomewhat unintuitivedefinitions, but we shall not go into the bag versions of these operations here.
4. Natural Join: Given two relations, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each

attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. The natural join of relations R and S is denoted R ⊲⊳ S. While we shall discuss executing only the natural join with MapReduce, all equijoins (joins where the tuple-agreement condition involves equality of attributes from the two relations that do not necessarily have the same name) can be executed in the same manner.

5. Grouping and Aggregation:Given a relation R, partition its tuples according to their values in one set of attributes G, called the grouping attributes. Then, for each group, aggregate the values in certain other attributes. The normally permitted aggregations are SUM, COUNT, AVG, MIN, and MAX, with the obvious meanings. Note that MIN and MAX require that the aggregated attributes have a type that can be compared, e.g., numbers or strings, while SUM and AVG require that the type allow arithmetic operations. We denote a grouping-and-aggregation operation on a relation R by γX(R), where X is a list of elements that are either
   (a) A grouping attribute, or
   (b) An expression θ(A), where θ is one of the five aggregation operations such as SUM, and A is an attribute not among the grouping attributes.
   The result of this operation is one tuple for each group. That tuple has a component for each of the grouping attributes, with the value common to tuples of that group. It also has a component for each aggregation, with the aggregated value for that group.

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 σC(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 passeseach 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 components (or only the key components) of the output.

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 πS(R) as follows.

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

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

Observe that the Reduce operation is duplicate elimination. This operation is associative and commutative, so a combiner associated with each Map task can eliminate whatever duplicates are produced locally. However, the Reduce tasks are still needed to eliminate two identical tuples coming from different Map tasks.

Union, Intersection, and Difference by MapReduce

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 case.

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.

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. We shall thus use the relation as the value associated with the key t. Here is a specification for the two functions.

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.

The Reduce Function: For each key t, if the associated value list is [R], then produce (t, t). Otherwise, produce nothing.

Computing Natural Join by MapReduce

The idea behind implementing natural join via MapReduce can be seen if 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. We shall use the B-value of tuples from either relation as the key. The value will be the other component and the name of the relation, so the Reduce function can know where each tuple came from.

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 be associated with a list of pairs that are either of the form (R, a) or (S, c). 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 values.

The same algorithm works if the relations have more than two attributes. You can think of A as representing all those attributes in the schema of R but not S. B represents the attributes in both schemas, and C represents attributes only in the schema of S. The key for a tuple of R or S is the list of values in all the attributes that are in the schemas of both R and S. The value for a tuple of R is the name R together with the values of all the attributes belonging to R but not to S, and the value for a tuple of S is the name S together with the values of the attributes belonging to S but not R. The Reduce function looks at all the key-value pairs with a given key and combines those values from R with those values of S in all possible ways. From each pairing, the tuple produced has the values from R, the key values, and the values from S.

- Grouping and Aggregation by MapReduce

As with the join, 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 γA,θ(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).

The Reduce Function: Each key a represents a group. Apply the aggregation operator θ to the list [b1, b2, . . . ,bn] of B-values associated with key a. Theoutput is thepair (a, x), where x is the result of applying θ to the list. For example, if θ is SUM, then x = b1 + b2 + · · · + bn, and if θ is MAX, then x is the largest of b1, b2, . . . , bn.

If there are several grouping attributes, then the key is the list of the values of a tuple for all these attributes. If there is more than one aggregation, then the Reduce function applies each of them to the list of values associated with a given key and produces a tuple consisting of the key, including components for all grouping attributes if there is more than one, followed by the results of each of the aggregations.

Matrix Multiplication

If M is a matrix with element mij in row i and column j, and N is a matrix with element njk in row j and column k, then the product P = MN is the matrix P with element pik in row i and column k, where

$$p_{ik} = \sum_{j} m_{ij} n_{jk}$$

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. 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. In that case, the Map function will have to be designed to construct the I, J, and K components of tuples from the position of the data. 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 (mij, njk). What we want instead is the product of these elements, that is, the four-component tuple (i, j, k, v × w), because that represents the product mijnjk. 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 × W as the aggregation. That is, we can implement matrix multiplication as the cascade of two MapReduce operations, as follows. First:

The Map Function: For each matrix element mij, produce the key value pair j, (M, i, mij) . Likewise, for each matrix element njk, produce the key valuepair j, (N, k, njk) . 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, mijnjk. Now, we perform a grouping and aggregation by another MapReduce operation.

The Map Function: This function is just the identity. That is, for every inputelement 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 = MN.

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. 5 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 iis a row of M and k is a column of N. Here is a synopsis of the Map and Reduce functions.

The Map Function: For each element mij of M, produce all the key-value pairs (i, k), (M, j, mij) for k = 1, 2, . . ., up to the number of columns of N. Similarly, for each element njk of N, produce all the key-value pairs (i, k), (N, j, njk) for i = 1, 2, . . ., 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.

The Reduce Function: Each key (i, k) will have an associated list with all the values (M, j, mij) and (N, j, njk), 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 jthvalues on each list must have their third components, mij and njkextracted and multiplied. Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

You may notice that if a row of the matrix M or a column of the matrix N is so large that it will not fit in main memory, then the Reduce tasks will be forced to use an external sort to order the values associated with a given key (i, k). However, in that case, the matrices themselves are so large, perhaps 1020 elements, that it is unlikely we would attempt this calculation if the matrices were dense. If they are sparse, then we would expect many fewer values to be associated with any one key, and it would be feasible to do the sum of products in main memory.

References:

Anand Rajaraman and Jeff Ullman "Mining of Massive Datasets", Cambridge University Press

Subjective Questions:

Q.3 What is Mapreduce?

Q. 4 Explain one Algorithm using MapReduce

University Questions:

Q.2 What are Combiners? When should one use combiner in MapReduce?

Q. 3 What is MapReduce? Explain How Map Reduce work? What is shuffling in MapReduce?

MODULE 3

NoSQL

Syllabus:

Module	**Syllabus**	**Duration**
1	NoSQL 3.1What is NoSQL? NoSQL business drivers; NoSQL case studies; 3.2NoSQL data architecture patterns: Key-value stores, Graph stores, Column family (Bigtable) stores, Document stores, Variations of NoSQL architectural patterns; 3.3 Using NoSQL to manage big data: What is a big data NoSQL solution? Understanding the types of big data problems; Analyzing big data with ashared-nothing architecture; Choosing distribution models: master-slaveversus peer-to-peer; Four ways that NoSQL systems handle big dataproblems	06

What is NoSQL?
One of the challenges with NoSQL is defining it. The term NoSQL is problematic sinceitdoesn't really describe the core themes in the NoSQL movement. The term originatedfrom a group in the Bay Area who met regularly to talk about common concernsand issues surrounding scalable open source databases, and it stuck. Descriptive or not, it seems to be everywhere: in trade press, product descriptions, and conferences. Use the term NoSQL as a way of differentiating a system froma traditional relational database management system (RDBMS).For our purpose, we define NoSQL in the following way:

NoSQL is a set of concepts that allows the rapid and efficient processing of data sets witha focus on performance, reliability, and agility.

So what is NoSQL?

- It's more than rows in tables—NoSQL systems store and retrieve data from many formats: key-value stores, graph databases, column-family (Bigtable) stores, document stores, and even rows in tables.

- It’s free of joins—NoSQL systems allow you to extract your data using simple interfaces without joins.
- It’s schema-free—NoSQL systems allow you to drag-and-drop your data into a folder and then query it without creating an entity-relational model.
- It works on many processors—NoSQL systems allow you to store your database on multiple processors and maintain high-speed performance.
- It uses shared-nothing commodity computers—Most (but not all) NoSQL systems leverage low-cost commodity processors that have separate RAM and disk.
- It supports linear scalability—When you add more processors, you get a consistent increase in performance.
- It’s innovative—NoSQL offers options to a single way of storing, retrieving, and manipulating data. NoSQL supporters (also known as NoSQLers) have an inclusive attitude about NoSQL and recognize SQL solutions as viable options. To the NoSQL community, NoSQL means “Not only SQL.”

Equally important is what NoSQL is not:

- It’s not about the SQL language—The definition of NoSQL isn’t an application that uses a language other than SQL. SQL as well as other query languages are used with NoSQL databases.
- It’s not only open source—Although many NoSQL systems have an open source model, commercial products use NOSQL concepts as well as open source initiatives. You can still have an innovative approach to problem solving with a commercial product.
- It’s not only big data—Many, but not all, NoSQL applications are driven by the inability of a current application to efficiently scale when big data is an issue. Though volume and velocity are important, NoSQL also focuses on variability and agility.
- It’s not about cloud computing—Many NoSQL systems reside in the cloud to take advantage of its ability to rapidly scale when the situation dictates. NoSQL systems can run in the cloud as well as in your corporate data center.
- It’s not about a clever use of RAM and SSD—Many NoSQL systems focus on the efficient use of RAM or solid state disks to increase performance. Though this is important, NoSQL systems can run on standard hardware.
- It’s not an elite group of products—NoSQL isn’t an exclusive club with a few products. There are no membership dues or tests required to join. To be considered a NoSQLer, you only need to convince others that you have innovative solutions to their business problems.

NoSQL applications use a variety of data store types (databases). From the simple keyvalue
store that associates a unique key with a value, to graph stores used to associaterelationships, to document stores used for variable data, each NoSQL type of datastore has unique attributes and uses as identified in table 1.

Table 1

Type	Typical usage	Examples
Key-value store—A simple data storage system that uses a key to access a value	• Image stores •Key-based filesystems	•Berkeley DB •Memcache

	• Object cache •Systems designed to scale	•Redis •Riak •DynamoDB
Column family store—A sparse matrixsystem that uses a row and a column as keys	•Web crawler results •Big data problems that canrelax consistency rules	• Apache HBase • Apache Cassandra • Hypertable • Apache Accumulo
Graph store—For relationshipintensive problems	•Social networks • Fraud detection •Relationship-heavy data	• Neo4j • AllegroGraph •Bigdata (RDF data store) •InfiniteGraph (Objectivity)
Document store—Storing hierarchicaldata structures directly in the database	•High-variability data •Document search • Integration hubs •Web content management • Publishing	•MongoDB (10Gen) •CouchDB •Couchbase •MarkLogic • eXist-db •Berkeley DB XML

NoSQL business drivers
The scientist-philosopher Thomas Kuhn coined the term paradigm shift to identify arecurring process he observed in science, where innovative ideas came in bursts andimpacted the world in nonlinear ways. We'll use Kuhn's concept of the paradigm shiftas a way to think about and explain the NoSQL movement and the changes inthought patterns, architectures, and methods emerging today.Many organizations supporting single-CPU relational systems have come to a crossroads:the needs of their organizations are changing. Businesses have found value inrapidly capturing and analyzing large amounts of variable data, and making immediatechanges in their businesses based on the information they receive.Figure 1.1 shows how the demands of volume, velocity, variability, and agility play akey role in the emergence of NoSQL solutions. As each of these drivers applies pressureto the single-processor relational model, its foundation becomes less stable andin time no longer meets the organization's needs

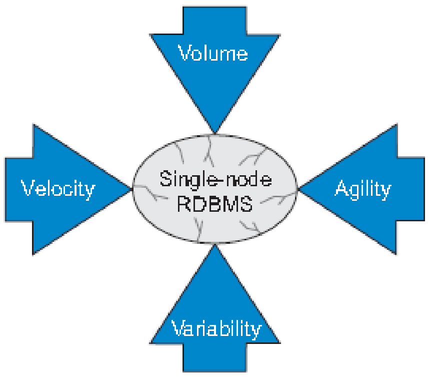


Fig 10how the business driversvolume, velocity, variability, and agility apply pressure to thesingle CPU system

- Volume

Without a doubt, the key factor pushing organizations to look at alternatives to their current RDBMSs is a need to query big data using clusters of commodity processors. Until around 2005, performance concerns were resolved by purchasing faster processors. In time, the ability to increase processing speed was no longer an option. As chip density increased, heat could no longer dissipate fast enough without chip overheating. This phenomenon, known as the power wall, forced systems designers to shift their focus from increasing speed on a single chip to using more processors working together. The need to scale out (also known as horizontal scaling), rather than scale up (faster processors), moved organizations from serial to parallel processing where data problems are split into separate paths and sent to separate processors to divide and conquer the work.

- Velocity

Though big data problems are a consideration for many organizations moving away from RDBMSs, the ability of a single processor system to rapidly read and write data is also key. Many single-processor RDBMSs are unable to keep up with the demands of real-time inserts and online queries to the database made by public-facing websites. RDBMSs frequently index many columns of every new row, a process which decreases system performance. When single-processor RDBMSs are used as a back end to a web store front, the random bursts in web traffic slow down response for everyone, and tuning these systems can be costly when both high read and write throughput is desired.

- Variability

Companies that want to capture and report on exception data struggle when attempting to use rigid database schema structures imposed by RDBMSs. For example, if a business unit wants to capture a few custom fields for a particular customer, all customer rows within the database need to store this information even though it doesn't apply. Adding new columns to an RDBMS requires the system be shut down and ALTER TABLE commands to be run. When a database is large, this process can impact system availability, costing time and money.

- Agility

The most complex part of building applications using RDBMSs is the process of putting data into and getting data out of the database. If your data has nested and repeated subgroups of data structures, you need to include an object-relational mapping layer. The responsibility of this layer is to generate the correct combination of INSERT, UPDATE, DELETE, and SELECT SQL statements to move object data to and from the RDBMS persistence layer. This process isn't simple and is associated with the largest barrier to rapid change when developing new or modifying existing applications. Generally, object-relational mapping requires experienced software developers who are familiar with object-relational frameworks such as Java Hibernate (or NHibernate for .Net systems). Even with experienced staff, small change requests can cause slowdowns in development and testing schedules.

NoSQL case studies
Our economy is changing. Companies that want to remain competitive need to findnew ways to attract and retain their customers. To do this, the technology and peoplewho create it must support these efforts quickly and in a cost-effective way. Newthoughts about how to implement solutions are moving away from traditional methodstoward processes, procedures, and technologies that at times seem bleeding-edge. The following case studies demonstrate how business problems have successfullybeen solved faster, cheaper, and more effectively by thinking outside the box. Table 2summarizes five case studies where NoSQL solutions were used to solve particular businessproblems. It presents the problems, the business drivers, and the ultimate findings.As you view subsequent sections, you'll begin to see a common theme emerge: somebusiness problems require new thinking and technology to provide the best solution.

Table-2 The key case studies associated with the NoSQL movement

Case study/standard	Driver	Finding
LiveJournal'sMemcache	Need to increase performance of database queries.	By using hashing and caching, data in RAM can be shared. This cuts down the number of read requests sent to the database, increasing performance.
Google's MapReduce	Need to index billions of web pages for search using low-cost hardware.	By using parallel processing, indexing billions of web pages can be done quickly with a large number of commodity processors.
Google's Bigtable	Need to flexibly store tabular data in a distributed system.	By using a sparse matrix approach, users can think of all data as being stored in a single table with billions of rows and millions of columns without the need for up-front data modeling.

Amazon’s Dynamo	Need to accept a web order 24 hours a day, 7 days a week.	A key-value store with a simple interface can be replicated even when there are large volumes of data to be processed.
MarkLogic	Need to query large collections of XML documents stored on commodity hardware using standard query languages.	By distributing queries to commodity servers that contain indexes of XML documents, each server can be responsible for processing data in its own local disk and returning the results to a query server.

- Case study: LiveJournal’sMemcache

Engineers working on the blogging system LiveJournal started to look at how their systems were using their most precious resource: the RAM in each web server. Live-Journal had a problem. Their website was so popular that the number of visitors usingthe site continued to increase on a daily basis. The only way they could keep up withdemand was to continue to add more web servers, each with its own separate RAM.To improve performance, the LiveJournal engineers found ways to keep the resultsof the most frequently used database queries in RAM, avoiding the expensive cost ofrerunning the same SQL queries on their database. But each web server had its owncopy of the query in RAM; there was no way for any web server to know that the servernext to it in the rack already had a copy of the query sitting in RAM.So the engineers at LiveJournal created a simple way to create a distinct “signature”of every SQL query. This signature or hash was a short string that represented a
SQL SELECT statement. By sending a small message between web servers, any webserver could ask the other servers if they had a copy of the SQL result already executed.If one did, it would return the results of the query and avoid an expensiveround trip to the already overwhelmed SQL database. They called their new systemMemcache because it managed RAM memory cache.Many other software engineers had come across this problem in the past. The conceptof large pools of shared-memory servers wasn’t new. What was different this timewas that the engineers for LiveJournal went one step further. They not only made thissystem work (and work well), they shared their software using an open source license,and they also standardized the communications protocol between the web front ends(called the memcached protocol). Now anyone who wanted to keep their database fromgetting overwhelmed with repetitive queries could use their front end tools.

• Case study: Google’s MapReduce—use commodity hardware to create search indexes

One of the most influential case studies in the NoSQL movement is the GoogleMapReduce system. In this paper, Google shared their process for transforming largevolumes of web data content into search indexes using low-cost commodity CPUs.Though sharing of this information was significant, the concepts of map and reduceweren’t new. Map and reduce functions are simply names for two stages of a datatransformation, as described in figure 11.The initial stages of the transformation are called the map operation. They’reresponsible for data extraction, transformation, and filtering of data. The results ofthe map operation are then sent to a second layer: the reduce function. The reducefunction is where the results are sorted, combined, and summarized to produce thefinal result.The core concepts behind the map and reduce functions are based on solid computerscience work that dates back to the 1950s when programmers at MIT implementedthese functions in the influential LISP system. LISP was different than otherprogramming languages because it emphasized functions that transformed isolatedlists of data. This focus is now the basis for many modern functional programminglanguages that have desirable properties on distributed systems.Google extended the map and reduce functions to reliably execute on billions ofweb pages on hundreds or thousands of low-cost commodity CPUs. Google made mapand reduce work reliably on large volumes of data and did it at a low cost. It wasGoogle's use of MapReduce that encouraged others to take another look at the powerof functional programming and the ability of functional programming systems toscale over thousands of low-cost CPUs. Software packages such as Hadoop have closelymodeled these functions.

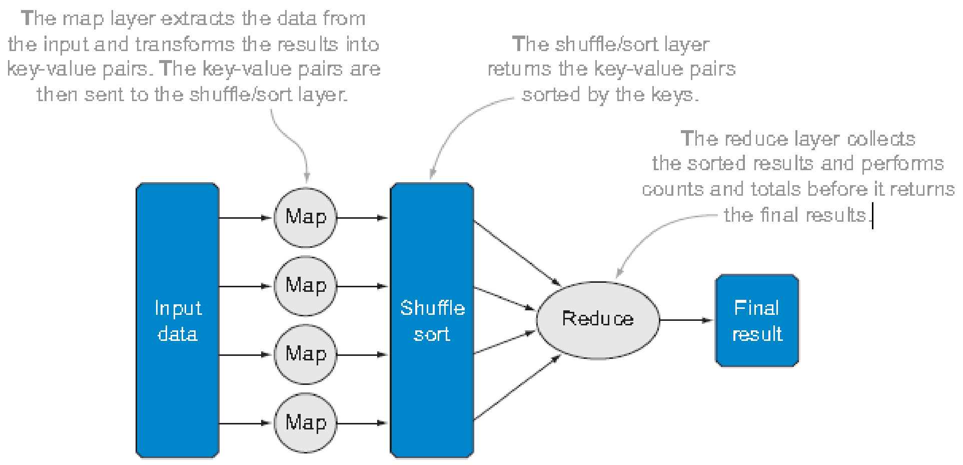

The map layer extracts the data from the input and transforms the results into key-value pairs. The key-value pairs are then sent to the shuffle/sort layer.
The shuffle/sort layer returns the key-value pairs sorted by the keys.
The reduce layer collects the sorted results and performs counts and totals before it returns the final results.
Input data
Map
Map
Map
Map
Shuffle sort
Reduce
Final result

Fig 11.The map and reduce functions are ways of partitioning large datasets intosmaller chunks that can be transformed on isolated and independent transformationsystems. The key is isolating each function so that it can be scaled onto manyservers.

The use of MapReduce inspired engineers from Yahoo! and other organizations to create open source versions of Google's MapReduce. It fostered a growing awareness of the limitations of traditional procedural programming and encouraged others to use functional programming systems.

• Case study: Google's Bigtable—a table with a billion rows and a million columns

Google also influenced many software developers when they announced their Bigtable system white paper titled A Distributed Storage System for Structured Data. The motivation behind Bigtable was the need to store results from the web crawlers that extract HTML pages, images, sounds, videos, and other media rom the internet. The resulting dataset was so large that it couldn't fit into a single relational database, so Google built their own storage system. Their fundamental goal was to build a system that would easily scale as their data increased without forcing them to purchase expensive hardware. The solution was neither a full relational database nor a filesystem, but what they called a "distributed storage system" that worked with structured data. By all accounts, the Bigtable project was extremely successful. It gave Google developers a single tabular view of the data by creating one large table that stored all the data they needed. In addition, they created a system that allowed the hardware to be located in any data center, anywhere in the world, and created an environment where developers didn’t need to worry about the physical location of the data they manipulated.

- Case study: Amazon’s Dynamo—accept an order 24 hours a day,7 days a week

Google’s work focused on ways to make distributed batch processing and reporting easier, but wasn’t intended to support the need for highly scalable web storefronts that ran 24/7. This development came from Amazon. Amazon published another significant NoSQL paper: Amazon’s 2007 Dynamo: A Highly Available Key-Value Store. The business motivation behind Dynamo was Amazon’s need to create a highly reliable web storefront that supported transactions from around the world 24 hours a day, 7 days a week, without interruption. Traditional brick-and-mortar retailers that operate in a few locations have the luxury of having their cash registers and point-of-sale equipment operating only during business hours. When not open for business, they run daily reports, and perform backups and software upgrades. The Amazon model is different. Not only are their customers from all corners of the world, but they shop at all hours of the day, every day. Any downtime in the purchasing cycle could result in the loss of millions of dollars. Amazon’s systems need to be iron-clad reliable and scalable without a loss in service. In its initial offerings, Amazon used a relational database to support its shopping cart and checkout system. They had unlimited licenses for RDBMS software and a consulting budget that allowed them to attract the best and brightest consultants fortheir projects. In spite of all that power and money, they eventually realized that a relationalmodel wouldn’t meet their future business needs.Many in the NoSQL community cite Amazon’s Dynamo paper as a significant turningpoint in the movement. At a time when relational models were still used, it challengedthe status quo and current best practices. Amazon found that because keyvaluestores had a simple interface, it was easier to replicate the data and more reliable.In the end, Amazon used a key-value store to build a turnkey system that was reliable,extensible, and able to support their 24/7 business model, making them one ofthe most successful online retailers in the world.

Case study: MarkLogic

In 2001 a group of engineers in the San Francisco Bay Area with experience in document search formed a company that focused on managing large collections of XML documents. Because XML documents contained markup, they named the company MarkLogic. MarkLogic defined two types of nodes in a cluster: query and document nodes. Query nodes receive query requests and coordinate all activities associated with executing a query. Document nodes contain XML documents and are responsible for executing queries on the documents in the local filesystem. Query requests are sent to a query node, which distributes queries to each remote server that contains indexed XML documents. All document matches are returned to the query node. When all document nodes have responded, the query result is then returned. The MarkLogic architecture, moving queries to documents rather than moving documents to the query server, allowed them to achieve linear scalability with petabytes of documents. MarkLogic found a demand for their products in US federal government systems that stored terabytes of intelligence information and large publishing entities that wanted to store and search their XML documents. Since 2001, MarkLogic has matured into a general-purpose highly scalable document store with support for ACID transactions and fine-grained, role-based access control. Initially, the primary language of MarkLogic developers was XQuery paired with REST; newer versions support Java as well as other language interfaces. MarkLogic is a commercial product that requires a software license for any datasets over 40 GB. NoSQL is associated with commercial as well as open source products that provide innovative solutions to business problems.

Q. 1 What is NoSQL?

Ans:One of the challenges with NoSQL is defining it. The term NoSQL is problematic sinceitdoesn't really describe the core themes in the NoSQL movement. The term originatedfrom a group in the Bay Area who met regularly to talk about common concernsand issues surrounding scalable open source databases, and it stuck. Descriptive or not, it seems to be everywhere: in trade press, product descriptions, and conferences. Use the term NoSQL as a way of differentiating a system froma traditional relational database management system (RDBMS).For our purpose, we define NoSQL in the following way:

NoSQL is a set of concepts that allows the rapid and efficient processing of data sets witha focus on performance, reliability, and agility.

Q. 2 Explain Applications of NoSQL
Ans:

Type	Typical usage	Examples
Key-value store—A simple data storage system that uses a key to access a value	• Image stores •Key-based filesystems • Object cache •Systems designed to scale	•Berkeley DB •Memcache •Redis •Riak •DynamoDB
Column family store—A sparse matrixsystem that uses a row and a column as keys	•Web crawler results •Big data problems that canrelax consistency rules	• Apache HBase • Apache Cassandra • Hypertable • Apache Accumulo
Graph store—For relationshipintensive problems	•Social networks • Fraud detection •Relationship-heavy data	• Neo4j • AllegroGraph •Bigdata (RDF data store) •InfiniteGraph (Objectivity)
Document store—Storing hierarchicaldata structures directly in the database	•High-variability data •Document search • Integration hubs	•MongoDB (10Gen) •CouchDB •Couchbase

	•Web content management • Publishing	•MarkLogic • eXist-db •Berkeley DB XML

References:

Anand Rajaraman and Jeff Ullman "Mining of Massive Datasets", Cambridge University Press

MODULE 4

Mining Data Streams

Syllabus

Module	Syllabus	Duration
4	**The Stream Data Model**: A Data-Stream-Management System, Examples of Stream Sources, Stream Queries, Issues in Stream Processing, Sampling Data techniques in a Stream, Filtering Streams: Bloom Filter with Analysis, Counting Distinct Elements in a Stream, Count-Distinct Problem, Flajolet-Martin Algorithm, Combining Estimates, Space Requirements, **Counting Frequent Items in a Stream,** Sampling Methods for Streams, frequent Itemsets in Decaying Windows, Counting Ones in a Window: The Cost of Exact Counts, The Datar-Gionis-Indyk-Motwani Algorithm, Query Answering in the DGIM Algorithm, Decaying Windows.	12

Introducation:
Most of the algorithms described in this book assume that we are mining a database. That is, all our data is available when and if we want it. In this chapter, 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. Another approach to summarizing a stream is to look at only a fixed-length "window" consisting of the last n elements for some (typically large) n. We then query the window as if it were a relation in a database. If there are many streams and/or n is large, we may not be able to store the entire window for every stream, so we need to summarize even the windows. We address the fundamental problem of maintaining an approximate count on the number of 1's in the window of a bit stream, while using much less space than would be needed to store the entire window itself. This technique generalizes to approximating various kinds of sums.

4.1 The Stream Data Model

Let us begin by discussing the elements of streams and stream processing. Weexplain the difference between streams and databases and the special problemsthat arise when dealing with streams. Some typical applications where thestream model applies will be examined.

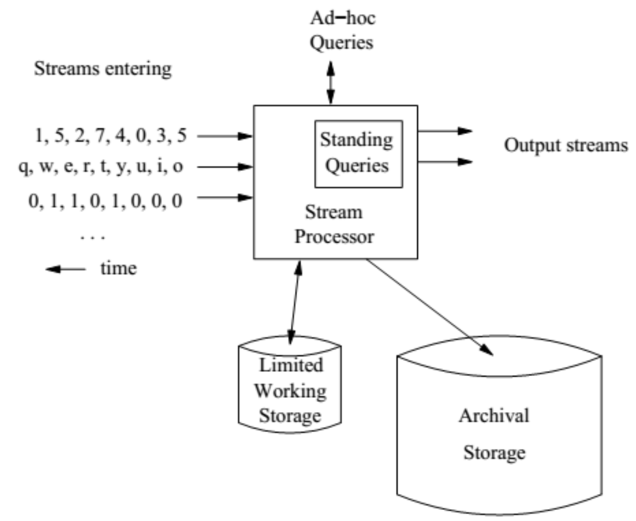


Fig- 4.1 A data-stream-management system

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. 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 underthe control of the system distinguishes stream processing from the processing of data that goes on within a database-management system. The latter system controls the rate at which data is read from the disk, and therefore never has to worry about data getting lost as it attempts to execute queries. Streams may be archived in a large archival store, but we assume it is not possible to answer queries from the archival store. It could be examined only under special circumstances using time-consuming retrieval processes. There is also a working store, into which summaries or parts of streams may be placed, and which can be used for answering queries. The working store might be disk, or it might be main memory, depending on how fast we need to process queries. But either way, it is of sufficiently limited capacity that it cannot store all the data from all the 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. 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 about ocean behavior, we might want to deploy a million sensors, each sending back a stream, at the rate of ten per second. 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. London is said to have six million such cameras, each producing a stream.

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. 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 couldindicate some news connected to that page, or it could mean that the link is broken and needs to be repaired.

Stream Queries
There are two ways that queries get asked about streams. We show in Fig. aplace within the processor where standing queries are stored. These queries are,in a sense, permanently executing, and produce outputs at appropriate times.
Example : The stream produced by the ocean-surface-temperature sensor mentioned at the beginning of Section 4.1.2 might have a standing queryto 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.Alternatively, we might have a standing query that, each time a new readingarrives, produces the average of the 24 most recent readings. That query alsocan be answered easily, if we store the 24 most recent stream elements. When anew 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 otherstanding query that requires it).Another query we might ask is the maximum temperature ever recorded bythat sensor. We can answer this query by retaining a simple summary: the maximum of all stream elements ever seen. It is not necessary to record the entire stream. 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 twovalues: 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. ✷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, then we can prepare for them by storing appropriate parts or summaries of streams as in Example. If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a sliding window of each stream in the working store. 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.

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-rocessing algorithm is executed in main memory, without access to secondary storage or with only rare accesses to secondary storage. Moreover, even when streams are "slow," as in the sensor-data example , there may be many such 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:

• Often, it is much more efficient to get an approximate answer to our problem than an exact solution.
• As in previous Chapter, 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.

4.2 Sampling Data in a Stream

As our first example of managing streaming data, we shall look at extractingreliablesamples from a stream. As with many stream algorithms, the "trick"involves using ashing 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. We shall look at a particularproblem, from which the general idea will emerge. 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.1 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.

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. Statistical fluctuations will introduce some noise into the data, but if users issue many queries, the law of large numbers will assure us that most users will have a fraction quite close to 1/10th of their queries stored. However, this scheme gives us the wrong answer to the query asking for the average number of duplicate queries for a user. 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. Of the d search queries issued twice, only d/100 will appear twice in the sample; that fraction is d times the probability that both occurrences of the query will be in the 1/10th sample. Of the queries that appear twice in the full stream, 18d/100 will appear exactly once. To see why, note that 18/100 is the probability that one of the two occurrences will be in the 1/10th of the stream that is selected, while the other is in the 9/10th that is not selected. The correct answer to the query about the fraction of repeated searches is d/(s+d). However, the answer we shall obtain from the sample is d/(10s+19d). To derive the latter formula, note that d/100 appear twice, while s/10+18d/100 appear once. Thus, the fraction appearing twice in the sample is d/100 dividedby d/100 + s/10 + 18d/100. This ratio is d/(10s + 19d). For no positive values of s and d is d/(s + d) = d/(10s + 19d).

Obtaining a Representative Sample

The query of previous section, like many 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 in the stream, we look up the user to see whether or not they are in the sample. 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. By using a hash function, one can avoid keeping thelist of users. That is, we hash each user name to one of ten buckets, 0 through9. If theuser 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. More generally, we can obtain a sample consisting of any rational fraction a/b of the users by hashing user names to b buckets, 0 through b − 1. Add the search query to the sample if the hash value is less than a.

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. 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. If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash-value. Theresult will be a sample consisting of all tuples with certain key values.The selected key values will be approximately a/b of all the key values appearing in the stream.

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. If we have a budget for how many tuples from the stream can be stored as the sample, then the fraction of key values must vary, lowering as time goes on. In order to assure that at all times, the sample consists of all tuples from a subset of the key values, we choose a hash function h from key values to a very large number of values 0, 1, . . . , B − 1. We maintain a threshold t, which initially can be the largest bucket number, B − 1. At all times, the sample consists of those tuples whose key K satisfies h(K) ≤ t. New tuples from the stream are added to the sample if and only if they satisfy the same condition. If the number of stored tuples of the sample exceeds the allotted space, we lower t to t − 1 and remove from the sample all those tuples whose key K hashes to t. For efficiency, we can lower t by more than 1, and remove the tuples with several of the highest hash values, whenever we need to throw some key values out of the sample. Further efficiency is obtained by maintaining an index on the hash value, so we can find all those tuples whose keys hash to a particular value quickly.

4.3 Filtering Streams

Another common process on streams is selection, or filtering. We want toaccept those tuples in the stream that meet a criterion. Accepted tuples arepassed to another process as a stream, while other tuples are dropped. If theselection criterion is a property of the tuple that can be calculated (e.g., thefirst component is less than 10), then the selection is easy to do. The problembecomes harder when the criterion involves lookup for membership in a set. Itis especially hard, when that set is too large to store in main memory. In this
section, we shall discuss the technique known as "Bloom filtering" as a way toeliminate most of the tuples that do not meet the criterion.
A Motivating Example
Again let us start with a running example that illustrates the problem andwhat we can do about it. Suppose we have a set S of one billion allowed emailaddresses – those that we will allow through because we believe them not tobe spam. The stream consists of pairs: an email address and the email itself.Since the typical email address is 20 bytes or more, it is not reasonable to storeS in main memory. Thus, we can either use disk accesses to determine whetheror not to let through any given stream element, or we can devise a method thatrequires no more main memory than we have available, and yet will filter mostof the undesired stream elements.Suppose for argument's sake that we have one gigabyte of available mainmemory. In the technique known as Bloom filtering, we use that main memoryas a bit array. In this case, we have room for eight billion bits, since one byte equals eight bits. Devise a hash function h from email addresses to eight billion
buckets. Hash each member of S to a bit, and set that bit to 1. All other bits of the array remain 0. Since there are one billion members of S, approximately 1/8th of the bits will be 1. The exact fraction of bits set to 1 will be slightly less than 1/8th, because it is possible that two members of S hash to the same bit. If the bit to which that email address hashes is 1, then we let the email through. But if the email address hashes to a 0, we are

certain that the address is not in S, so we can drop this stream element. Unfortunately, some spam email will get through. Approximately 1/8th of the stream elements whose email address is not in S will happen to hash to a bit whose value is 1 and will be let through. Nevertheless, since the majority of emails are spam (about 80% according to some reports), eliminating 7/8th of the spam is a significant benefit. Moreover, if we want to eliminate every spam, we need only check for membership in S those good and bad emails that get through the filter. Those checks will require the use of secondary memory to access S itself. There are also other options, as we shall see when we study the general Bloom-filtering technique. As a simple example, we could use a cascadeof filters, each of which would eliminate 7/8th of the remaining spam.

The Bloom Filter
A Bloom filter consists of:

1. An array of n bits, initially all 0's.
2. A collection of hash functions h1, h2, . . . ,hk. Each hash function maps "key" values to n buckets, corresponding to the n bits of the bit-array.
3. A set S of m key values.

The purpose of the Bloom filter is to allow through all stream elements whose keys are in S, while rejecting most of the stream elements whose keys are not in S. To initialize the bit array, begin with all bits 0. Take each key value in S and hash it using each of the k hash functions. Set to 1 each bit that is hi(K) for some hash function hi and some key value K in S. To test a key K that arrives in the stream, check that all of h1(K), h2(K), . . . , hk(K) are 1's in the bit-array. If all are 1's, then let the stream element through. If one or more of these bits are 0, then K could not be in S, so reject the stream element.

Analysis of Bloom Filtering
If a key value is in S, then the element will surely pass through the Bloom filter. However, if the key value is not in S, it might still pass. We need to understand how to calculate the probability of a false positive, as a function of n, the bit-array length, m the number of members of S, and k, the number of hash functions. The model to use is throwing darts at targets. Suppose we have x targets and y darts. Any dart is equally likely to hit any target. After throwing the darts, how many targets can we expect to be hit at least once? The analysis is similar to the analysis as follows:
•The probability that a given dart will not hit a given target is (x − 1)/x.
•The probability that none of the y darts will hit a given target is x−x 1 y. We can write this expression as (1 − x1)x(yx).
•Using the approximation (1 − ǫ)1/ǫ= 1/e for small ǫ, we conclude that the probability that none of the y darts hit a given target is e−y/x.

4.4 Counting Distinct Elements in a Stream

In this section we look at a third simple kind of processing we might want todo on a stream. As with the previous examples – sampling and filtering – it issomewhat tricky to do what we want in a reasonable amount of main memory,so we use a variety of hashing and a randomized algorithm to get approximatelywhat we want with little space needed per stream.

The Count-Distinct Problem
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: As a useful example of this problem, 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,2sequences 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 and there is little problem obtaining an exact answer to the question how many distinct elements appear in the stream. However, if the number of distinct elements is too great, or if there are too many streams that need to be processed at once (e.g., Yahoo! wants to count the number of unique users viewing each of its pages in a month), then we cannot store the needed data in main memory. There are several options. We
could use more machines, each machine handling only one or several of the streams. We could store most of the data structure in secondary memory and batch stream elements so whenever we brought a disk block to main memory there would be many tests and updates to be performed on the data in that block. Or we could use the strategy to be discussed in this section, where weonly estimate the number of distinct elements but use much less memory than the number of distinct elements.

Frequent Itemsets

Definition of Frequent Itemsets
Intuitively, a set of items that appears in many baskets is said to be “frequent.”To be formal, we assume there is a number s, called the support threshold. IfI is a set of items, the support for I is the number of baskets for which I is asubset. We say I is frequent if its support is s or more.
Handling Larger Datasets in Main Memory
The A-Priori Algorithm is fine as long as the step with the greatest requirementfor main memory – typically the counting of the candidate pairs C2 – has enoughmemory that it can be accomplished without thrashing (repeated moving ofdata between disk and main memory). Several algorithms have been proposedto cut down on the size of candidate set C2. Here, we consider the PCYAlgorithm, which takes advantage of the fact that in the first pass of A-Priorithere is typically lots of main memory not needed for the counting of singleitems. Then we look at the Multistage Algorithm, which uses the PCY trickand also inserts extra passes to further reduce the size of C2.

The Algorithm of Park, Chen, and Yu
This algorithm, which we call PCY after its authors, exploits the observationthat there may be much unused space in main memory on the first pass. If thereare a million items and gigabytes of main memory, we do not need more than10% of the main memory for the two tables suggested in Fig. 6.3 – a translationtable from item names to small integers and an array to count those integers.The PCY Algorithm

uses that space for an array of integers that generalizesthe idea of a Bloom filter (see Section 4.3). The idea is shown schematically inFig. 6.5.Think of this array as a hash table, whose buckets hold integers rather thansets of keys (as in an ordinary hash table) or bits (as in a Bloom filter). Pairs of
items are hashed to buckets of this hash table. As we examine a basket duringthe first pass, we not only add 1 to the count for each item in the basket, but we generate all the pairs, using a double loop. We hash each pair, and we add1 to the bucket into which that pair hashes. Note that the pair itself doesn'tgo into the bucket; the pair only affects the single integer in the bucket.At the end of the first pass, each bucket has a count, which is the sum ofthe counts of all the pairs that hash to that bucket. If the count of a bucketis at least as great as the support threshold s, it is called a frequent bucket.We can say nothing about the pairs that hash to a frequent bucket; they couldall be frequent pairs from the information available to us. But if the count ofthe bucket is less than s (an infrequent bucket), we know no pair that hashesto this bucket can be frequent, even if the pair consists of two frequent items.That fact gives us an advantage on the second pass. We can define the set ofcandidate pairs C2 to be those pairs {i, j} such that:
1. i and j are frequent items.
2. {i, j} hashes to a frequent bucket.
It is the second condition that distinguishes PCY from A-Priori.

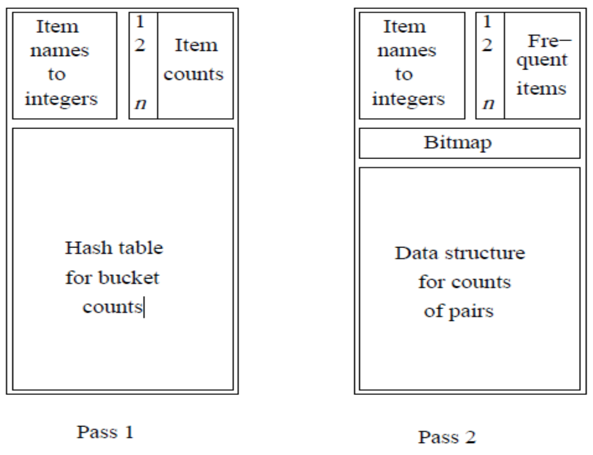


Figure 4.2: Organization of main memory for the first two passes of the PCYAlgorithm

The Multistage Algorithm:
The Multistage Algorithm improves upon PCY by using several successive hashtables to reduce further the number of candidate pairs. The tradeoff is that6.3. HANDLING LARGER DATASETS IN MAIN MEMORY 221Multistage takes more than two passes to find the frequent pairs. An outline ofthe Multistage Algorithm is shown in Fig. 6.6.

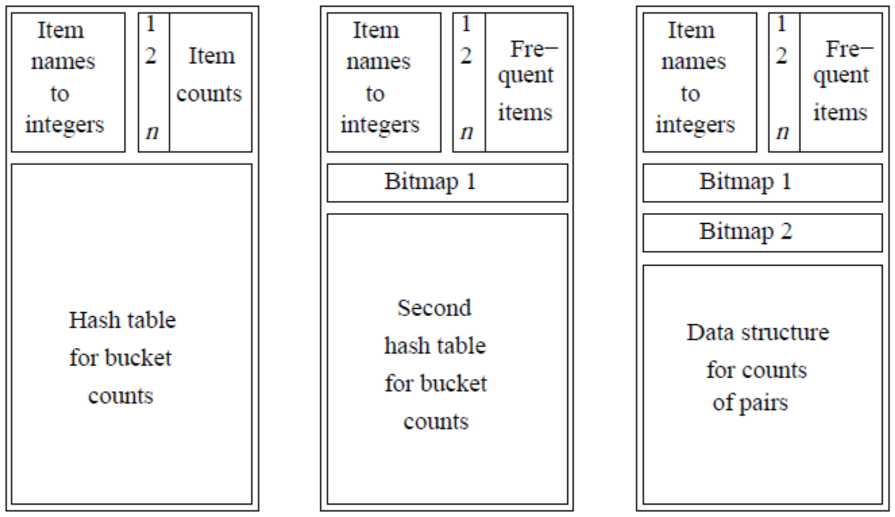

Item names to integers
1
2
n
Item counts
Hash table for bucket counts
Fre–quent items
Bitmap 1
Second hash table for bucket counts
Bitmap 2
Data structure for counts of pairs

Figure 4.3: The Multistage Algorithm uses additional hash tables to reduce thenumber of candidate pairs

The first pass of Multistage is the same as the first pass of PCY. After thatpass, the frequent buckets are identified and summarized by a bitmap, againthe same as in PCY. But the second pass of Multistage does not count thecandidate pairs. Rather, it uses the available main memory for another hashtable, using another hash function. Since the bitmap from the first hash tabletakes up 1/32 of the available main memory, the second hash table has almostas many buckets as the first.On the second pass of Multistage, we again go through the file of baskets.There is no need to count the items again, since we have those counts fromthe first pass. However, we must retain the information about which items arefrequent, since we need it on both the second and third passes. During the
second pass, we hash certain pairs of items to buckets of the second hash table.A pair is hashed only if it meets the two criteria for being counted in the secondpass of PCY; that is, we hash {i, j} if and only if i and j are both frequent,and the pair hashed to a frequent bucket on the first pass. As a result, the sumof the counts in the second hash table should be significantly less than the sumfor the first pass. The result is that, even though the second hash table hasonly 31/32 of the number of buckets that the first table has, we expect thereto be many fewer frequent buckets in the second hash table than in the first.After the second pass, the second hash table is also summarized as a bitmap,and that bitmap is stored in main memory.The two bitmaps together take up slightly less than 1/16th of the available main memory, so there is still plentyof space to count the candidate pairs on the third pass. A pair {i, j} is in C2 ifand only if:

1. i and j are both frequent items.
2. {i, j} hashed to a frequent bucket in the first hash table.
3. {i, j} hashed to a frequent bucket in the second hash table.

The third condition is the distinction between Multistage and PCY.It might be obvious that it is possible to insert any number of passes betweenthe first and last in the multistage Algorithm. There is a limiting factor thateach pass must store the bitmaps from each of the previous passes. Eventually,there is not enough space left in main memory to do the counts. No matterhow many passes we use, the truly frequent pairs will always hash to a frequentbucket, so there is no way to avoid counting them.

The Multihash Algorithm

Sometimes, we can get most of the benefit of the extra passes of the MultistageAlgorithm in a single pass. This variation of PCY is called the MultihashAlgorithm. Instead of using two different hash tables on two successive passes,use two hash functions and two separate hash tables that share main memoryon the first pass, as suggested by Fig. 6.7.The danger of using two hash tables on one pass is that each hash table hashalf as many buckets as the one large hash table of PCY. As long as the averagecount of a bucket for PCY is much lower than the support threshold, we canoperate two half-sized hash tables and still expect most of the buckets of bothhash tables to be infrequent. Thus, in this situation we might well choose themultihash approach.

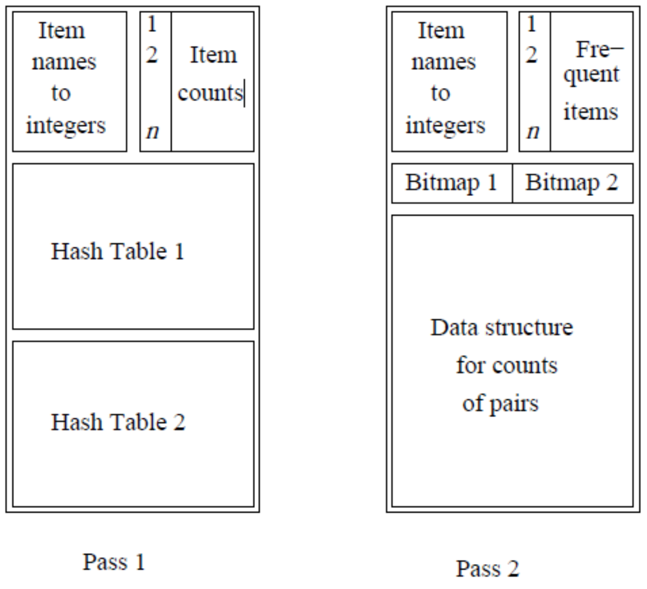


Figure 4.4: The Multihash Algorithm uses several hash tables in one pass

The SON Algorithm and MapReduce
The SON algorithm lends itself well to a parallel-computing environment. Eachof the chunks can be processed in parallel, and the frequent itemsets from eachchunk combined to form the candidates. We can distribute the candidates tomany processors, have each processor count the support for each candidatein a subset of the baskets, and finally sum those supports to get the supportfor each candidate itemset in the whole dataset. This process does not haveto be implemented in MapReduce, but there is a natural way of expressingeach of the two passes as a MapReduce operation. We shall summarize thisMapReduce-MapReduce sequence below.First Map Function: Take the assigned subset of the baskets and find theitemsets frequent in the subset using the algorithm of Section 6.4.1. As describedthere, lower the support threshold from s to ps if each Map task getsfraction p of the total input file. The output is a set of key-value pairs (F, 1),where F is a frequent itemset from the sample. The value is always 1 and isirrelevant.First Reduce Function: Each Reduce task is assigned a set of keys, whichare itemsets. The value is ignored, and the Reduce task simply produces thosekeys (itemsets) that appear one or more times. Thus, the output of the firstReduce function is the candidate itemsets.Second Map Function: The Map tasks for the second Map function takeall the output from the first Reduce Function (the candidate itemsets) and aportion of the input data file. Each Map task counts the number of occurrencesof each of the candidate itemsets among the baskets in the portion of the datasetthat it was assigned. The output is a set of key-value pairs (C, v), where C is oneof the candidate sets and v is the support for that itemset among the basketsthat were input to this Map task.Second Reduce Function: The Reduce tasks take the itemsets they aregiven as keys and sum the associated values. The result is the total supportfor each of the itemsets that the Reduce task was assigned to handle. Thoseitemsets whose sum of values is at least s are

frequent in the whole dataset, the Reduce task outputs these itemsets with their counts. Itemsets that do nothave total support at least s are not transmitted to the output of the Reducetask.2

Counting Frequent Items in a Stream

Suppose that instead of a file of baskets we have a stream of baskets, fromwhich we want to mine the frequent itemsets. Recall from Chapter 4 that thedifference between a stream and a data file is that stream elements are onlyavailable when they arrive, and typically the arrival rate is so great that wecannot store the entire stream in a way that allows easy querying. Further, itis common that streams evolve over time, so the itemsets that are frequent intoday's stream may not be frequent tomorrow.A clear distinction between streams and files, when frequent itemsets areconsidered, is that there is no end to a stream, so eventually an itemset is goingto exceed the support threshold, as long as it appears repeatedly in the stream.As a result, for streams, we must think of the support threshold s as a fraction ofthe baskets in which an itemset must appear in order to be considered frequent.Even with this adjustment, we still have several options regarding the portionof the stream over which that fraction is measured.In this section, we shall discuss several ways that we might extract frequentitemsets from a stream. First, we consider ways to use the sampling techniquesof the previous section. Then, we consider the decaying-window modelfrom Section 4.7, and extend the method described in Section 4.7.3 for finding"popular" items.

Sampling Methods for Streams:

In what follows, we shall assume that stream elements are baskets of items.Perhaps the simplest approach to maintaining a current estimate of the frequentitemsets in a stream is to collect some number of baskets and store it as a file.Run one of the frequent-itemset algorithms discussed in this chapter, meanwhileignoring the stream elements that arrive, or storing them as another file to beanalyzed later. When the frequent-itemsets algorithm finishes, we have anestimate of the frequent itemsets in the stream. We then have several options.

1. We can use this collection of frequent itemsets for whatever application isat hand, but start running another iteration of the chosen frequent-itemsetalgorithm immediately. This algorithm can either:

(a) Use the file that was collected while the first iteration of the algorithmwas running. At the same time, collect yet another file to be
used at another iteration of the algorithm, when this current iterationfinishes.

(b) Start collecting another file of baskets now, and run the algorithmwhen an adequate number of baskets has been collected.

2. We can continue to count the numbers of occurrences of each of thesefrequent temsets, along with the total number of baskets seen in thestream, since the counting started. If any itemset is discovered to occurin a fraction of the baskets that is significantly below the threshold fractions, then this set can be dropped from the collection of frequent itemsets.When computing the fraction, it is important to include the occurrencesfrom the original file of baskets, from which the frequent itemsets werederived. If not, we run the risk that we shall encounter a short period inwhich a truly frequent itemset does not appear sufficiently frequently andthrow it out. We should also allow some way for new frequent itemsetsto be added to the current collection.

Possibilities include:

(a) Periodically gather a new segment of the baskets in the stream anduse it as the data file for another iteration of the chosen frequentitemsetsalgorithm. The new collection of frequent items is formedfrom the result of this iteration and the frequent itemsets from the
previous collection that have survived the possibility of having beendeleted for becoming infrequent.

(b) Add some random itemsets to the current collection, and count theirfraction of occurrences for a while, until one has a good idea ofwhether or not they are currently frequent. Rather than choosingnew itemsets completely at random, one might focus on sets withitems that appear in many itemsets already known to be frequent.For example, a good choice is to pick new itemsets from the negativeborder (Section 6.4.5) of the current set of frequent itemsets.

Frequent Itemsets in Decaying Windows
Recall from Section that a decaying window on a stream is formed by pickinga small constant c and giving the ith element prior to the most recent elementthe weight (1 − c)i, or approximately e−ci. Section 4.7.3 actually presented amethod for computing the frequent items, provided the support threshold isdefined in a somewhat different way. That is, we considered, for each item, astream that had 1 if the item appeared at a certain stream element and 0 ifnot. We defined the "score" for that item to be the sum of the weights whereits stream element was 1. We were constrained to record all items whose scorewas at least 1/2. We can not use a score threshold above 1, because we do notinitiate a count for an item until the item appears in the stream, and the firsttime it appears, its score is only 1 (since 1, or (1 − c)0, is the weight of thecurrent item).If we wish to adapt this method to streams of baskets, there are two modificationswe must make. The first is simple. Stream elements are baskets ratherthan individual items, so many items may appear at a given stream element.Treat each of those items as if they were the "current" item and add 1 to theirscore after multiplying all current scores by 1−c, as described in Section 4.7.3.
If some items in a basket have no current score, initialize the scores of thoseitems to 1.The second modification is trickier. We want to find all frequent itemsets,not just singleton itemsets. If we were to initialize a count for an itemsetwhenever we saw it, we would have too many counts. For example, one basketof 20 items has over a million subsets, and all of these would have to be initiatedfor one basket. On the other hand, as we mentioned, if we use a requirementabove 1 for initiating the scoring of an itemset, then we would never get anyitemsets started, and the method would not work.A way of dealing with this problem is to start scoring certain itemsets assoon as we see one instance, but be conservative about which itemsets we start.We may borrow from the A-Priori trick, and only start an itemset I if all itsimmediate proper subsets are already being scored. The consequence of thisrestriction is that if I is truly frequent, eventually we shall begin to count it,but we never start an itemset unless it would at least be a candidate in thesense used in the A-Priori Algorithm.

4.5 Counting Ones in a Window

We now turn our attention to counting problems for streams. Suppose we have a window of length N on a binary stream. We want at all times to be able to answer queries of the form "how many 1's are there in the last k bits?" for any k ≤ N. As in previous sections, we focus on the situation where we cannot afford to store the entire window. After showing an approximate algorithm for the binary case, we discuss how this idea can be extended to summing numbers.

The Cost of Exact Counts

To begin, suppose we want to be able to count exactly the number of 1's in the last k bits for any k ≤ N. Then we claim it is necessary to store all N bits of the window, as any representation that used fewer than

N bits couldnot work. In proof, suppose we have a representation that uses fewer than N bits to represent the N bits in the window. Since there are 2N sequences of N bits, but fewer than 2N representations, there must be two different bit strings w and x that have the same representation. Since w 6= x, they must differ in at least one bit. Let the last k − 1 bits of w and x agree, but let them differ on the kth bit from the right end.

Example: If w = 0101 and x = 1010, then k = 1, since scanning from the right, they first disagree at position 1. If w = 1001 and x = 0101, then k = 3, because they first disagree at the third position from the right.Suppose the data representing the contents of the window is whatever sequence of bits represents both w and x. Ask the query "how many 1's are in the last k bits?" The query-answering algorithm will produce the same answer, whether the window contains w or x, because the algorithm can only see their representation. But the correct answers are surely different for these two bit-strings. Thus, we have proved that we must use at least N bits to answer queries about the last k bits for any k. In fact, we need N bits, even if the only query we can ask is "how many 1's are in the entire window of length N?" The argument is similar to that used above. Suppose we use fewer than N bits to represent the window, and therefore we can find w, x, and k as above. It might be that w and x have the same number of 1's, as they did in both cases of Example 4.10. However, if we follow the current window by any N − k bits, we will have a situationwhere the true window contents resulting from w and x are identical except for the leftmost bit, and therefore, their counts of 1's are unequal. However, since the representations of w and x are the same, the representation of the window must still be the same if we feed the same bit sequence to these representations. Thus, we can force the answer to the query "how many 1's in the window?" to be incorrect for one of the two possible window contents.

References:

Anand Rajaraman and Jeff Ullman "Mining of Massive Datasets", Cambridge University Press

Objective Question (minimum 10-15) with answers.

Ques 1.Data mining can also applied to other forms such as ....

i) Data streams

ii) Sequence data

iii) Networked data

iv) Text data

v) Spatial data

A) i, ii, iii and v only

B) ii, iii, iv and v only

C) i, iii, iv and v only

D) All i, ii, iii, iv and v

Ans: D) All i, ii, iii, iv and v

Quse. 2 ____________ is the process of finding a model that describes and distinguishes data classes or concepts.

a) Data characterization
b) Data classification
c) Data discrimination
d) Data selection

Ans: Data Classification

Subjective Question

Q.1 What is Mining of data streams?

Most of the algorithms described in this book assume that we are mining a database. That is, all our data is available when and if we want it. In this chapter, 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. Another approach to summarizing a stream is to look at only a fixed-length "window" consisting of the last n elements for some (typically large) n. We then query the window as if it were a relation in a database. If there are many streams and/or n is large, we may not be able to store the entire window for every stream, so we need to summarize even the windows. We address the fundamental problem of maintaining an approximate count on the number of 1's in the window of a bit stream, while using much less space than would be needed to store the entire window itself. This technique generalizes to approximating various kinds of sums.

Q.2 What is data stream?

Q. 3 Define Distance measure.

University Questions

Q.1. How to count distinct elements in a stream? Explain Flajolet-martin algorithm.

Q.2. What is data stream Management System? Explain with block diagram

Q.3 Explain Collaborative Filtering based recommendation System. How it is different form content based recommendation systems?

Q. 4 With respect to data stream querying, give example of

a) One time queries

b)Continuous queries

c)Pre-defined queries

d)Ad-hoc queries

MODULE 5
Finding Similar Items

Motivation:

In this chapter the main motivation is to examine the fundamental data-mining problem. We will begin by phrasing the problem of similarity as one of finding sets with a relatively large intersection.

Learning Objectives:

- Understand to find similarities between items in large dataset.
- Perceive similarities as form of nearest neighbor search.
- Learn and understand the concept of a data stream.
- Learn about several popular stream-based algorithms like counting Distinct elements in a stream, counting ones in a window.

Syllabus:

Module	Syllabus	Duration
5	Finding Similar Items 5.1 Applications of Near-Neighbor Search, Jaccard Similarity of Sets, Similarity of Documents, Collaborative Filtering as a Similar-Sets Problem. 5.2Distance Measures: Definition of a Distance Measure, Euclidean Distances, Jaccard Distance, Cosine Distance, Edit Distance, Hamming Distance. 5.2 CURE Algorithm, Stream-Computing , A Stream-Clustering Algorithm, Initializing & Merging Buckets, Answering Queries	08

Learning Outcomes:

At the end of this Module, students will possess the following:

- Learn and understand different similarity and distance measures like Euclidean measures, Jaccard measures among others.
- Apply the appropriate distance measure to the given application.

- Understand the importance of sampling and get introduced to several stream sampling techniques.

A fundamental data-mining problem is to examine data for “similar” items. We shall take up applications in Section 3.1, but an example would be looking at acollection of Web pages and finding near-duplicate pages. These pages could be plagiarisms, for example, or they could be mirrors that have almost the same content but differ in information about the host and about other mirrors. We begin by phrasing the problem of similarity as one of finding sets witha relatively large intersection. We show how the problem of finding textually similar documents can be turned into such a set problem by the technique known as “shingling.” Then, we introduce a technique called “minhashing,” which compresses large sets in such a way that we can still deduce the similarity ofthe underlying sets from their compressed versions. Another important problem that arises when we search for similar items ofany kind is that there may be far too many pairs of items to test each pair fortheir degree of similarity, even if computing the similarity of any one pair can bemade very easy. That concern motivates a technique called “locality-sensitivehashing,” for focusing our search on pairs that are most likely to be similar.Finally, we explore notions of “similarity” that are not expressible as intersection of sets. This study leads us to consider the theory of distance measuresin arbitrary spaces. It also motivates a general framework for locality-sensitivehashing that applies for other definitions of “similarity.”

5.1 Applications of Near-Neighbor Search

We shall focus initially on a particular notion of “similarity”: the similarity of sets by looking at the relative size of their intersection. This notion of similarity is called “Jaccard similarity,” We then examine some of the uses of finding similar sets. These include finding textually similar documents and collaborative filtering by finding similar customers and similar products. In order to turn the problem of textual similarity of documents into one of set intersection, we use a technique called “shingling,”

5.2 Jaccard Similarity of Sets

The Jaccard similarity of sets S and T is $| S \cap T | / | S \cup T |$, that is, the ratio of the size of the intersection of S and T to the size of their union. We shall denote the Jaccard similarity of S and T by SIM(S, T).Example 3.1: In Fig. 3.1 we see two sets S and T. There are three elements in their intersection and a total of eight elements that appear in S or T or both. Thus, SIM(S, T) = 3/8.

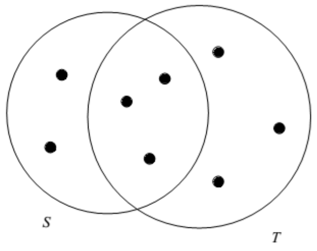


Fig- Two sets with Jaccard similarity 3/8

5.3 Similarity of Documents

An important class of problems that Jaccard similarity addresses well is that of finding textually similar documents in a large corpus such as the Web or a collection of news articles. We should understand that the aspect of similarity we are looking at here is character-level similarity, not “similar meaning,” which requires us to examine the words in the documents and their uses. That problem is also interesting but is addressed by other techniques. However, textual similarity also has important uses. Many of these involve finding duplicates or near duplicates. First, let us observe that testing whether two documents are exact duplicates is easy; just compare the two documents character-by-character, and if they ever differ then they are not the same. However, in many applications, the documents are not identical, yet they share large portions of their text. Here are some examples:

Plagiarism

Finding plagiarized documents tests our ability to find textual similarity. The plagiarizer may extract only some parts of a document for his own. He may alter a few words and may alter the order in which sentences of the original appear. Yet the resulting document may still contain 50% or more of the original. No simple process of comparing documents character by character will detect a sophisticated plagiarism.

Mirror Pages

It is common for important or popular Web sites to be duplicated at a number of hosts, in order to share the load. The pages of these mirror sites will be quite similar, but are rarely identical. For instance, they might each contain information associated with their particular host, and they might each have links to the other mirror sites but not to themselves. A related phenomenon is the appropriation of pages from one class to another. These pages might include class notes, assignments, and lecture slides. Similar pages might change the name of the course, year, and make small changes from year to year. It is important to be able to detect similar pages of these kinds, because search engines produce better results if they avoid showing two pages that are nearly identical within the first page of results.

Articles from the Same Source

It is common for one reporter to write a news article that gets distributed, say through the Associated Press, to many newspapers, which then publish the article on their Web sites. Each newspaper changes the article somewhat. They may cut out paragraphs, or even add material of their own. They mostlikely will surround the article by their own logo, ads, and links to other articles at their site. However, the core of each newspaper’s page will be the original article. News aggregators, such as Google News, try to find all versions of such an article, in order to show only one, and that task requires finding when two Web pages are textually similar, although not identical.

5.4 Collaborative Filtering as a Similar-Sets Problem

Another class of applications where similarity of sets is very important is calledcollaborative filtering, a process whereby we recommend to users items that were liked by other users who have exhibited similar tastes.

On-Line Purchases
Amazon.com has millions of customers and sells millions of items. Its database records which items have been bought by which customers. We can say two customers are similar if their sets of purchased items have a high Jaccard similarity. Likewise, two items that have sets of purchasers with high Jaccard similaritywill be deemed similar. Note that, while we might expect mirror sites to have Jaccard similarity above 90%, it is unlikely that any two customers have Jaccard similarity that high (unless they have purchased only one item). Even a Jaccard similarity like 20% might be unusual enough to identify customers with similar tastes. The same observation holds for items; Jaccard similarities need not be very high to be significant. Collaborative filtering requires several tools, in addition to finding similar customers or items, as we discuss in Chapter 9. For example, two Amazon customers who like science-fiction might each buy many science-fiction books,
but only a few of these will be in common. However, by combining similarityfinding with clustering (Chapter 7), we might be able to discover that sciencefiction books are mutually similar and put them in one group. Then, we can get a more powerful notion of customer similarity by asking whether they made purchases within many of the same groups.

Movie Ratings
NetFlix records which movies each of its customers rented, and also the ratings assigned to those movies by the customers. We can see movies as similar if they were rented or rated highly by many of the same customers, and see customers as similar if they rented or rated highly many of the same movies. The same observations that we made for Amazon above apply in this situation: similarities need not be high to be significant, and clustering movies by genre will make things easier. When our data consists of ratings rather than binary decisions (bought/did not buy or liked/disliked), we cannot rely simply on sets as representations of customers or items. Some options are:

1. Ignore low-rated customer/movie pairs; that is, treat these events as if the customer never watched the movie.
2. When comparing customers, imagine two set elements for each movie, “liked” and “hated.” If a customer rated a movie highly, put the “liked” for that movie in the customer’s set. If they gave a low rating to a movie, put “hated” for that movie in their set. Then, we can look for high Jaccard similarity among these sets. We can do a similar trick when comparing movies.
3. If ratings are 1-to-5-stars, put a movie in a customer’s set n times if they rated the movie n-stars. Then, use Jaccard similarity for bags when measuring the similarity of customers. The Jaccard similarity for bagsB and C is defined by counting an element n times in the intersection if n is the minimum of the number of times the element appears in B and C. In the union, we count the element the sum of the number of times it appears in B and in C.

Example: The bag-similarity of bags {a, a, a, b} and {a, a, b, b, c} is 1/3. The intersection counts a twice and b once, so its size is 3. The size of the union of two bags is always the sum of the sizes of the two bags, or 9 in this case. Since the highest possible Jaccard similarity for bags is 1/2, the score of 1/3 indicates the two bags are quite similar, as should be apparent from an examination of their contents.

5.5 Distance Measures: Definition of a Distance Measure

We now take a short detour to study the general notion of distance measures. The Jaccard similarity is a measure of how close sets are, although it is not really a distance measure. That is, the closer sets are, the higher the Jaccardsimilarity. Rather, 1 minus the Jaccard similarity is a distance measure, as we shall see; it is called the Jaccard distance. However, Jaccard distance is not the only measure of closeness that makes sense. We shall examine in this section some other distance measures that have applications. Then, how some of these distance measures also have an LSH technique that allows us to focus on nearby points without comparing all points.

Definition of a Distance Measure:
Suppose we have a set of points, called a space. A distance measure on this space is a function d(x, y) that takes two points in the space as arguments and produces a real number, and satisfies the following axioms:

1. d(x, y) ≥ 0 (no negative distances).
2. d(x, y) = 0 if and only if x = y (distances are positive, except for the distance from a point to itself).
3. d(x, y) = d(y, x) (distance is symmetric).
4. d(x, y) ≤ d(x, z) + d(z, y) (the triangle inequality).

The triangle inequality is the most complex condition. It says, intuitively, that to travel from x to y, we cannot obtain any benefit if we are forced to travel via some particular third point z. The triangle-inequality axiom is what makes all distance measures behave as if distance describes the length of a shortest path from one point to another.

5.6 Euclidean Distances

The most familiar distance measure is the one we normally think of as "distance." An n-dimensional Euclidean space is one where points are vectors of n real numbers. The conventional distance measure in this space, which we shall refer to as the L2-norm, is defined:

$$d([x_1, x_2, \ldots, x_n],\ [y_1, y_2, \ldots, y_n]) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

That is, we square the distance in each dimension, sum the squares, and take the positive square root. It is easy to verify the first three requirements for a distance measure are satisfied. The Euclidean distance between two points cannot be negative, because the positive square root is intended. Since all squares of real numbers are nonnegative, any i such that xi 6= yi forces the distance to be strictly positive. On the other hand, if xi = yi for all i, then the distance is clearly 0. Symmetry follows because (xi − yi)2 = (yi − xi)2. The triangle inequality requires a good deal of algebra to verify. However, it is well understood to be a property of Euclidean space: the sum of the lengths of any two sides of a triangle is no less than the length of the third side. There are other distance measures that have been used for Euclidean spaces. For any constant r, we can define the Lr-norm to be the distance measure d defined by:

$$d([x_1, x_2, \ldots, x_n],\ [y_1, y_2, \ldots, y_n]) = (\sum_{i=1}^{n} |x_i - y_i|^r)^{1/r}$$

The case r = 2 is the usual L2-norm just mentioned. Another common distance measure is the L1-norm, or Manhattan distance. There, the distance between two points is the sum of the magnitudes of the differences in each dimension. It is called "Manhattan distance" because it is the distance one would have totravel between points if one were constrained to travel along grid lines, as on the streets of a city such as Manhattan. Another interesting distance measure is the L∞-norm, which is the limit as r approaches

infinity of the Lr-norm. As r gets larger, only the dimension with the largest difference matters, so formally, the L∞-norm is defined as the maximum of | xi − yi| over all dimensions i.
Example: Consider the two-dimensional Euclidean space (the customary plane) and the points (2, 7) and (6, 4). The L2-norm gives a distance of p(2 − 6)2 + (7 − 4)2 = √42 + 32 = 5. The L1-norm gives a distance of|2 − 6| + |7 − 4| = 4 + 3 = 7. The L∞-norm gives a distance of

max(|2 − 6| , |7 − 4|) = max(4, 3) = 4

5.7 Jaccard Distance

As mentioned at the beginning of the section, we define the Jaccard distance of sets by d(x, y) = 1 − SIM(x, y). That is, the Jaccard distance is 1 minus theratio of the sizes of the intersection and union of sets x and y. We must verifythat this function is a distance measure.
1. d(x, y) is nonnegative because the size of the intersection cannot exceed the size of the union.
2. d(x, y) = 0 if x = y, because x ∪x = x ∩ x = x. However, if x 6= y, then the size of x ∩ y is strictly less than the size of x ∪y, so d(x, y) is strictly positive.
3. d(x, y) = d(y, x) because both union and intersection are symmetric; i.e., x ∪y = y ∪x and x ∩ y = y ∩ x.
4. For the triangle inequality, recall from Section 3.3.3 that SIM(x, y) is the probability a random minhash function maps x and y to the same value. Thus, the Jaccard distance d(x, y) is the probability that a random minhash function does not send x and y to the same value. We can therefore translate the condition d(x, y) ≤ d(x, z) + d(z, y) to the statement that if h is a random minhash function, then the probability that h(x) 6= h(y) is no greater than the sum of the probability that h(x) 6= h(z) and the probability that h(z) 6= h(y). However, this statement is true because whenever h(x) 6= h(y), at least one of h(x) and h(y) must be different from h(z). They could not both be h(z), because then h(x) and h(y) would be the same.

5.8 Cosine Distance

The cosine distance makes sense in spaces that have dimensions, including Euclidean spaces and discrete versions of Euclidean spaces, such as spaces where points are vectors with integer components or boolean (0 or 1) components. In such a space, points may be thought of as directions. We do not distinguish between a vector and a multiple of that vector. Then the cosine distance between two points is the angle that the vectors to those points make. This angle will be in the range 0 to 180 degrees, regardless of how many dimensions the space has. We can calculate the cosine distance by first computing the cosine of the angle, and then applying the arc-cosine function to translate to an angle in the 0-180 degree range. Given two vectors x and y, the cosine of the angle between them is the dot product x.y divided by the L2-norms of x and y (i.e., their Euclidean distances from the origin). Recall that the dot product of vectors [x1 , x2, . . . , xn] .[y1 , y2, . . . , yn] is Pni=1 xiyi.

Example : Let our two vectors be x = [1, 2, −1] and = [2, 1, 1]. The dot product x.y is 1 × 2 + 2 × 1 + (−1) × 1 = 3. The L2-norm of both vectors is √6. For example, x has L2-norm p12 + 22 + (−1)2 = √6. Thus, the cosine of the angle between x and y is 3/(√6√6) or 1/2. The angle whose cosine is ½ is 60 degrees, so that is the cosine distance between x and y. We must show that the cosine distance is indeed a distance measure. We have defined it so the values are in the range 0 to 180, so no negative distances are possible. Two vectors have angle 0 if and only if they are the same direction.4 Symmetry is obvious: the angle between x and y is the same as the angle between y and x. The triangle inequality is best argued by physical reasoning. One way to rotate from x to y is to rotate to z and thence to y. The sum of those two rotations cannot be less than the rotation directly from x to y.

5.9 Edit Distance

This distance makes sense when points are strings. The distance between two strings x = x1 x2 · · · xn and y = y1y2 · · · ym is the smallest number of insertions and deletions of single characters that will convert x to y.
Example : The edit distance between the strings x = abcde and y = acfdeg is 3. To convert x to y:
1. Delete b.
2. Insert f after c.
3. Insert g after e.
No sequence of fewer than three insertions and/or deletions will convert x to y. Thus, d(x,y) = 3.Another way to define and calculate the edit distance d(x,y) is to compute a longest common subsequence (LCS) of x and y. An LCS of x and y is a string that is constructed by deleting positions from x and y, and that is as long as any string that can be constructed that way. The edit distance d(x,y) can be calculated as the length of x plus the length of y minus twice the length of their LCS.

5.10 Hamming Distance

Given a space of vectors, we define the Hamming distance between two vectors to be the number of components in which they differ. It should be obvious that Hamming distance is a distance measure. Clearly the Hamming distance cannot be negative, and if it is zero, then the vectors are identical. The distance does not depend on which of two vectors we consider first. The triangle inequality should also be evident. If x and z differ in m components, and z and y differ in n components, then x and y cannot differ in more than m+ncomponents. Most commonly, Hamming distance is used when the vectors are boolean; they consist of 0's and 1's only. However, in principle, the vectors can have components from any set.

Example: The Hamming distance between the vectors 10101 and 11110 is 3. That is, these vectors differ in the second, fourth, and fifth components, while they agree in the first and third components.

What is Clustering?

Clustering is the process of examining a collection of "points," and groupingthe points into "clusters" according to some distance measure. The goal is thatpoints in the same cluster have a small distance from one another, while pointsin different clusters are at a large distance from one another. A suggestion ofwhat clusters might look like was seen in Fig. 1.1. However, there the intentwas that there were three clusters around three different road intersections, buttwo of the clusters blended into one another because they were not sufficientlyseparated.

We now turn to another large-scale-clustering algorithm in the point-assignmentclass. This algorithm, called CURE (Clustering Using REpresentatives), assumesa Euclidean space. However, it does not assume anything about theshape of clusters; they need not be normally distributed, and can even havestrange bends, S-shapes, or even rings. Instead of representing clusters by theircentroid, it uses a collection of representative points, as the name implies.

Initialization in CURE

We begin the CURE algorithm by:
1. Take a small sample of the data and cluster it in main memory. In principle,any clustering method could be used, but as CURE is designed tohandle oddly shaped clusters, it is often advisable to use a hierarchicalmethod in which clusters are merged when they have a close pair of points.This issue is discussed in more detail in Example 7.11 below.
2. Select a small set of points from each cluster to be representative points.These points should be chosen to be as far from one another as possible,using the method described in Section 7.3.2.
3. Move each of the representative points a fixed fraction of the distancebetween its location and the centroid of its cluster. Perhaps 20% is agood fraction to choose. Note that this step requires a Euclidean space,since otherwise, there might not be any notion of a line between twopoints.

Stream-Computing

A Stream- Clustering Algorithm :In this section, we shall present a greatly simplified version of an algorithmreferred to as BDMO (for the authors, B. Babcock, M. Datar, R. Motwani,and L. O'Callaghan). The true version of the algorithm involves much morecomplex structures, which are designed to provide performance guarantees in
the worst case.The BDMO Algorithm builds on the methodology for counting ones in astream that was described in Section 4.6. Here are the key similarities and
differences:
• Like that algorithm, the points of the stream are partitioned into, andsummarized by, buckets whose sizes are a power of two. Here, the size ofa bucket is the number of points it represents, rather than the number ofstream elements that are
1.
• As before, the sizes of buckets obey the restriction that there are one ortwo of each size, up to some limit. However, we do not assume that thesequence of allowable bucket sizes starts with 1. Rather, they are requiredonly to form a sequence where each size is twice the previous size, e.g.,3, 6, 12, 24, ...
• Bucket sizes are again restrained to be nondecreasing as we go back intime. As in Section 4.6, we can conclude that there will be O(logN)buckets.
• The contents of a bucket consists of:
1. The size of the bucket.
2. The timestamp of the bucket, that is, the most recent point thatcontributes to the bucket. As in Section 4.6, timestamps can berecorded modulo N.
3. A collection of records that represent the clusters into which thepoints of that bucket have been partitioned. These records contain:
(a) The number of points in the cluster.
(b) The centroid or clustroid of the cluster.
(c) Any other parameters necessary to enable us to merge clusters and maintain approximations to the full set of parameters for themerged cluster.

Initializing & Merging Buckets
Our smallest bucket size will be p, a power of 2. Thus, every p stream elements, we create a new bucket, with the most recent p points. The timestamp for thisbucket is the timestamp of the most recent point in

the bucket. We may leaveeach point in a cluster by itself, or we may perform a clustering of these pointsaccording towhatever clustering strategy we have chosen. For instance, if wechoose a k-means algorithm, then (assuming $k < p$) we cluster the points intokclusters by some algorithm.Whatever method we use to cluster initially, we assume it is possible tocompute the centroids or clustroids for the clusters and count the points ineach cluster. This information becomes part of the record for each cluster. We also compute whatever other parameters for the clusters will be needed in themerging process.

Merging Buckets
Following the strategy from Section 4.6, whenever we create a new bucket, weneed to review the sequence of buckets. First, if some bucket has a timestamp that is more than N time units prior to the current time, then nothing of thatbucket is in the window, and we may drop it from the list. Second, we may havecreated three buckets of size p, in which case we must merge the oldest two ofthe three. The merger may create two buckets of size 2p, in which case we may have to merge buckets of increasing sizes, recursively, just as in Section 4.6.To merge two consecutive buckets, we need to do several things:

1. The size of the bucket is twice the sizes of the two buckets being merged.
2. The timestamp for the merged bucket is the timestamp of the more recentof the two consecutive buckets.
3. We must consider whether to merge clusters, and if so, we need to computethe parameters of the merged clusters. We shall elaborate on this part ofthe algorithm by considering several examples of criteria for merging andways to estimate the needed parameters.

Answering Queries
Recall that we assume a query is a request for the clusters of the most recent mpoints in the stream, where $m \leq N$. Because of the strategy we have adoptedof combining buckets as we go back in time, we may not be able to find a setof buckets that covers exactly the last m points. However, if we choose thesmallest set of buckets that cover the last m points, we shall include in thesebuckets no more than the last 2m points. We shall produce, as answer to thequery, the centroids or clustroids of all the points in the selected buckets. Inorder for the result to be a good approximation to the clusters for exactly thelast m points, we must assume that the points between 2m and m+ 1 will nothave radically different statistics from the most recent m points. However, ifthe statistics vary too rapidly, recall from Section 4.6.6 that a more complexbucketing scheme can guarantee that we can find buckets to cover at most thelast $m(1 + ϙ)$ points, for any $ϙ> 0$.Having selected the desired buckets, we pool all their clusters. We then usesome methodology for deciding which clusters to merge. Examples 7.14 and
7.16 are illustrations of two approaches to this merger. For instance, if we arerequired to produce exactly k clusters, then we can merge the clusters with theclosest centroids until we are left with only k clusters.

References:

Anand Rajaraman and Jeff Ullman “Mining of Massive Datasets”, Cambridge University Press

MODULE 6

REAL TIME BIG DATA MODELS

Motivation:

The motivation of this module is to characterize WWW by its massive size and an absolute lack of coordination in its development. One of the most important features of the WWW is the fact that the web users and the content creatures come from diverse backgrounds and have different motivations for using the web.

Learning Objectives:

- Learn about the existence of "spam" and understand the way search results can be manipulated due to term spam.
- Clearly understand the page rank algorithm and its several variants.
- Learn about different memory efficient techniques to execute the traditional FIM algorithms.

Syllabus:

Module	Syllabus	Duration
6	**PageRank Overview,** Efficient computation of PageRank: PageRank Iteration Using MapReduce, Use of Combiners to Consolidate the Result Vector **A Model for Recommendation Systems,** Content-Based Recommendations, Collaborative Filtering **Social Networks as Graphs,** Clustering of Social-Network Graphs, Direct Discovery of Communities in a social graph**.**	10

Learning Objectives:

At the end of this Module, students will possess the following:

- Clearly understand the PageRank algorithm and its several variants.
- Capable of applying the HITS search algorithm.
- Learn about the simple stream based frequent itemset mining method.

6.1 PageRank

We begin with a portion of the history of search engines, in order to motivate the definition of PageRank, 2 a tool for evaluating the importance of Web pages in a way that it is not easy to fool. We introduce the idea of "random surfers," to explain why PageRank is effective. We then introduce the technique of "taxation" or recycling of random surfers, in order to avoid certain Web structuresthat present problems for the simple version of PageRank.

Definition of PageRank

PageRank is a function that assigns a real number to each page in the Web (or at least to that portion of the Web that has been crawled and its links discovered). The intent is that the higher the PageRank of a page, the more "important" it is. There is not one fixed algorithm for assignment of PageRank, and in fact variations on the basic idea can alter the relative PageRank of any two pages. We begin by defining the basic, idealized PageRank, and follow itby modifications that are necessary for dealing with some real-world problems
concerning the structure of the Web. Think of the Web as a directed graph, where pages are the nodes, and there is an arc from page p1 to page p2 if there are one or more links from p1 to p2. Figure is an example of a tiny version of the Web, where there are only four pages. Page A has links to each of the other three pages; page B has links to A and D only; page C has a link only to A, and page D has links to B and C only.

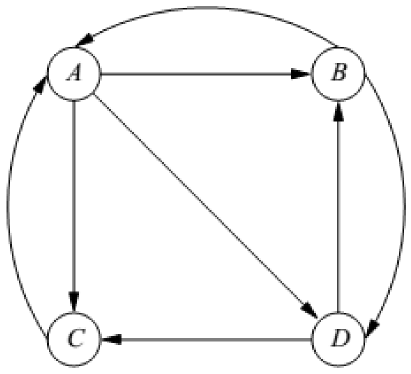


Fig-A hypothetical example of the Web

Suppose a random surfer starts at page A in Fig. 5.1. There are links to B, C, and D, so this surfer will next be at each of those pages with probability 1/3, and has zero probability of being at A. A random surfer at B has, at the next step, probability 1/2 of being at A, 1/2 of being at D, and 0 of being at B or C. In general, we can define the transition matrix of the Web to describe what happens to random surfers after one step. This matrix M has n rows and columns, if there are n pages. The element mij in row i and column j has value 1/k if page j has k arcs out, and one of them is to page i. Otherwise, mij = 0.

Structure of the Web

It would be nice if the Web were strongly connected like Fig.. However, it is not, in practice. An early study of the Web found it to have the structure shown in Fig. 5.2. There was a large strongly connected component (SCC), butthere were several otherportions that were almost as large.

1. The in-component, consisting of pages that could reach the SCC by following links, but were not reachable from the SCC.

2. The out-component, consisting of pages reachable from the SCC but unable to reach the SCC.

3. Tendrils, which are of two types. Some tendrils consist of pages reachable from the in-component but not able to reach the in-component. The other tendrils can reach the out-component, but are not reachable from the out-component.

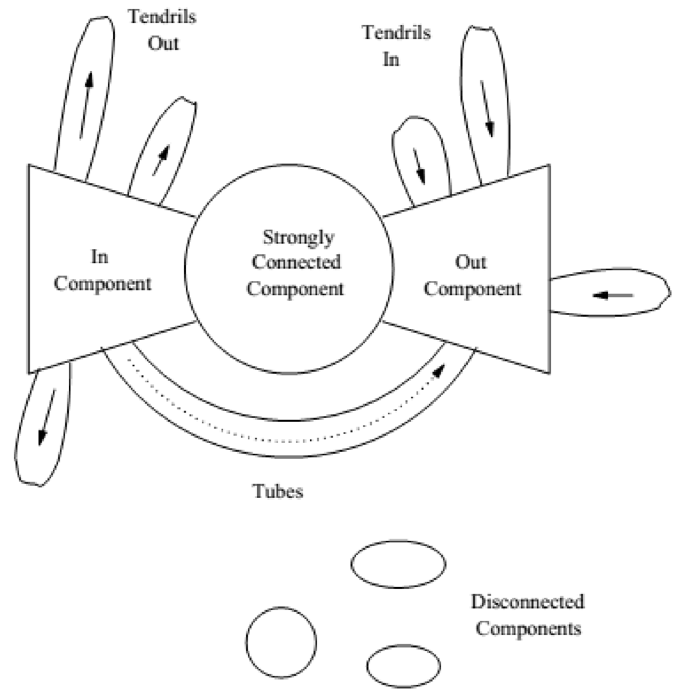


Figure 5.2: The "bowtie" picture of the Web

(a) Tubes, which are pages reachable from the in-component and able to reach the outcomponent, but unable to reach the SCC or be reached from the SCC.
(b) Isolated components that are unreachable from the large components (the SCC, in- and out-components) and unable to reach those components.

Several of these structures violate the assumptions needed for the Markovprocess iteration to converge to a limit. For example, when a random surfer enters the out-component, they can never leave. As a result, surfers starting in either the SCC or in-component are going to wind up in either the outcomponent or a tendril off the in-component. Thus, no page in the SCC or incomponent winds up with any probability of a surfer being there. If we interpret this probability as measuring the importance of a page, then we conclude falsely that nothing in the SCC or in-component is of any importance. As a result, PageRank is usually modified to prevent such anomalies. There are really two problems we need to avoid. First is the dead end, a page that has no links out. Surfers reaching such a page disappear, and the result is that in the limit no page that can reach a dead end can have any PageRank at all. The second problem is groups of pages that all have outlinks but they never link to any other pages. These structures are called spider traps.3 Both theseproblems are solved by a method called "taxation," where we assume a randomsurfer has afinite probability of leaving the Web at any step, and new surfers are started at each page. We shall illustrate this process as we study each of the two problem cases.

Avoiding Dead Ends

Recall that a page with no link out is called a dead end. If we allow dead ends, the transition matrix of the Web is no longer stochastic, since some of the columns will sum to 0 rather than 1. A matrix whose column sums are at most 1 is called substochastic. If we compute Mivfor increasing powers of a substochastic matrix M, then some or all of the components of the vector go to 0. That is, importance “drains out” of the Web, and we get no information about the relative importance of pages.

Example :In Fig. by removing the arc from C to A. Thus, C becomes a dead end. In terms of random surfers, when a surfer reaches C they disappear at the next round.

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

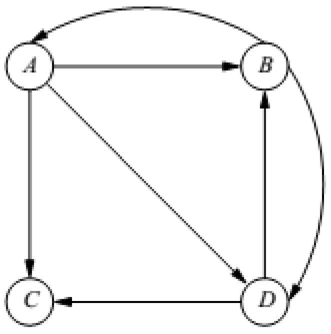


Fig C is now a dead end

There are two approaches to dealing with dead ends.

1. We can drop the dead ends from the graph, and also drop their incoming arcs. Doing so may create more dead ends, which also have to be dropped, recursively. However, eventually we wind up with a strongly-connected component, none of whose nodes are dead ends. In terms of Fig, recursive deletion of dead ends will remove parts of the out-component, tendrils, and tubes, but leave the SCC and the in-component, as well as parts of any small isolated components.4
2. We can modify the process by which random surfers are assumed to move about the Web. This method, which we refer to as “taxation,” also solves the problem of spider traps.

Using PageRank in a Search Engine

Having seen how to calculate the PageRank vector for the portion of the Web that a search engine has crawled, we should examine how this information is used. Each search engine has a secret formula that decides the order in which to show pages to the user in response to a search query consisting of one or more search terms (words). Google is said to use over 250 different properties of pages, from which a linear order of pages is decided.First, in order to be considered for the ranking at all, a page has to have at
least one of the search terms in the query. Normally, the weighting of properties is such that unless all the search terms are present, a page has very little chance of being in the top ten that are normally shown first to the user. Among the qualified pages, a score is computed for each, and an important component of this score is the PageRank of the page. Other components include the presence or absence of search terms in prominent places, such as headers or the links to the page itself.

Efficient Computation of PageRank
To compute the PageRank for a large graph representing the Web, we have to perform a matrix–vector multiplication on the order of 50 times, until the vector is close to unchanged at one iteration. To a first approximation, the MapReduce method given is suitable. However, we must deal with two issues:

1. The transition matrix of the Web M is very sparse. Thus, representing it by all its elements is highly inefficient. Rather, we want to represent the matrix by its nonzero elements.
2. We may not be using MapReduce, or for efficiency reasons we may wish to use a combiner (see Section 2.2.4) with the Map tasks to reduce the amount of data that must be passed from Map tasks to Reduce tasks. In this case, the striping approach discussed is not sufficient to avoid heavy use of disk (thrashing).

PageRank Iteration Using MapReduce
One iteration of the PageRank algorithm involves taking an estimated PageRank vector v and computing the next estimate v′ by $v' = \beta Mv + (1 - \beta)e/n$ Recall β is a constant slightly less than 1, e is a vector of all 1's, and n is the number of nodes in the graph that transition matrix M represents. If n is small enough that each Map task can store the full vector v in main memory and also have room in main memory for the result vector v′, then there
is little more here than a matrix–vector multiplication. The additional steps are to multiply each component of Mv by constant β and to add $(1 - \beta)/n$ to each component. However, it is likely, given the size of the Web today, that v is much too large to fit in main memory. As we discussed in Section 2.3.1, the method of striping, where we break M into vertical stripes (see Fig. 2.4) and break v into corresponding horizontal stripes, will allow us to execute the MapReduceprocess efficiently, with no more of v at any one Map task than can conveniently fit in main memory. Use of Combiners to Consolidate the Result Vector
There are two reasons the method might not be adequate.

1. We might wish to add terms for vi′, the ith component of the result vector v, at the Map tasks. This improvement is the same as using a combiner, since the Reduce function simply adds terms with a common key. Recall that for a MapReduce implementation of matrix–vector multiplication, the key is the value of i for which a term mijvjis intended.
2. We might not be using MapReduce at all, but rather executing the iteration step at a single machine or a collection of machines.

We shall assume that we are trying to implement a combiner in conjunction with a Map task; the second case uses essentially the same idea. Suppose that we are using the stripe method to partition a matrix and vector that do not fit in main memory. Then a vertical stripe from the matrix M and a horizontal stripe from the vector v will contribute to all components of the result vector v′. Since that vector is the same length as v, it will not fit in main memory either. Moreover, as M is stored column-by-column for efficiency reasons, a column can affect any of the components of v′. As a result, it is unlikely that when we need to add a term to some component vi′, that component will already be in main memory. Thus, most terms will require that a page be brought into main memory to add it to the proper component. That situation, called thrashing, takes orders of magnitude too much time to be feasible.

Topic-Sensitive PageRank

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. In the next section, we shall see how the topic-sensitive idea can also be applied to negate the effects of a new kind of spam, called "'link spam," that has developed to try to fool the PageRank algorithm.

Link Spam

When it became apparent that PageRank and other techniques used by Google made term spam ineffective, spammers turned to methods designed to fool the PageRank algorithm into overvaluing certain pages. The techniques for artificially increasing the PageRank of a page are collectively called link spam. In this section we shall first examine how spammers create link spam, and then see several methods for decreasing the effectiveness of these spamming techniques, including TrustRank and measurement of spam mass.

Architecture of a Spam Farm

A collection of pages whose purpose is to increase the PageRank of a certain page or pages is called a spam farm. Figure shows the simplest form of spam farm. From the point of view of the spammer, the Web is divided into three parts:

1. Inaccessible pages: the pages that the spammer cannot affect. Most of the Web is in this part.
2. Accessible pages: those pages that, while they are not controlled by the spammer, can be affected by the spammer.
3. Own pages: the pages that the spammer owns and controls.

Hubs and Authorities

An idea called "hubs and authorities' was proposed shortly after PageRank was first implemented. The algorithm for computing hubs and authorities bears some resemblance to the computation of PageRank, since it also deals with the iterative computation of a fixedpoint involving repeated matrix–vector multiplication. However, there are also significant differences between the two ideas, and neither can substitute for the other. This hubs-and-authorities algorithm, sometimes called HITS (hyperlinkinducedtopic search), was originally intended not as a preprocessing step before handling search queries, as PageRank is, but as a step to be done along with the processing of a search query, to rank only the responses to that query. We shall, however, describe it as a technique for analyzing the entire Web, or the portion crawled by a search engine. There is reason to believe that something like this approach is, in fact, used by the Ask search engine.

The Intuition Behind HITS
While PageRank assumes a one-dimensional notion of importance for pages, HITS views important pages as having two flavors of importance.

1. Certain pages are valuable because they provide information about a topic. These pages are called authorities.
2. Other pages are valuable not because they provide information about any topic, but because they tell you where to go to find out about that topic. These pages are called hubs.

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. Ifthere is a degree associated with the relationship, this degree is represented bylabeling the edges. Often, social graphs are undirected, as for the Facebookfriends graph. But they can be directed graphs, as for example the graphs offollowers on Twitter or Google+.

Clustering of Social- Network Graphs:
An important aspect of social networks is that they contain communities ofentities that are connected by many edges. These typically correspond to groups of friends at school or groups of researchers interested in the same topic, forexample. In this section, we shall consider clustering of the graph as a way toidentify communities. It turns out that the techniques we learned in Chapter 7are generally unsuitable for the problem of clustering social-network graphs.

Direct Discovery of Communities
In the previous section we searched for communities by partitioning all the individualsin a social network. While this approach is relatively efficient, it doeshave several limitations. It is not possible to place an individual in two differentcommunities, and everyone is assigned to a community. In this section, we shallsee 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 largefrequentitemsets, as was discussed in Chapter 6.

Finding Cliques
Our first thought about how we could find sets of nodes with many edgesbetween them is to start by finding a large clique (a set of nodes with edgesbetween any two of them). However, that task is not easy. Not only is findingmaximal cliques NP-complete, but it is among the hardest of the NP-completeproblems 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 betweenthem, and yet have only relatively small cliques.

Complete Bipartite Graphs
Recall our discussion of bipartite graphs from Section 8.3. A complete bipartite graphconsists of s nodes on one side and t nodes on the other side, with all stpossible edges between the nodes of one side and the other present. We denote this graph by Ks,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 graphand denoted Ks, while a complete bipartite subgraph is

sometimes called abi-clique. While as we saw in Example 10.10, 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.1 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 as discussed in Section 10.1.4, 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, in Example 10.2, we could focus on the tag and page nodes of a graph like Fig. 10.2 and try to find communities of tags and Web pages. Such 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.

Finding Complete Bipartite Subgraphs
Suppose we are given a large bipartite graph G , and we want to find instances ofKs,twithin it. It is possible to view the problem of finding instances of Ks,twithinGas 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 Ks,twe are looking for has t nodes on the left side, and we shall also assume for efficiency that t ≤s. The "baskets" correspond to the nodes on the other side of G (the right side). The members of the basket for node vare 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 Ks,thas on the right side. We can now state the problem of finding instances of Ks,tas that of finding frequentitemsetsFof 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 sof the baskets in which all those items appear form an instance of Ks,t.

SimRank &Counting triangles using Map-Reduce:

In this section, we shall take up 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. 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. This conclusion comes from the theory of Markov processes that we mentioned in Section 5.1.2, although the independence from the starting point requires additional conditions besides connectedness that the graph of Fig. 10.21 does satisfy.
10.6.1 Random Walkers on a Social Graph

Recall our view of PageRank in Section 5.1 as reflecting what a “random surfer” would do if they walked on the Web graph. We can similarly 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. 10.2, which we reproduce here as Fig. 10.21. 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. We conclude that, starting atT1, there is a good chance the walker would visitT2, at least initially, and that chance is better than the chance it would visitT3 or T4. It would be interesting if we could infer that tags T1 and T2 are 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
3_ sets of three nodes, or approximately n3/6 sets of three nodes that might be a triangle. The probability of an edge between any two given nodes being added is m/□n
2, or approximately 2m/n2. 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/n2)3 = 8m3/n6. Thus, the expected number of triangles in a graph of nnodes and m randomly selected edges is approximately (8m3/n6)(n3/6) =
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 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 measurethe extent to which a graph looks like a social network.We can also look at communities within a social network. It has beendemonstrated that the age of a community is related to the density of triangles.That is, when a group has just formed, people pull in their like-minded friends,but the number of triangles is relatively small. If A brings in friends B andC, it may well be that B and C do not know each other. As the communitymatures, B and C may interact because of their membership in the community.Thus, there is a good chance that at sometimethe triangle {A,B,C} will becompleted.

References:

Anand Rajaraman and Jeff Ullman “Mining of Massive Datasets”, Cambridge University Press